Conservative From-Point Visibility

Sampsa Gummerus

University of Tampere
Department of Computer Science
Master's Thesis
December 2003

University of Tampere
Department of Computer Science
Sampsa Gummerus: Conservative From-Point Visibility
Master's Thesis, 69 pages
December 2003

Abstract

Visibility determination has been an important part of the computer graphics research for several decades. First studies of the visibility were hidden line removal algorithms, and later hidden surface removal algorithms. Today's visibility determination is mainly concentrated on conservative, object level visibility determination techniques. Conservative methods are used to accelerate the rendering process when some exact visibility determination algorithm is present. The Z-buffer is a typical exact visibility determination algorithm. The Z-buffer algorithm is implemented in practically every modern graphics chip.

This thesis concentrates on a subset of conservative visibility determination techniques. These techniques are sometimes called from-point visibility algorithms. They attempt to estimate a set of visible objects as seen from the current viewpoint. These techniques are typically used with real-time graphics applications such as games and virtual environments. Concentration is on the view frustum culling and occlusion culling. View frustum culling discards objects that are outside of the viewable volume. Occlusion culling algorithms try to identify objects that are not visible because they are behind some other objects. Also spatial data structures behind the efficient implementations of view frustum culling and occlusion culling are reviewed. Spatial data structure techniques like maintaining of dynamic scenes and exploiting spatial and temporal coherences are reviewed.

Keywords: view frustum culling, occlusion culling, occlusion queries, kD-tree, octree, BSP-tree, dynamic scenes, temporal coherence, spatial coherence.

Contents

1. Introduction

The determination of visible objects has been an important part of the computer graphics research for several decades. First studies of visibility were hidden line removal algorithms, and later hidden surface removal algorithms. Today's visibility determination is largely concentrated on more conservative, object level visibility determination techniques. Conservative methods are used to accelerate the rendering process when some exact visibility determination algorithm is present. The Z-buffer [Catm74] is a typical exact visibility determination algorithm. The Z-buffer algorithm is implemented in practically every modern graphics chip.

The computer graphics hardware and rendering techniques have evolved very rapidly during the last decade. In the past few years the capabilities of the graphics hardware have increased faster than the Moore's Law predicts (doubling every 18 months). Regardless of the increased processing power of the graphics hardware, very large scenes can still cause serious performance problems. This is a typical problem in virtual environments and computer games where the viewer is inside the virtual world and only a small part of the world is visible.

In virtual environments, there usually exist hidden objects behind visible objects. The Z-buffer takes care that these objects stay behind in the final image. Regardless of the correct rendering order, the objects are still drawn before the hidden pixels are discarded by the Z-buffer. The processing power is still wasted on rendering these hidden objects. This power could be used to render better-looking graphics.

Visibility in games is traditionally solved using some preprocessing technique. The preprocessed visibility overestimates the set of visible objects from some region. The estimation should be more than actually visible set, but significantly less than the whole scene. At runtime the rendered geometry can be limited to manageable size with the preprocessed visibility information. The region where the viewpoint lies is identified at runtime, and the preprocessed set of visible objects is rendered using the Z-buffer.

Preprocessing visibility techniques are limited to static scenes. The visibility information cannot be updated at runtime because the preprocessing stage is usually very time-consuming [Abra97].

Visibility can be solved also from a viewpoint once per frame. Some from-point algorithms need to preprocess the scene; others do not need that visibility information. Algorithms that do not need the preprocessing stage are usually more capable of handling dynamic scenes.

This thesis concentrates on conservative from-point visibility algorithms that are suitable for dynamic scenes. These algorithms are used with some exact visibility algorithms, such as the Z-buffer algorithm. Their primary function is to accelerate the rendering process by removing the invisible geometry from the rendering pipeline [Möll99].

The visibility problem is more precisely defined along with the related terms and concepts in Chapter 2.

Chapter 3 reviews common spatial data structures which are used to organize the scene. If the organization is hierarchical, the data structure allows a logarithmic search for visible objects. Spatial data structures are the base of any performance optimized visibility algorithm. Chapter 3 also reviews methods to maintain spatial data structures with dynamic scenes. Visibility determination can be accelerated in many cases by exploiting various coherences. Techniques to exploit spatial and temporal coherence are reviewed in Chapter 3.

View frustum culling is a widely used conservative visibility determination method. Practically all virtual environment applications and games use view frustum culling. Chapter 4 reviews view frustum culling related concepts and terms and optimization techniques.

Chapter 5 reviews several well-known occlusion culling algorithms. Occlusion culling algorithms try to find objects that are hidden by other objects. Occlusion culling has traditionally had very little significance in real life applications, merely academic. An exception is Hybrid's dPVS [Aila01] which is a commercial visibility library that implements an occlusion culling system. The practical usability of the reviewed occlusion culling algorithms is also evaluated in Chapter 5.

The latest graphics hardware supports occlusion queries which can be used to replace expensive software implementations. Hardware occlusion queries are now available also at the entry-level graphics cards. Occlusion queries are exposed through common graphics application programming interfaces (APIs), DirectX 9 and OpenGL. Existing occlusion queries are reviewed in Chapter 5.

Chapter 6 concludes the thesis.

2. Visibility Problem

The primary goal of the visibility determination is to find objects and primitives that are visible from the current viewpoint, and render them in correct order. Another important goal is to classify hidden and visible objects as early as possible in order not to waste processing power on objects that are not visible. The process of determining hidden objects is called *hidden surface removal* (HSR).

Visibility algorithms can be divided into two categories: *exact* and *conservative* visibility algorithms. Exact visibility algorithms are such HSR methods that solve the visibility for every pixel and thus guarantee the final image to be correct. The Z-buffer [Catm74] algorithm is an example of an exact visibility algorithm.

The Z-buffer is an image-space algorithm that uses a two-dimensional buffer called Z-buffer. The size of the Z-buffer is same as the frame buffer. The Z-buffer contains the depth value (z-value) for each pixel of the frame buffer. When a new polygon is rendered its pixels are compared to the Z-buffer. If the z-value of a new pixel is smaller than the previous one (it is closer than the previous one), the new z-value is updated to the Z-buffer and the new pixel is updated to the frame buffer. If the z-value of the pixel is greater, the pixel is not updated to the frame buffer and no updates are made to the Z-buffer. When all polygons are drawn, the frame buffer contains only those pixels that are visible to the viewer. The Z-buffer algorithm guarantees the final image to be correct when all objects are opaque. If the scene contains transparent objects, they have to be rendered separately [Möll99].

Because the Z-buffer algorithm solves the visibility problem in image-space, it is independent of the model's geometric presentation. One of the main advantages of the Z-buffer algorithm is its simplicity.

The Z-buffer starts to slow down when many polygons are behind each others. Hidden polygons do not contribute to the final image but they are still processed and finally discarded by the Z-buffer. In these kinds of scene the depth complexity is high.

The term *depth complexity* refers to how many times a pixel is drawn on the frame buffer [Eber00]. The desired depth complexity is 1; that is, each pixel is drawn once. Higher depth complexity slows down the rendering process.

Over the past decade research has focused more on conservative visibility techniques, probably because of the fact that the hardware implementation of the Z-buffer can be found in practically every graphics system.

To decrease the depth complexity and the workload of the rendering pipeline it would be beneficial to remove hidden objects as early as possible. Even if not all of the hidden objects are removed, the workload of the rendering pipeline decreases. The term *culling* is often used for this type of hidden object removal [Möll99].

A conservative HSR algorithm is an algorithm that finds a superset of visible objects. When a rough estimation of the visible objects is found, they are sent normally through the rendering pipeline. The final visibility is solved with some exact visibility algorithm, usually with the Z-buffer. A conservative culling takes place before rasterization, and usually before transformation and lighting, see figure 1.

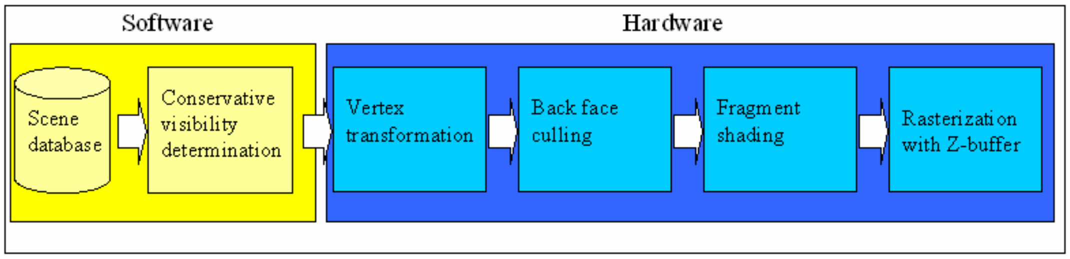

Figure 1. A diagram of the modern graphics pipeline. The scene management and the conservative culling is done on software, before the geometry is sent to the graphics hardware.

The critical requirement of a conservative visibility algorithm is that the conservative approximation of the visible objects is larger or equal to the actually visible set. The conservative approximation should be as close as possible to the actually visible set. The general observation is that the more time is spent on the conservative step, more exact the reported set is. On the other hand, the conservative step should not consume more time than the rendering of all primitives.

Definition 1: The *conservativity* of a visibility algorithm is defined as

$$\frac{R}{V},$$

where R is the amount of visible objects reported by a conservative visibility algorithm and V is the amount of the actually visible objects [Aila01]. It can easily be seen from the equation that the conservativity of 1.0 is the perfect result, and values under 1.0 indicate erroneous non-conservative results.

To be able to speed up the rendering, conservative methods have to solve the set of visible objects faster than what it takes to render otherwise hidden objects. The common criterion of a usable conservative visibility algorithm is output-sensitivity.

Definition 2: A visibility algorithm is called *output-sensitive* if its runtime per frame (excluding any initialization) is $O(n + f(N))$, where N is the number of primitives in the entire model, n is the number of visible primitives and $f(N) < N$. $f(N)$ is the (inevitable) overhead imposed by the model. An output-sensitive visibility calculation algorithm is also called a visibility culling algorithm [Suda96].

The output-sensitivity can be achieved by using hierarchical processing of the scene. Hierarchical data structures enable fast determination of the set of interest. The scene is partitioned into the cells, and the cells are organized into some hierarchical data structure. Hierarchical data structures are usually tree structures. Chapter 3 reviews the most common hierarchical data structures used with the output-sensitive visibility algorithms.

Many visibility algorithms [Aila01, Bitt98, Coor97, Gree93] use *coherences* to improve their performance. A coherence property indicates under what conditions entities cohere, or stick together [Same90b]. In the scope of the visibility algorithms, at least three types of coherences can be used; *object-space coherence*, *image-space coherence* and *temporal coherence* [Bitt01].

The object-space coherence means that the objects that reside spatially near each other are projected near each other onto the view plane [Same90b]. If a group of objects is bound by a volume, and the volume is found to be hidden, objects inside the volume are automatically also hidden.

The image-space coherence means that neighboring pixels on the view plane tend to have the same value (i.e., be projections of the same object) [Same90b].

Temporal coherence exists between successive frames when the movement of the viewer is smooth. For instance, objects that are hidden in one frame are most likely still hidden in the next frame. Temporal coherence is very hard to exploit. If the orientation of the viewer changes very rapidly, it is likely that the temporal coherence is lost.

Methods to solve the visibility problem can be classified in one of the following three categories: *visibility along a line*, *visibility from region* or *visibility from point* [Bitt02].

Ray tracing methods solve visibility along a line. For each pixel a ray is casted from the viewpoint. Ray tracing techniques are commonly used with global illumination methods. Ray tracing is currently not considered as to be a real-time rendering methods and is omitted from this paper. More information about visibility studies along the line can be found in Bittner's PhD dissertion [Bitt02].

Methods that solve the visibility from region are based on the idea that the scene is partitioned into *cells* or regions, and the visibility between cells can be estimated. Cells are connected with *portals* to each other. Visibility information of one cell contains a list of all the other cells that can be viewed from some point in the cell. This information is commonly called as *Potentially Visible Set* (PVS) [Aire90]. The PVS information is calculated as a preprocessing stage. At runtime the cell which contains the viewpoint is identified and the geometry contained by the PVS of the cell is rendered.

Natural environments for PVS based visibility systems are indoor environments where the cell subdivision roughly corresponds to the rooms and portals correspond to the doorways.

Visibility systems based on PVS are usually very fast at runtime but they are limited only to static environments. Dynamic environments cannot utilize the PVS information because when dynamic objects change their location, the PVS information becomes outdated. PVS calculations are typically very time consuming, and cannot be updated at runtime [Abra97].

Memory requirements for the PVS information can increase very rapidly [Bitt02], although a very efficient implementation also exists [Abra97].

Since dynamic scenes are also considered in this paper, methods that solve visibility from region are out of the scope. Different approaches for solving visibility from region can be found for example in the works of Airey [Aire90], Funkhouser [Funk93], Durand [Dura96, Dura97, Dura00] and Luebke [Lueb95].

Most of the modern real-time visibility algorithms solve the visibility using a from-point method. Visible objects are identified every time the viewpoint or the orientation of the viewer changes. In real-time applications such as games, visible objects are typically identified for every frame. This thesis concentrates on algorithms that solve the visibility from point.

The process of solving the visibility from point can be described with a *visibility pipeline* [Aila01]. It contains five steps:

1. View Frustum Culling
2. Occlusion Culling
3. Back Face Culling
4. Exact Visibility Determination
5. Contribution Culling.

In the three-dimensional space the view frustum is a truncated pyramid constructed from six planes. The view frustum describes the visible volume from the current viewpoint. View frustum culling removes objects that are outside of the view frustum. Figure 2 illustrates the idea of view frustum in the two-dimensional space.

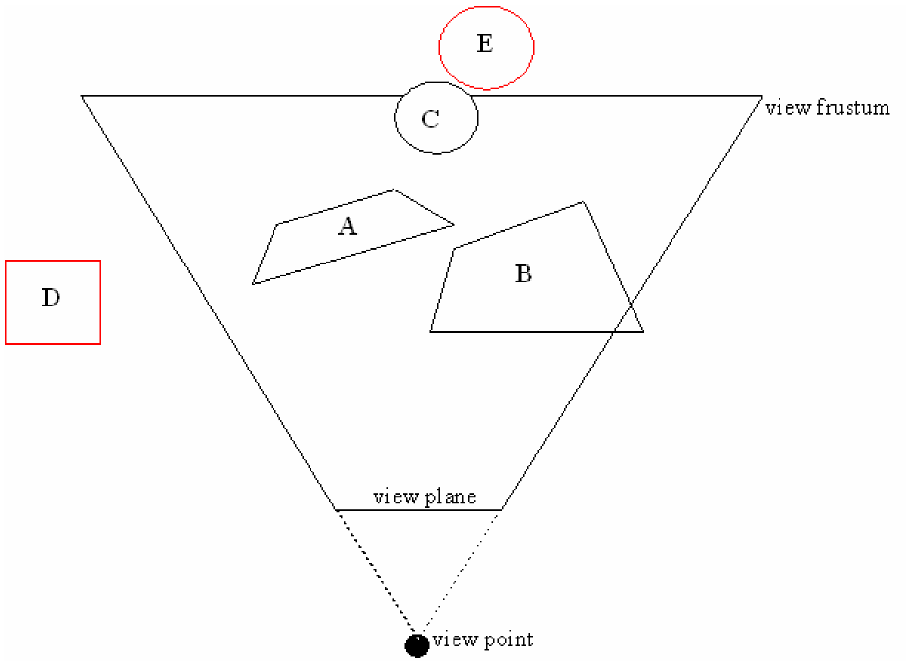

Figure 2. The view frustum limits the viewable volume of the scene space. Objects D and E are culled by view frustum culling.

Some of objects that are inside the view frustum might still be hidden from the current viewpoint. These objects are obstructed by other objects that are in front of the hidden objects. Occlusion culling makes a conservative estimation of hidden objects and removes them from further processing, see figure 3.

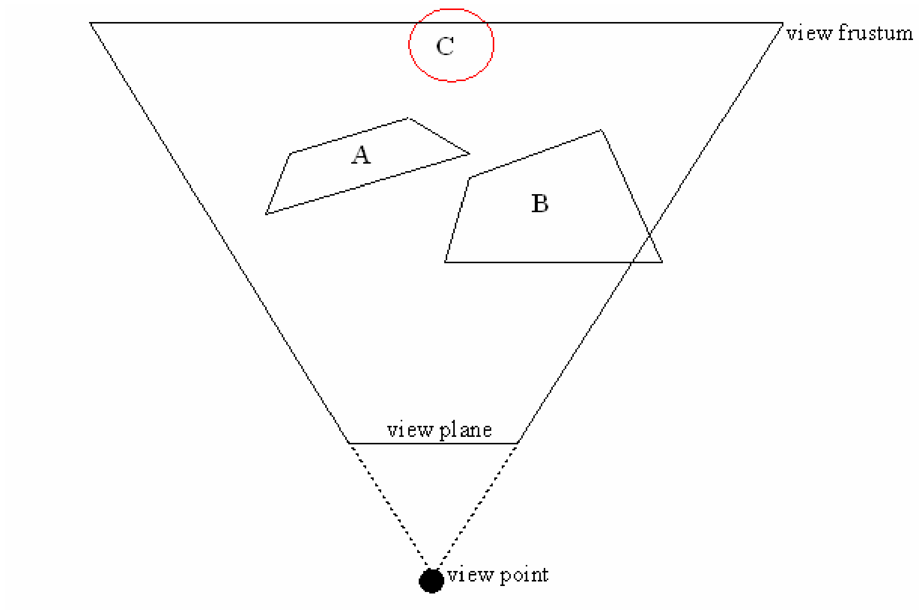

Figure 3. The object C is removed by occlusion culling. Objects A and B obstruct C from the current viewpoint.

Approximately half of the primitives of the visible objects are hidden because they are facing away from the viewer [Möll99]. Back face culling removes polygons that are facing away, see figure 4. Back face culling is implemented in modern graphics APIs like DirectX [Micr03] and OpenGL [Open03]. Back face culling can simply be enabled or disabled while rendering the objects.

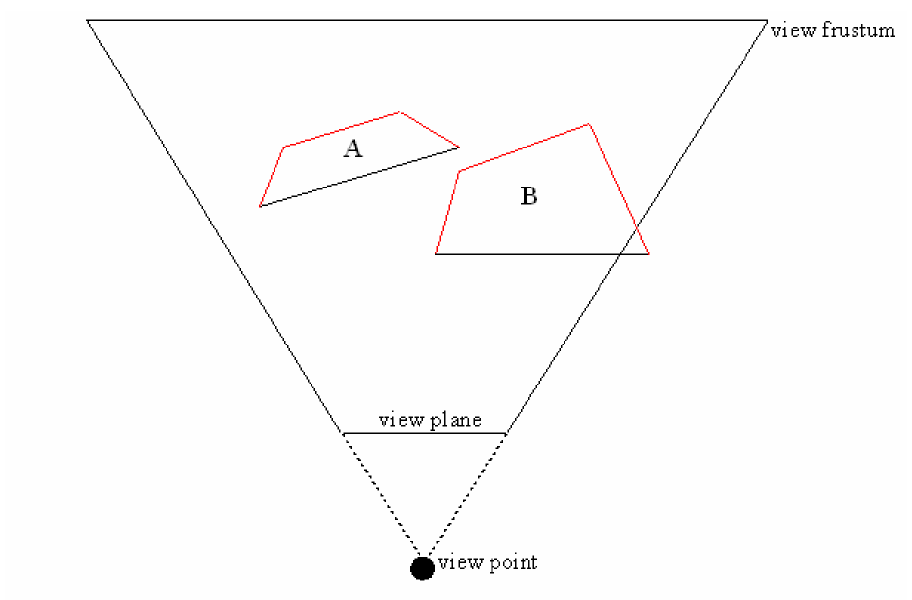

Figure 4. The faces of objects A and B that are facing away from the viewer are culled by back face culling.

The exact visibility determination resolves the visibility for each pixel in the final image. The Z-buffer algorithm is currently the most used exact visibility determination algorithm.

The contribution culling removes objects that would have only minor impact on the final image. Other four sub-categories leave the output image intact, contribution culling introduces artifacts. The term aggressive culling is also used to refer to non-conservative methods.

3. Scene Organization

In order to achieve output-sensitivity, hidden objects must be separated from visible ones with as small number of tests as possible. This can be achieved by processing the scene hierarchically. To process a scene hierarchically, the scene has to be organized into a hierarchical data structure. Data structures that organize multi-dimensional data are commonly called spatial data structures [Same90a]. Not all spatial data structures are hierarchical, such as regular grids, see Section 3.3.

A spatial hierarchy can be understood as a search tree that is used to find objects of interest. In the case of visibility algorithms, objects of interest are the ones that are visible (or alternatively hidden). Using the hierarchical processing of the scene, objects of interest can be found in *O(log n)* time [Same90a].

Hierarchical scene organization can be used for several purposes, like accelerating collision detection, ray tracing and visibility determination. For the purposes of visibility determination, it is important that the data structure provides at least approximate front-to-back order for all objects in the scene from any viewpoint. Occlusion culling algorithms usually benefit greatly if the front-to-back order of the objects is known. The modern graphics hardware also benefits from front-to-back rendering the scene [Rigu02].

3.1. Bounding Volume Hierarchies and Scene Graphs

A bounding volume (BV) is a convex volume that encapsulates an object or a group of objects. The capacity of the BV is thus always greater or equal to capacity of the bounded objects. Bounding volumes are not rendered to the final image; they are used to speed up object-level visibility tests. If the object is more complex than its BV, it is usually beneficial to test the visibility of BV, instead of the object's own geometry. The use of BVs is based on the fact that if the BV is found to be hidden, the object itself is hidden. Bounding volumes are also widely used in other areas of computer graphics, for instance accelerating collision detection and ray tracing. Typical bounding volumes are axis-aligned bounding boxes (AABB), oriented bounding boxes (OBB) and bounding spheres, see figure 5.

An AABB is defined by two points $\vec{p}_{\min} = (x_{\min}, y_{\min}, z_{\min})$ and $\vec{p}_{\max} = (x_{\max}, y_{\max}, z_{\max})$ which describe the minimum and maximum corners the axis-aligned cube [Eber00].

An OBB is defined by the center point $\vec{c}$, three orthogonal axes $\vec{u}_i$, that form a right-handed coordinate system, and three extents $e_i > 0$, where $i = 1,2,3$ [Eber00].

A sphere is defined by a center point $\vec{c}$ and the radius $r > 0$ [Eber00].

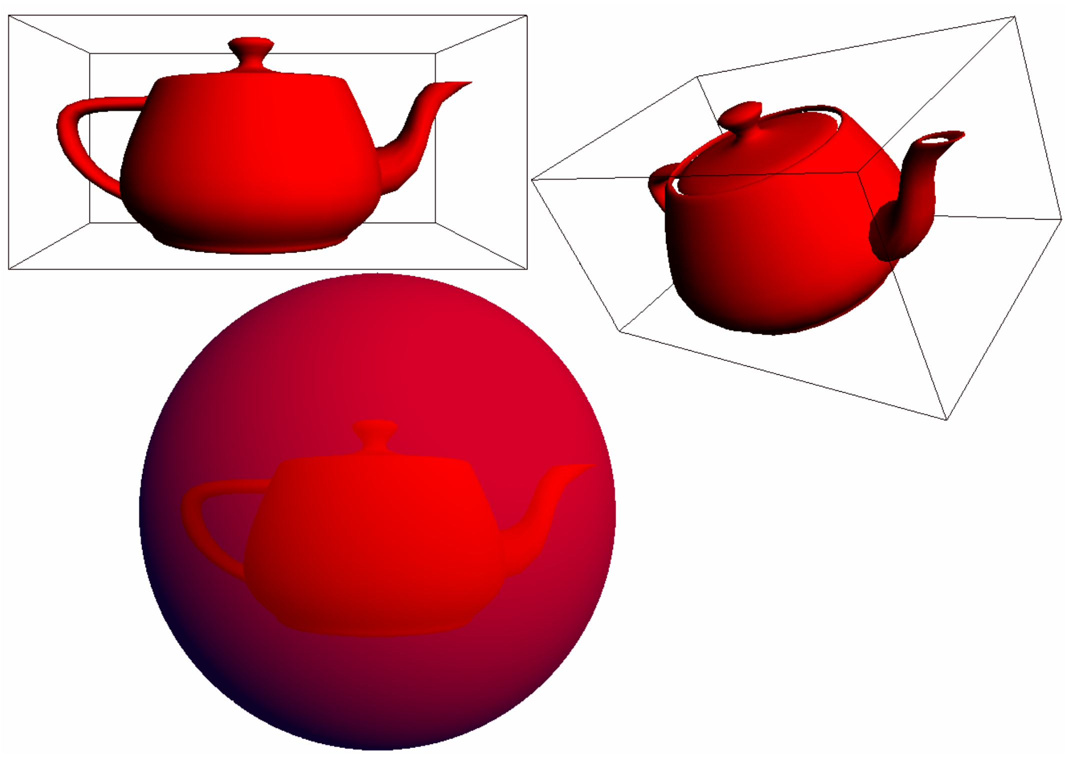

Figure 5. Different bounding volumes of the same object: Top-left: AABB, Top-right: OBB and Bottom: Sphere.

In computer games and virtual reality applications scenes usually contain several objects. If each object in the scene is bounded by a BV, the visibility determination is probably accelerated because of simpler object–level tests but the visibility determination process is not output-sensitive. If objects are near to each other, their BVs can be grouped inside one larger BV. This way a hierarchy of bounding volumes can be built to contain the entire scene. For each frame the hierarchy is traversed in a top-down manner. If an inner node is determined as hidden all child nodes can be also determined as hidden without further testing.

Another way to build a bounding volume hierarchy is from top to down. First the entire scene is bounded by one bounding volume which corresponds to the root of the hierarchy. After this the scene is recursively decomposed into subsets that are bounded by smaller bounding volumes. Recursion is terminated when the minimum number of objects inside one BV is reached, or some predefined threshold for the size of the hierarchy is reached. It is very important that the BVs in the hierarchy entirely contain their child nodes; otherwise nothing can be said about the visibility of the child nodes. Commonly used BV-hierarchies are AABB-tree, OBB-tree and sphere-tree, according to the related bounding volume. Figure 6 illustrates a scene decomposed by a sphere-tree. The number of children a node has can be fixed, or decided when the hierarchy is built.

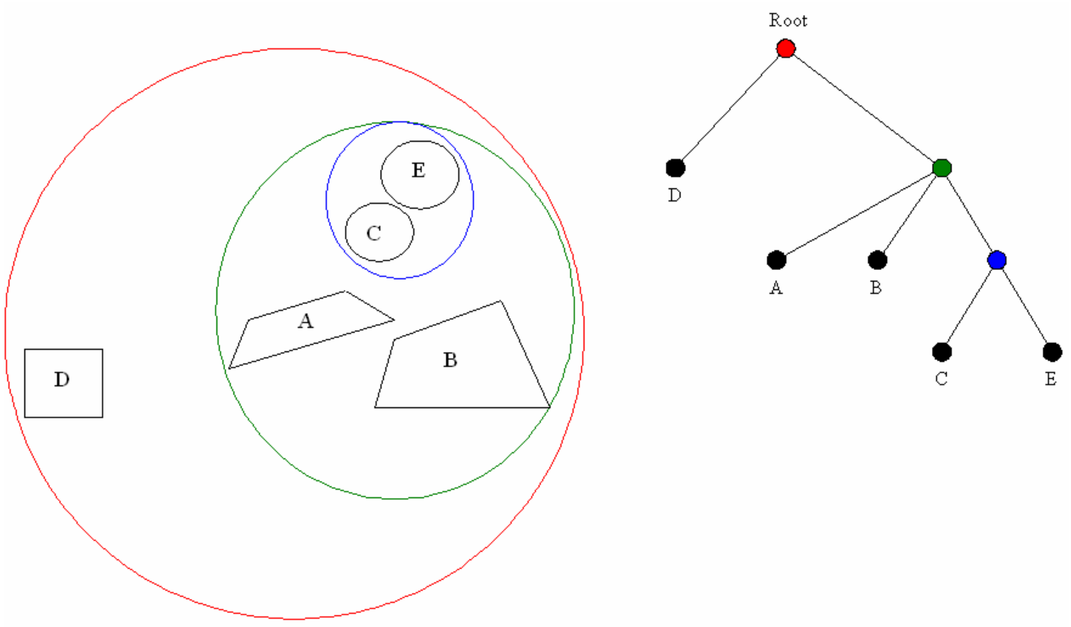

Figure 6. A Bounding volume (sphere) -hierarchy, and the corresponding tree diagram.

A *scene graph* is a *directed acyclic graph* (DAG) that organizes and stores all of the data needed to render a virtual scene. Usually a scene graph is just a tree-structured database containing a hierarchy of branches and leaf nodes. The root node of the scene graph represents the whole scene. The leaves contain the geometric data of the objects in the scene. The inner nodes provide the grouping mechanism for objects, and other information needed to render the scene. The scene graph groups objects that are spatially and/or semantically close to each other.

In theory, the DAG structure can be more complicated structure than a tree. Inner nodes can contain references to other branches of the DAG. The DAG structure supports higher level sharing of objects. However, the memory costs and code complexity to maintain such a graph do not justify using it [Eber00].

For the visibility determination purposes the scene graph is essentially a BV-hierarchy. The bounding volume hierarchy can be used with view frustum culling. If the bounding volume of a node is outside the view frustum then the child nodes must also be outside the view frustum, and no culling tests need to be done on the children [Eber00].

Many higher level APIs such as Java3D [Java03] and VRML [VRML03] use a scene graph to organize the scene. Scene graph based APIs usually perform view frustum culling but do not perform occlusion culling. Scene graphs' and BV-hierarchies' lack of proximity information prevents them to be used with efficient occlusion culling. If the depth complexity of the scene is low, i.e. most of the objects within the view frustum are visible; a BV-hierarchy and view frustum culling alone may be efficient enough for the conservative visibility step.

3.2. Spatial Data Structures

Even though scene graph structure supports view frustum culling, it is not efficient data structure for the HSR algorithms. The scene graph structure does not contain proximity information which makes it impossible to efficiently find objects located spatially close to each other [Aila01]. The front-to-back traversal of the scene inside the view frustum is critical to efficient occlusion culling algorithms. This section reviews commonly used spatial data structures that provide the proximity information needed to perform occlusion culling.

Spatial data structures used in computer graphics can roughly be divided into two categories depending on whether they contain the actual geometry of the scene, or they are used as auxiliary data structures that partition the space. BSP-trees can be used in both ways. Octrees and kD-trees are typically used as an auxiliary data structures.

Occlusion culling algorithms generally use some spatial data structure as an auxiliary data structure to partition the scene space. The spatial partition of the space is a discrete data structure that spans the continuous space, mapping every object in the scene onto one or more of the finite number of cells. If an object intersects with multiple cells, the object can be either split or mapped to all intersecting cells. If the object is split, new objects are mapped to corresponding cells.

Mapped objects can be logical objects of the virtual world, polygons, vertices or other rendering primitives [Same90a]. If the scene contains mainly static objects, it might be efficient to map every polygon to the spatial database. On the other hand, if the scene contains several dynamic objects, mapping may be more efficient if it is done on the object level.

When a spatial data structure is used to solve the visibility problem, typical operations of the spatial data structure are construction, traversal and possibly updating. If the scene is fully static, updates are not needed. During the construction time objects are inserted in the spatial data structure.

The insertion of an object does not necessarily mean that the object is stored in the database. Auxiliary spatial structures usually store only a reference to the actual objects and additional information needed for visibility calculations, typically a BV [Aila01].

During the traversal of the spatial data structure visible objects are determined. View frustum culling is used to find objects that are inside the view frustum. Occlusion culling further refines the visibility of the objects and removes any object that is hidden by other objects. If the scene contains moving objects, their spatial locations have to be updated to reflect their new location.

3.3. Regular Grids

The regular grid method is probably the simplest way to subdivide multidimensional space. It divides the space into equal-sized cells. Cells are often referred as buckets. Each bucket contains a list of objects that lie inside the cell. The regular grid method is essentially a directory in the form of a k-dimensional array. Regular grids are sometimes called as fixed-grids or fixed-cells [Same90a].

The grid structure can be relatively efficient if the data is uniformly distributed over the space. Non-uniform distribution causes inefficient because buckets may unevenly be filled leading to unnecessary testing of empty or nearly empty cells [Same90a]. Figure 7 illustrates a two-dimensional scene subdivided by a regular grid.

The regular grid structure does not provide logarithmic search property which is basically the foundation of output sensitive visibility determination. Only few algorithms [Bata01, Hill02] are based on the grid structure.

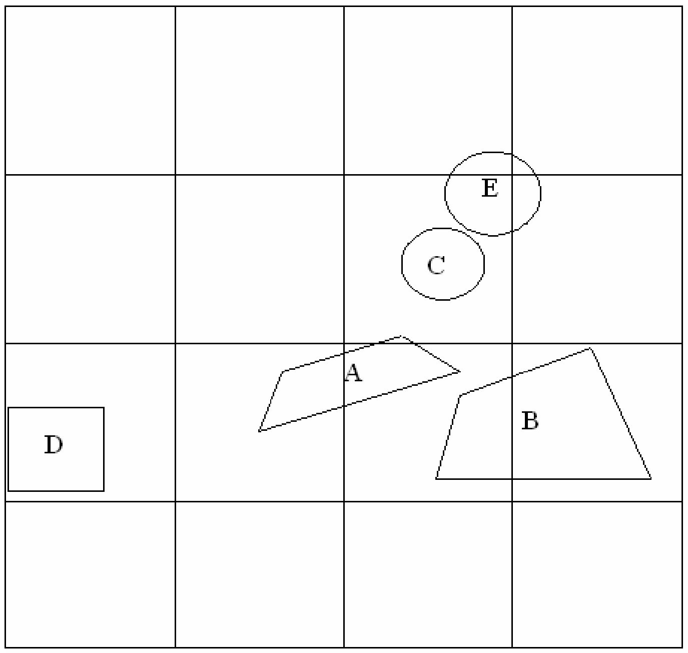

Figure 7. A two-dimensional scene subdivided by the regular grid.

3.4. Quadtrees and Octrees

The octree data structure is often used with conservative hidden surface removal methods [Gree93, Gree96, Suda98]. It belongs to the data structure family known as quadtrees.

The term quadtree is used to describe a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space. They can be differentiated on the following bases:

1. The type of data they are used to represent.
2. The principle guiding the decomposition process.
3. The resolution (variable or not) [Same90a].

The term quadtree is typically used to describe data structures that decompose two-dimensional space (usually images). Figure 8 illustrates a two-dimensional scene subdivided by a quadtree. The quadtree recursively decomposes two-dimensional square-area to four quadrants. The square is axis-aligned and each side is of equal length. Every node in a quadtree has four or zero children (leaves have no children). Recursion is terminated when some predefined criterion is reached, for example minimum number of objects in a quadrant, or maximum number of subdivisions.

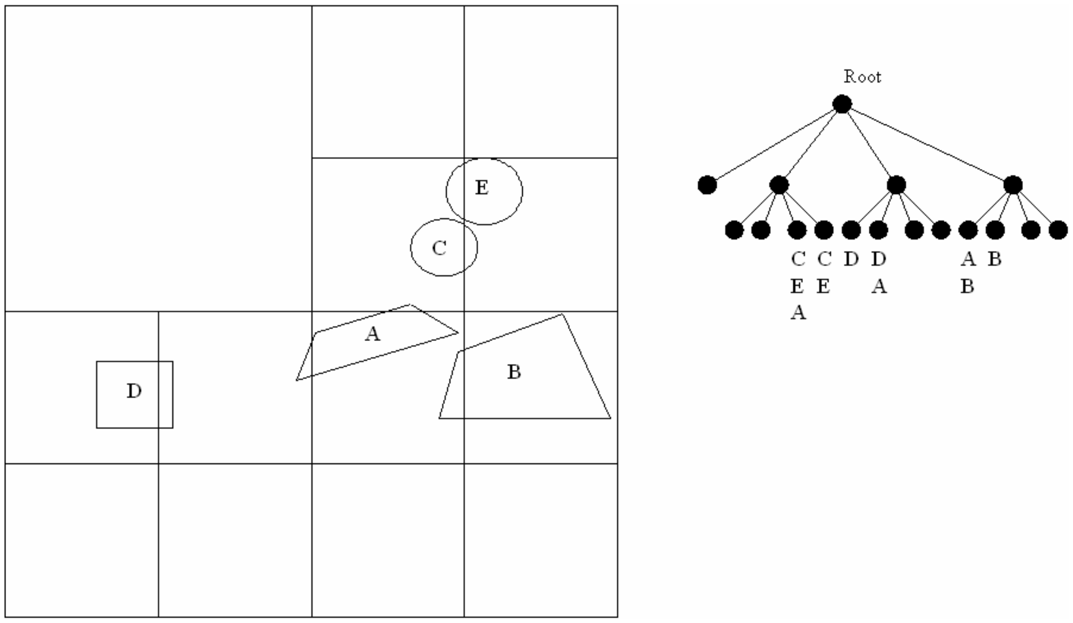

Figure 8. A two-dimensional scene subdivided by the quadtree. Under the leaf-nodes are the objects that intersect with the cell.

The octree data structure is a three-dimensional variant of the quadtree. An octree recursively decomposes three-dimensional space, shape of a cube, to eight sub-cubes. Octree nodes have eight or zero child nodes.

Quadtrees and octrees can be used with several kinds of multidimensional data, for example, with point data, areas, curves, surfaces and volumes. The decomposition may result into equal parts on each level (i.e. regular decomposition), or it may be governed by the input. The resolution of the decomposition (i.e. the number of times that the decomposition process is applied) may be fixed beforehand, or it may be governed by the properties of the input data [Same90a].

Polygons and objects are usually stored using buckets. This kind of quadtree is called a region quadtree, and its three-dimensional variant is called a region octree.

The region octree usually uses regular decomposition. If the subdivision point of the node is selected depending on the distribution of the objects, the decomposition is irregular. The irregular decomposition has the potential of requiring less space. Its drawback is that the determination of optimal partition points may be computationally expensive. A closely related problem, decomposing a region into a minimum number of rectangles, is known to be NP-complete if the region is permitted to contain holes [Same90a].

Two-dimensional quadtrees are sometimes used with terrain rendering [Lars03, Paja98, Zhao01]. Octrees are typically region octrees with regular decomposition [Gree93, Suda98]. The region octree is the simplest variant of the octree data structure. It is usually the one that requires the most space.

To reduce the construction time and the memory requirements, the decomposition can be terminated on object level, instead of rendering primitive level. This way, inserted objects can be BVs and not the primitives of the objects. Figure 9 illustrates a scene with three objects, and an octree that decomposes the scene space.

Each octree node typically contains eight child pointers, one parent pointer, extents of the cube, and a list of pointers to the content of the cell. Every octree node has either zero or eight child nodes. The parent pointer might be useful if the upward traversal is needed. Extents of the cell can be described with a center point and the extent. Alternative way to describe the extents of the cell is the same as AABBs, with two points that describe the minimum and maximum corners the cube. The content list of the cell might not be needed in the inner nodes if the actual geometry is stored only in the leaf nodes.

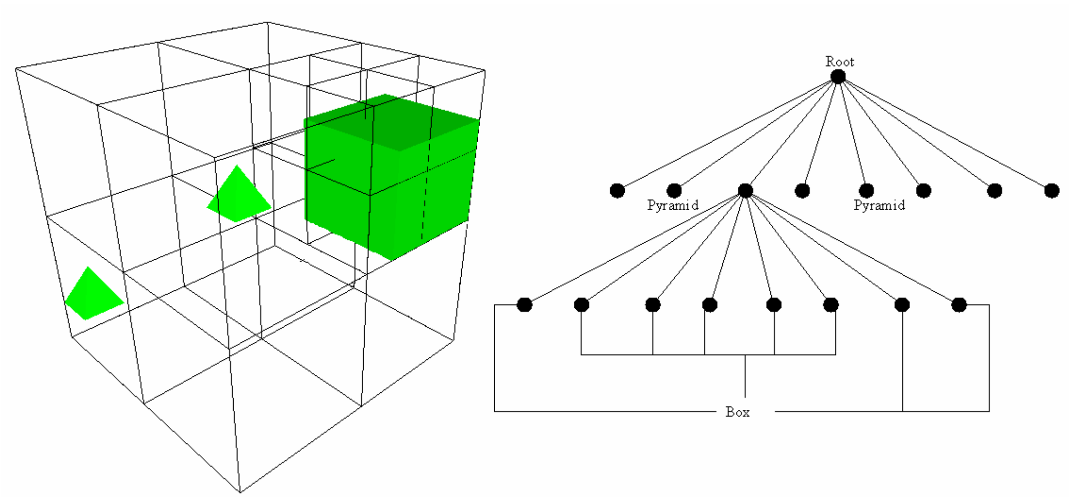

Figure 9. A three-dimensional scene subdivided by the octree and the corresponding tree diagram.

Construction

The first step in the construction of an octree is to define the spatial extent of the scene. The scene is bounded by a rectangular axis-aligned cube whose sides are of equal length. The length of the sides is the maximum of spatial extents of the scene.

The second step is to divide the space recursively into eight subcubes. The subcubes are congruent with one another. Objects of the scene are divided into each new cell. If an object intersects with the boundaries of multiple cells, the object is either split or associated with all intersecting cells. Splitting an object to smaller objects usually increases the polygon count, and is essentially against what is the purpose of the HSR process.

The second step is repeated recursively until a predefined count of objects in a cell is reached. An alternative criterion to terminate the recursion could be the height of the octree.

When a regular decomposition method is used, the subdivision point is the center of the parent cube. Alternatively irregular composition can be used. During each subdivision step the division point has to be found. The subdivision should divide the cell so that each child cell will contain roughly the same number of objects. On the other hand, object splitting should be minimized. This can become quite expensive, because costs have to be evaluated along each axis.

Each node of the octree contains an array of objects in the cell. If the object information in inner nodes is not needed, object arrays can be stored only in the leaves.

Front-to-Back Traversal

Given that the octree is constructed as described above, the octree must be traversed to carry out the visibility calculations. The objective is to find all visible objects from the current viewpoint. The view frustum limits the viewable volume of the space. Cells that are inside the view frustum or intersect with the view frustum are classified as visible by view frustum culling. Occlusion culling can be used to further refine the visibility. Visible cells should be processed in front-to-back order so that the potential occluders could be found first. Occluders are used to identify objects that are obstructed by occluders.

The octree traversal begins from the root and continues recursively toward the leaves testing all eight subnodes. The subnodes are tested in front-to-back order. The order can be calculated by comparing the distances of the center points of the subnodes with the current viewpoint.

An approximate front-to-back order of subnodes can be also determined by checking on which side the viewpoint is compared to three planes that divide the axis-aligned node. The subdivision point lies on all three planes. Each plane is parallel to one of the axes.

In approximate front-to-back order, the closest child node (cube) is first visited, then three cubes that share a face with the first one, then three cubes that share a face with the farthest, and last the farthest cube. Three cubes after the first cube and three cubes before the last cube can be visited in any order. The approximate order can be precalculated for back-to-front [Fole90] and front-to-back order.

The position of the viewpoint related to three planes that subdivide the cube can also be expressed with three sign bits. If one of the bits is toggled, the result is one of nodes that share the face with the node that original bits represent. If all three bits are toggled, the result points to the far corner of the octree. See figure 10 for front-to-back traversal of the octree.

The visibility test can classify a node as visible, hidden or partially visible. If a node is classified as fully visible, the recursion can be terminated, and the corresponding subtree can be rendered. Also if a node is classified as fully hidden, the corresponding subtree can be culled without further testing. If a node is determined as partially visible, the visibility of the corresponding subtree is determined by recursively testing its nodes until all nodes are classified as fully visible or hidden, or leaf nodes are reached. If the traversal reaches leaf nodes, the visibility of objects inside leaf nodes is determined.

The result of the octree traversal is the list of visible objects in the scene. Objects that were determined as partially visible during the traversal are usually treated the same way as fully visible objects. Depending on the implementation, visible objects may be rendered during the traversal as they are found, or a list of visible objects may be reported after the traversal [Aila01].

```
void Octree::Node::TraverseFTB(Point& eye)
{
        if(m_Children[0] != NULL)
        {
                int iFirst =     ((eye.X < m_Center.X) ? 1 : 0) |
                                 ((eye.Y < m_Center.Y) ? 2 : 0) |
                                 ((eye.Z < m_Center.Z) ? 4 : 0) );

                m_Children[iFirst]->TraverseFTB(eye);

                m_Children[iFirst^1] ->TraverseFTB(eye); //Toggle bits 0
                m_Children[iFirst^2] ->TraverseFTB(eye); //Toggle bits 1
                m_Children[iFirst^4] ->TraverseFTB(eye); //Toggle bits 2

                m_Children[iFirst^3] ->TraverseFTB(eye); //Toggle bits 0,1
                m_Children[iFirst^5] ->TraverseFTB(eye); //Toggle bits 0,2
                m_Children[iFirst^6] ->TraverseFTB(eye); //Toggle bits 1,2

                m_Children[iFirst^7] ->TraverseFTB(eye); // Toggle bits 0,1,2
        }
        else //Leaf node
        {
                Render();
        }
}
```

Figure 10. The octree front-to-back traversal algorithm.

3.5. KD-Trees

The kD-tree data structure is a binary search tree generalized for a k-dimensional search space, invented by Bentley [Bent75]. Each node of the kD-tree contains zero or two child pointers, i.e., the size of the node is independent of the number of the dimensions of the search space.

The kD-tree data structure was originally developed for point data, later it has also been used for regional data [Bitt98, Aila01, Coor97]. The term axis-aligned BSP-tree is also used of kD-tree that stores regional data [Möll99].

Each node in the kD-tree represents a rectilinear parallelepiped whose faces are aligned with axes but whose sides do not have to be of equal length. The decomposition is typically not regular. The node contains an axis-aligned hyper plane that cuts its region in two parts. The two regions are the regions of the child nodes. The partitioning plane is always parallel with one of the axes.

Perhaps the biggest difference between the kD-tree and the octree is that the kD-tree is binary tree. Each parent node has precisely two children. Also the non-regular decomposition can provide more fitting cells. Figure 11 illustrates a two-dimensional scene that is partitioned by a kD-tree.

Each kD-tree node typically contains two child pointers, one parent pointer, extents of the cube, and a list of pointers to the content of the cell.

The parent pointer might be useful if the upward traversal is needed. Extents of the cell can be described the same way as AABBs, with two vectors that describe the minimum and the maximum corners of the axis-aligned cube. The content list of the cell might not be needed in inner nodes if the actual geometry is stored only in the leaf nodes.

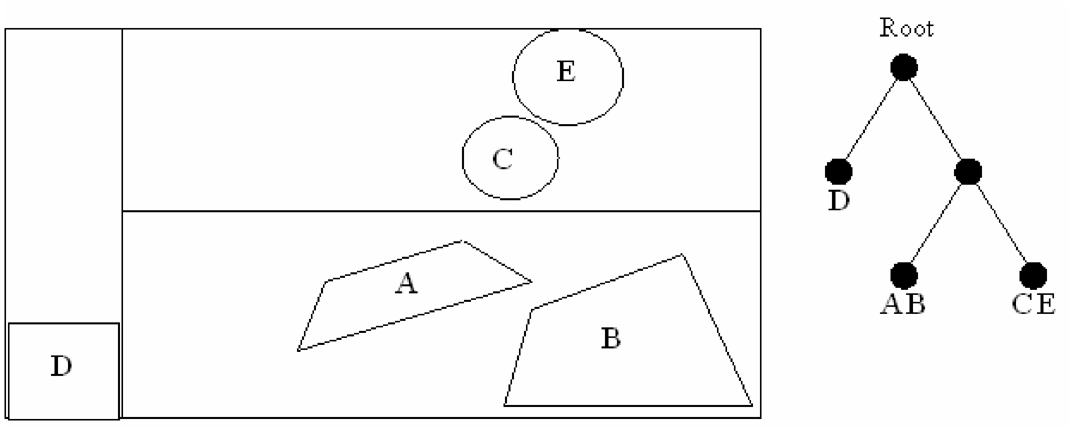

Figure 11. A two-dimensional scene subdivided by the kD-tree.

Construction

The first step in the construction of a kD-tree is as with octrees, to define the spatial extents of the scene. The kD-tree approach is to make the problem space a rectangular parallelepiped whose sides are, in general, of unequal length. The length of the sides is the maximum spatial extent of the scene in each spatial dimension. For example, in three-dimensional space the cell is a parallelepiped whose sides are the greatest separation of objects in each of the three spatial dimensions.

The second step is to subdivide the space recursively. The space is divided into two parts along one of axes so that both new cells contain roughly the same number of objects. The right child node of a kD-tree node is called the positive child node, and the left child node is called the negative child node. All objects that are on the positive side of the splitting plane are associated with the right child node and objects that are on the negative side are associated with the left child node.

In the original algorithm by Bentley [Bent75] the partitioning plane was chosen by alternating through the dimensions of the space in predefined order, for example x, y, z and again x, and so on.

In the case of the regional data, an alternative method is to independently decide which axis the splitting plane is perpendicular to. This method probably results more fitting cells. The splitting plane of a node can be selected by identifying the axis with the largest extent of the node's parallelepiped, and search the plane perpendicular to that selected axis [Bitt98, Aila01].

In both cases, the goal in selecting the splitting plane is to minimize the number of objects cut by the plane, and to keep the tree balanced. The selected plane equally divides the objects into two sets (the median cut plane). The plane can be found by identifying the boundaries of object bounding boxes located within a certain distance from the spatial median of the node's parallelepiped. Each identified boundary induces one bounding plane. The object splits by each boundary plane are evaluated, and the boundary plane with the lowest number of split objects is selected as the splitting plane [Bitt98, Aila01].

This process is repeated recursively until a predefined count of objects in a cell is reached. An alternative criterion to terminate the recursion could be the height of the kD-tree.

The actual geometry of the scene is typically stored in the leaf nodes using buckets.

Front-to-Back Traversal

At runtime the kD-tree is traversed to carry out the visibility calculations. As with an octree the objective is to find all visible objects from the current viewpoint. The view frustum limits the viewable volume of the space. Cells that are inside the view frustum or intersect with the view frustum are classified as visible by view frustum culling. Visible cells should be processed in front-to-back order so that the potential occluders could be found first. Occlusion culling uses occluders to identify objects that are occluded by occluders.

The kD-tree traversal begins from the root and continues recursively toward the leaves testing both child nodes. Child nodes are tested in front-to-back order, see figure 12. The order can be determined by calculating the dot product of the viewpoint and the normal of the plane that separates child nodes. The result of the dot product is the distance of the viewpoint from the plane. If the distance is positive the child nearest to the viewer is the one that is on the positive side of the plane. Otherwise the nearest child is the one on the negative side.

The visibility test can classify a node as visible, hidden or partially visible. The subtree of a visible node is also fully visible and the recursion can be terminated. Also the subtree of a hidden node is fully hidden and can be culled without further testing. If a node is determined as partially visible, the visibility of the corresponding subtree is determined by recursively testing its nodes until all nodes are classified as fully visible or hidden, or leaf nodes are reached. If the traversal reaches leaf nodes, the visibility of objects inside leaf nodes is determined.

The result of the kD-tree traversal is the list of visible objects in the scene. Objects that were determined as partially visible during the traversal are usually treated the same way as fully visible objects. Depending on the implementation, visible objects may be rendered during the traversal as they are found, or visible objects may be returned as a list after the traversal.

```
void KDTree::Node::TraverseFTB(Point& eye)
{
        if(m_Children[0] != NULL)
        {
                int iNear = (m_SplitPlane.Dot(eye) >= 0.0);
                m_Children[iNear]->TraverseFTB(eye);
                m_Children[iNear^1]->TraverseFTB(eye);
        }
        else // Leaf node
        {
                Render();
        }
}
```

Figure 12. The kD-tree front-to-back traversal algorithm.

3.6. BSP-Trees

The *Binary Space Partitioning tree* (BSP-tree) [Fuch80] data structure is many ways similar to kD-tree; it is a binary search tree generalized for a k-dimensional search space. The main difference is that the subdividing hyper planes are arbitrarily oriented in a BSP-tree, see figure 13. The BSP-tree is always a binary tree, regardless of the dimension of the search space.

The dimension of the subdividing hyper plane depends on the dimension of the search space. For k-dimensional search space, the dimension of the hyper plane is k-1. In three-dimensional space subdivisions are made by two-dimensional plane. In two-dimensional space, subdivisions are made by one-dimensional lines. BSP-trees are typically used with two and three-dimensional polygonal data [Abra97, Same90b].

The right child node of a BSP-tree node is called the positive child node, and the left child node is called the negative child node. If all vertices of a polygon satisfy the equation $Ax + By + Cz \geq 0$, then the polygon is on the positive side of the subdivision plane. If none of the vertices of a polygon satisfy the equation above, the polygon is on the negative side of the partitioning plane. If a polygon intersects with the plane, the polygon is either split in two new polygons along the partitioning plane, or the polygon is associated with both child nodes. Partitioning planes end when they intersect their parent node's planes.

Each node in the BSP-tree represents a convex volume. The volume can be of arbitrary shape but it is always convex. The actual geometry of the scene is typically stored in the leaf nodes. Alternatively polygons can be stored in inner nodes, if partitioning planes are chosen to be coplanar with polygons. All polygons that lie on the partitioning plane are stored in the content list of the current node [Nayl98].

Each BSP-tree node typically contains two child pointers, one parent pointer and a list of pointers to the content of the cell. If the node is a leaf node, it contains the extents of the cell. If the node is an internal node, it contains the splitting plane.

The parent pointer might be useful if the upward traversal is needed. The content list of the cell might not be needed in the inner nodes if the actual geometry is stored only in the leaf nodes.

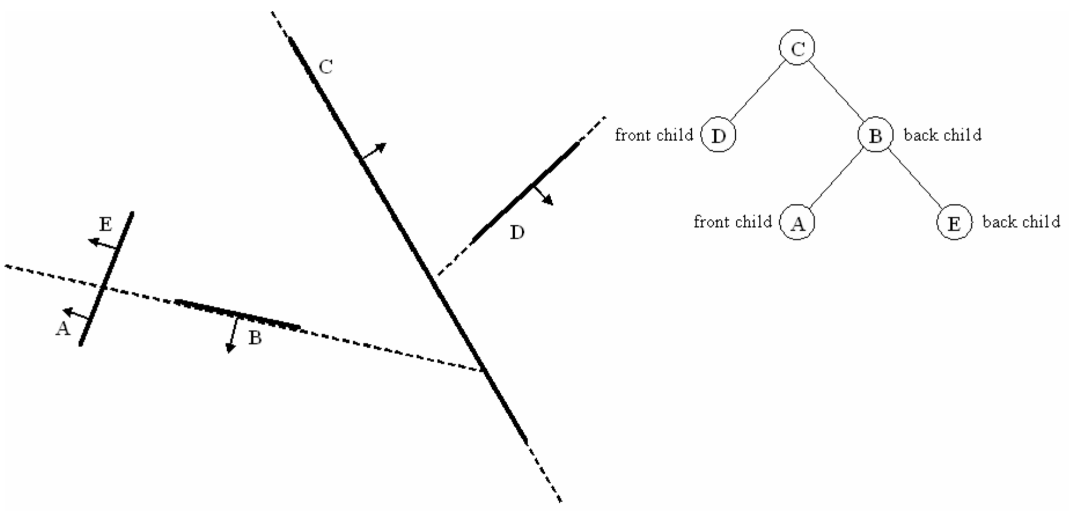

Figure 13. A two-dimensional scene contains four walls (A, B, C, D). The wall A is split by the BSP-tree decomposition to two new walls A, E.

Construction

A BSP-tree for a set of polygons is constructed by first selecting a splitting plane and then partitioning the polygons with the selected plane. The partitioning plane divides polygons into two sets which are associated with corresponding negative and positive child nodes. The current node is associated with the partitioning plane. This process is recursively repeated to both child nodes. The recursion is terminated by some predefined criteria, for example minimum number of polygons per node.

If the partitioning plane intersects a polygon, the polygon is divided into two new polygons that are inserted into corresponding child nodes. Alternatively the polygon can be associated with both child nodes without splitting it. If the splitting planes are selected so that one or more polygons lie on the plane, the polygons can be stored also in inner nodes [Nayl98].

Constructing a good BSP-tree is not an easy task. The primary goal is to get the shortest possible path to the objects of interest. Usually a good BSP-tree is as balanced as possible, splits as few polygons as possible and is as small as possible [Abra97].

Often used heuristics to build good BSP-trees are [Abra97, Nayl93]:

- Polygon aligned splitting planes.
- Large polygons are selected first.
- Splitting planes that split polygons as few as possible.
- Splitting planes that evenly subdivide the data.
- Randomly selected splitting planes.

Front-to-Back Traversal

During the traversal of the BSP-tree all visible objects are determined. The view frustum limits the viewable volume of the space. Cells that are inside the view frustum or intersect with the view frustum are classified as visible by view frustum culling. Visible cells should be processed in front-to-back order so that the potential occluders could be found first. Occlusion culling uses occluders to identify objects that are occluded by occluders.

The BSP-tree traversal begins from the root and continues recursively toward the leaves testing both child nodes. Child nodes are tested in front-to-back order, see figure 14. The order can be determined by calculating the dot product of the viewpoint and the normal of the plane that separates child nodes. The result of the dot product is the signed distance of the viewpoint from the plane. If the distance is positive the child nearest to the viewer is the one that is on the positive side of the plane; i.e. the right child node. Otherwise the nearest child is the one on the negative side; i.e. the left child node. If the BSP-tree stores polygons also in the inner nodes, the current node is processed after the near child node, and before the far child node.

The visibility test can classify a node as visible, hidden or partially visible. The subtree of a visible node is also fully visible and the recursion can be terminated. Also the subtree of a hidden node is fully hidden and can be culled without further testing. If a node is determined as partially visible, the visibility of the corresponding subtree is determined by recursively testing its nodes until all nodes are classified as fully visible or hidden, or leaf nodes are reached. If the traversal reaches leaf nodes, the visibility of objects inside leaf nodes is determined.

The result of the BSP-tree traversal is the list of visible polygons. Objects that were determined as partially visible during the traversal are usually treated the same way as fully visible objects. Depending on the implementation, visible objects may be rendered during the traversal as they are found, or visible objects may be returned as a list after the traversal.

```
void BSPTree::Node::TraverseFTB(Point& eye)
{
        if(m_Children[0] != NULL)
        {
                int iNear = (m_SplitPlane.Dot(eye) >= 0.0);
                m_Children[iNear]->TraverseFTB(eye);
                //If inner nodes contain geometry, they are rendered
                // here, before entering to far child node.
                m_Children[iNear^1]->TraverseFTB(eye);
        }
        else // Leaf node
        {
                Render();
        }
}
```

Figure 14. BSP-tree front-to-back traversal algorithm.

3.7. Exploiting Spatial and Temporal Coherence

The importance of exploiting various types of coherences in designing efficient visibility algorithms is recognized by many authors [Aila01, Bitt98, Bitt01, Coor97, Gree93]. Since practically all image-space operations in today's graphics systems are handled by the hardware, the focus has moved more on the spatial and temporal coherence rather than the image-space coherence.

Hierarchical visibility algorithms make use of the spatial coherence by utilizing a spatial hierarchy [Bitt01]. Spatial hierarchies group objects that are near to each other, and thus enable quick determination of invisible objects.

When the camera moves smoothly, the temporal coherence usually exists. Objects that were hidden in the previous frame are likely to be hidden in the next frame. If the camera movement is very rapid or the orientation and/or location of the camera changes significantly between frames, the temporal coherence is usually broken. One way of thinking about temporal coherence strategy is by guessing the final solution [Gree93]. If the guess is good, the underlying visibility algorithm has to resolve only the visibility of those objects that were not guessed.

Greene uses very basic technique to exploit temporal coherence in hierarchical Z-Buffer algorithm [Gree93]. The scene is first organized into the octree. At runtime the algorithm maintains the temporal coherence list; a list of visible octree nodes from the previous frame. In the next frame, objects in the temporal coherence list are first rendered without using the visibility algorithm. After the rendering of the temporal coherence list, rest of the hierarchy is processed normally using the visibility algorithm. After rendering of the new frame, the temporal coherence list is updated by checking the visibility of each node in the list.

Coorg and Teller [Coor97] use the concept of supporting and separating planes to cull hidden geometry. The algorithm exploits temporal coherence by caching the list of supporting and separating planes at each visited node of the spatial hierarchy. When the viewpoint changes, the algorithm only needs to check existing supporting and separating planes and update those nodes whose visibility states have changed. The algorithm works only in static environments.

Bittner and Havran present a series of improvements for the traditional hierarchical visibility algorithm [Bitt01]. Improvements increase the amount of exploited spatial and temporal coherence. Hierarchical visibility algorithms traditionally traverse the spatial hierarchy starting at the root node and descending towards leaf nodes. Traversal is typically performed in front-to-back order. View frustum culling is first applied on the current node. If the node is found to be outside of the view frustum, all its child nodes are also hidden. For all nodes that are inside the view frustum the occlusion culling test is applied. The occlusion culling stage can also classify whole subtrees as visible or hidden. If a node is not classified as visible or hidden but partially visible, the corresponding subtree is traversed further to refine its visibility state. To improve this kind of hierarchical visibility determination, three additional stages are proposed by Bittner and Havran: *hierarchical updating*, *visibility propagation* and *conservative hierarchical updating*.

Hierarchical Updating

During the traversal of a spatial hierarchy, all nodes are classified as visible, hidden or partially visible. Bittner and Havran use a notation of *termination node* for all fully visible and hidden nodes, and a notation of *open node* for partially visible nodes. Open nodes usually reside higher in the spatial hierarchy and termination nodes are closer to leaf nodes. If the camera movement is smooth the set of open nodes between successive frames can be expected to remain about the same. Hierarchical updating aims to eliminate the repeated visibility tests for the set of open nodes from the previous frame. Visibility tests are applied only on termination nodes.

During the traversal of the spatial hierarchy the visibility state is marked in each termination node. Visibility states are pulled up in the hierarchy if the visibility state of both child nodes is either visible or hidden; the node is marked respectively as visible or hidden. Otherwise the node remains to be open. Figure 15 illustrates the hierarchical updating.

For spatial hierarchies that are binary trees, hierarchical updating saves almost half of the visibility test that would have been applied on the inner nodes of the spatial hierarchy.

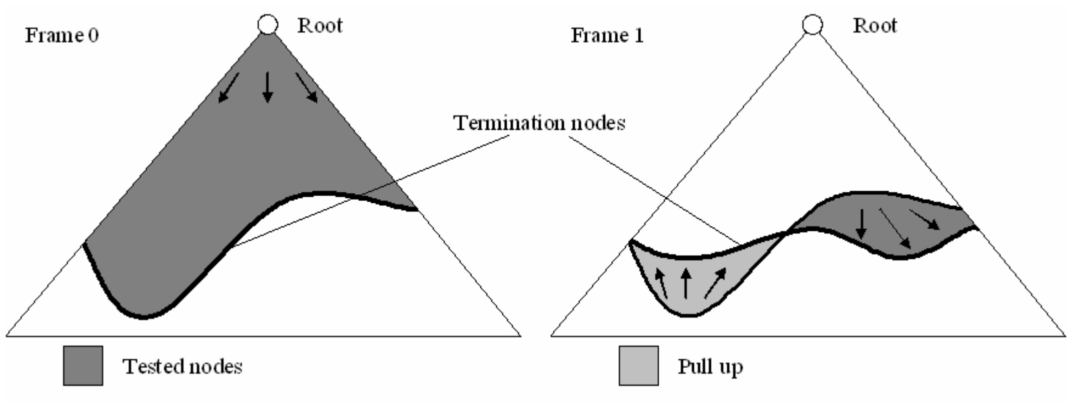

Figure 15. Hierarchical updating: partially visible nodes are marked as opened in the current frame. Visibility tests of the opened nodes are skipped in the next frame.

Visibility Propagation

The visibility of a node can be determined by determining the visibility of its faces. If the node corresponds to a rectilinear box, at most three faces have to be examined to determine the visibility of the node.

The visibility propagation attempts to further exploit the spatial coherence by trying to determine the visibility of a node by combining visibility classifications of its neighboring cells (figure 16). If the combination fails, the normal visibility test is applied to the node. If the tested node is fully occluded by nodes that are earlier determined hidden, the node can be also determined as hidden.

If the visibility propagation fails, it introduces additional overhead into the visibility determination. To avoid the overhead, visibility propagation is only applied to the nodes that were in the previous frame classified as fully visible or hidden. Also if the visibility propagation succeeded in the previous frame for some node it is applied to that node in the next frame.

Bittner and Havran use ropes [Havr98] for kD-trees to link neighboring cells. Visibility propagation does not work very efficiently in dynamic scene, because ropes or other links between adjacent cells have to be updated every time the cell structure changes.

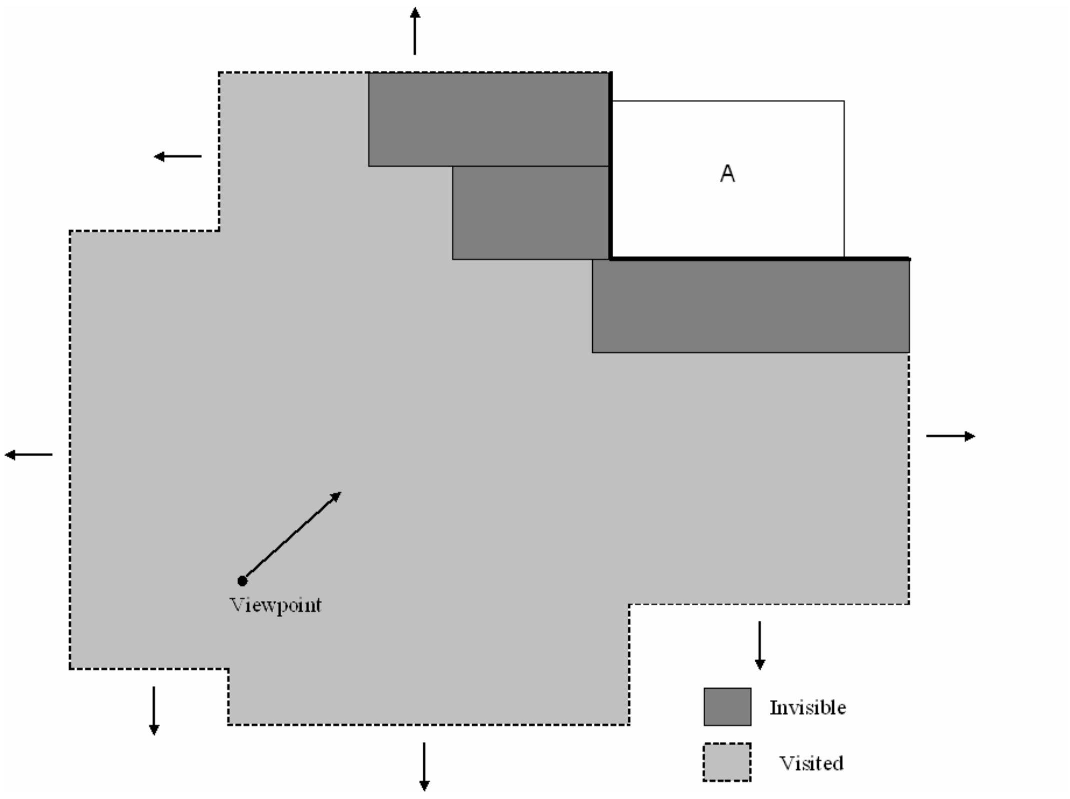

Figure 16. Visibility propagation: visibility of the node A can be determined as hidden using the visibility status of the appropriate neighbour nodes.

Conservative Hierarchy Updating

In addition to previous algorithms, conservative hierarchy updating attempts to overestimate the set of visible nodes with a certain specified probability. which avoids the further determination of the visibility of the nodes that probably are still visible. Conservative hierarchy updating does not report visible nodes as hidden but may needlessly report invisible nodes as visible.

3.8. Dynamic Scenes

Spatial data structures for static scenes have to be build only once at the preprocessing stage. The same data structure is then traversed for every frame. If the scene contains moving objects, the changes have to be updated to the spatial data structure. Updating the whole data structure obviously wastes unnecessarily time by reconstructing parts of the scene that have not changed.

Constructing the spatial data structure for each frame is too time-consuming to be done in real-time [Abra97, Aila01].

A moving object could be deleted and re-inserted to reflect its new location. But deleting and re-inserting an object would needlessly waste time by merging nodes which are immediately split again [Suda98].

Sudarsky [Suda98] proposes a more efficient method for updating a spatial data structure. The improvement is not to update the entire spatial data structure for a moving object but only the subtree whose root is the *least common ancestor* (LCA) of the object's old and new position. Sudarsky also proves two following claims:

- The spatial data structure update under the LCA is correct, i.e. it delivers the same results as updating the entire spatial data structure.
- The spatial data structure update under the LCA is optimal, in the sense that it does not needlessly delete or create any nodes in the spatial data structure (with the possible exception of old empty leaf nodes).

Sudarsky uses the octree data structure but the results are also applicable in other similar hierarchical spatial data structures, like kD-tree.

Aila and Miettinen [Aila01] present more efficient solution to update a spatial data structure. Their method does not delete or re-insert object at all. Instead, updating is performed locally by moving the object upwards in the tree until it fits completely inside the node. Later, when the spatial data structure is traversed, the object is potentially pushed down in the data structure if its location is more accurate in the child nodes. The object is pushed downwards until either a leaf node is reached or the volume of the node is smaller than the volume of the object. Leaf nodes containing no objects are destroyed. All spatial structure updates are performed during the visibility traversals. The lazy updating can potentially save some unnecessary updates compared to the LCA method by Sudarsky [Suda98].

Shagam and Pfeiffer's [Shag03] method to manage dynamic scenes does not require that objects are always in the leaf nodes of the spatial data structure. Objects are stored at the deepest level of the octrees at which they fit entirely in a single node. Shagam and Pfeiffer use octree data structure but do not require that the number of splitting planes is always three, or that the splitting point is always at the center of the node.

Similar to the method by Aila and Miettinen [Aila01], Shagam and Pfeiffer's method [Shag03] updates the spatial data structure during the traversals. When a node is split, the splitting point is calculated by taking the mean of every object's center, weighted by the inverse of its radius. Objects in the node are tested against three splitting planes created through the splitting point. If the number of objects intersecting with a splitting plane exceeds a threshold, the plane is removed. If all planes are removed the plane with the least intersections is left. Dynamic objects are updated by deleting and re-inserting the objects.

If the updating of a spatial data structure is performed for every moving object, the overall algorithm will not be output-sensitive, it will waste time on updating hidden objects as well as visible ones [Suda98].

To maintain output-sensitivity, dynamic updates should be performed only on visible parts of the spatial data structure. The method by Aila and Miettinen [Aila01] does not update hidden parts of the scene because updates are performed during the visibility query traversal. Sudarsky's LCA method also updates the hidden parts of the scene. If the hidden object is just ignored after it is determined as hidden, it might not be displayed again when it should be [Suda98]. This happens when a part of the scene where the object should be becomes visible but the object is not updated because it was previously determined as hidden.

Temporal Bounding Volumes

Sudarsky [Suda98] presented a method to avoid updating hidden parts of the scene. The method uses *temporal bounding volumes* (TBV). A TBV is a volume guaranteed to contain a dynamic object from the moment of the creation of the TBV until some later time. This time is called the *expiry date* of the TBV, and the length of the time from the creation of TBV to the expiry date is the *validity period* [Suda98]. Figure 17 illustrates an object that is bounded by a TBV during the time period from 0 to n.

In order to calculate the TBV for an object, something about the object's behavior has to be known. If the velocity and the trajectory of the object are known, well fitting TBV can be calculated. If only the maximum velocity is known, a sphere can be used as a TBV.

TBVs are inserted into the spatial data structure. If a TBV stays hidden during its validity period, the object's location does not have to be updated until the expiry date.

The exact validity period can be very complicated to estimate. In many cases, an exact validity period cannot be calculated at all. If the validity of the TBV is chosen to expire too soon, time is unnecessarily used to update the TBV again. On the other hand, if the validity period is chosen to end too far in the future, the size of the TBV becomes very large, and can intersect with the visible parts of the scene, even if the object itself is not near.

Sudarsky suggested adaptive validity periods in general situations. If a TBV expires before it is revealed, then its validity period was too short because it would have been possible to postpone the costly reference to the dynamic object by choosing a later expiry date. Therefore, a longer period is selected for the object's next TBV. In the opposite case - if the TBV is seen before its expiry – the period was too long, because a shorter period would have produced a smaller TBV that might have remained occluded for a longer time. Therefore, if the object itself is still hidden a shorter validity period will be chosen for its next TBV.

If the constraints of an object's movement are very loose, the resulting TBV might become very big, and not contribute much to output-sensitivity. For this kind of situations Sudarsky suggested using *fuzzy TBVs*. A fuzzy TBV is a volume that is not necessarily guaranteed to contain its dynamic object throughout its validity period but is only assumed to do so with some probability.

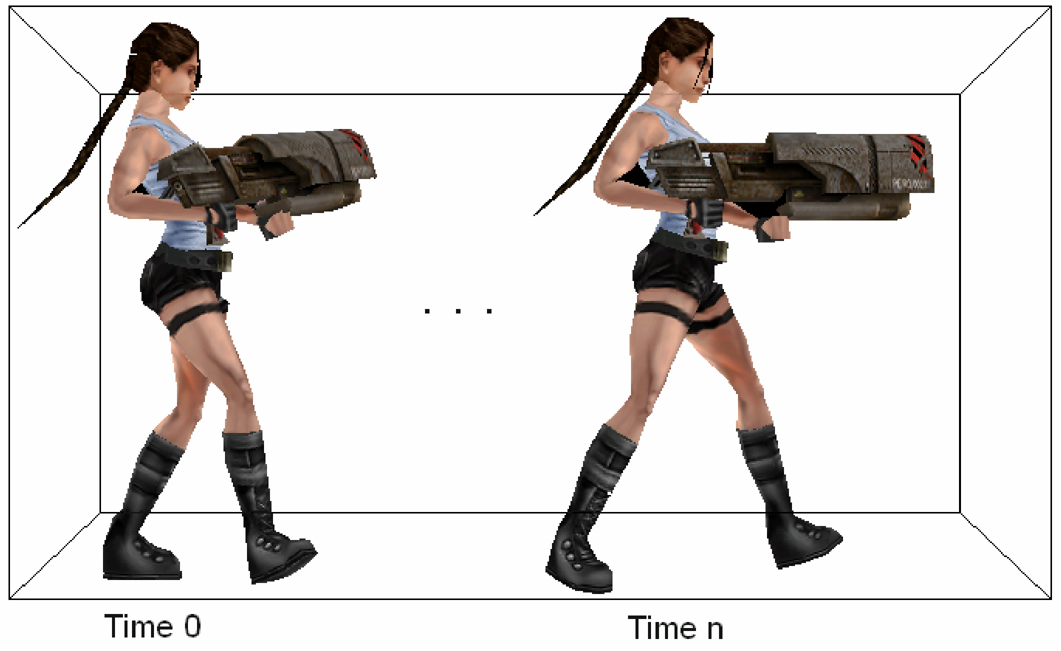

Figure 17. A temporal bounding volume (AABB) that contains the character during the time period from 0 to n.

TBVs are used in dPVS by Hybrid [Aila01]. They manage to significantly reduce the number of required database updates. TBVs also provide temporal coherence to bounding volume test of dynamic objects [Aila01]. Also Batagelo and Wu [Bata01] use TBVs for dynamic objects.

3.9. Summary

Efficient rendering of large scenes requires a spatial data structure to accelerate the determination of visible objects. Spatial data structures enable the logarithmic search for visible objects.

The BSP-tree data structure is almost the standard in modern games. It enables efficient PVS based visibility system [Abra97]. A downside is that the PVS based visibility system requires time consuming preprocessing stage, and it is usable only with static parts of the scene.

The differences between BSP-trees, kD-trees and octrees are relatively small. BSP-trees enable the most adaptive way to decide the decomposition. Calculating the optimal subdivision of a BSP-tree node can be very time consuming, or even impossible.

Octrees typically use regular decomposition. Determining the subdivision of an octree node is done in constant time. The only way to control the subdivision of an octree, is to decide when to stop the subdivision.

BSP-trees usually provide more fitting cells whereas octrees usually contain more empty, or unevenly filled cells. An axis-aligned version of the BSP-tree data structure, kD-tree, provides better fitting cells than the octree but is restricted to parallelepiped cells.

Visibility determination during the traversal of a spatial hierarchy can be accelerated by exploiting coherences. In Section 3.7 several techniques to exploit spatial and temporal coherence were reviewed.

In dynamic scenes the spatial hierarchy needs to be updated when an object changes its locations. Techniques to efficiently update spatial hierarchies were reviewed in Section 3.8.

4. View Frustum Culling

Practically all rendering systems for virtual environments, games or other applications that require high performance use view frustum culling (VFC). View frustum culling is an efficient way to cull away hidden parts of the scene. VFC takes place before the transformation and shading, and thus decreases the workload in a very early stage of the graphics pipeline.

VFC is usually done on the object level using bounding volumes [Eber00]. It divides objects of the scene into three categories; visible, partially visible and hidden. Visible objects are the ones that are completely inside the view frustum, partially visible objects intersect with view frustum planes, and hidden objects are the ones that are completely outside of the view frustum (figure 18).

Modern graphics systems typically treat partially visible objects in the same way as fully visible; they are rendered without clipping the polygons. This way the clipping of partially visible objects is handled by the graphics hardware. The clipping is practically free when done with modern graphics hardware [Diet01], regardless of the overhead caused by hidden polygons. The clipping is normally done in software only when the appropriate hardware implementation is not available. If the modern hardware is available, clipping in software is probably more time consuming than sending the whole object to graphics hardware which handles the clipping.

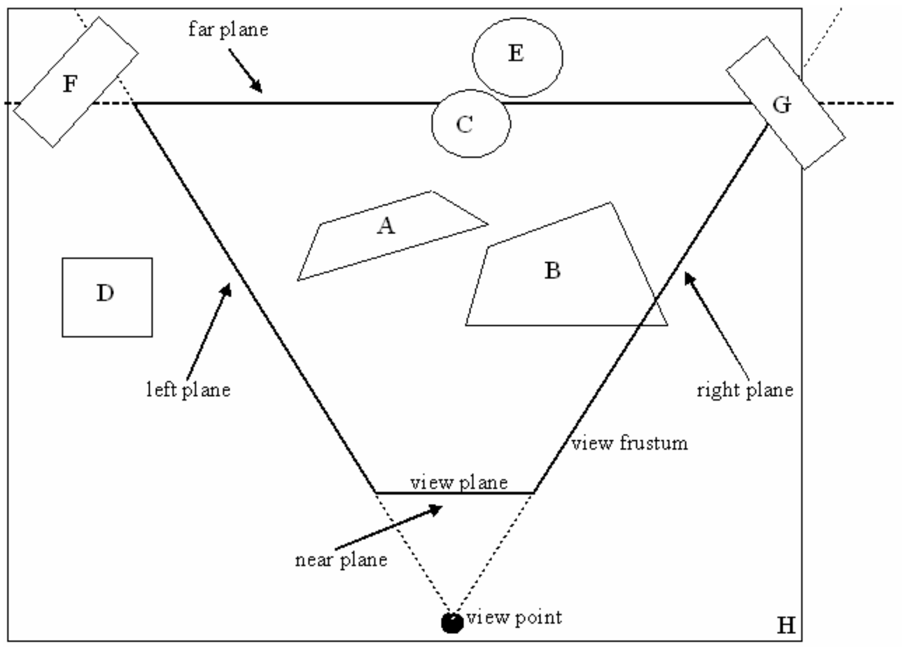

Figure 18. Top-view of the view frustum. Visible object: A, B, C, G and H. Hidden objects: D, E and F.

4.1. View Frustum Construction

If the used projection transformation is a perspective projection, the view frustum is a truncated pyramid. The viewpoint is located at the apex of the pyramid, and the viewing direction is towards the bottom of the pyramid.

If the parallel projection is used, the view frustum is a parallelepiped. Games and virtual environment applications typically use perspective projection, while CAD programs often use parallel projection. This thesis concentrates on the perspective projection since the emphasis is on games and virtual environments. Introductions to parallel projections can be found for example in the books by Foley [Fole90] and Watt [Watt99].

In games and virtual environments the view frustum is typically symmetric, meaning that left and right planes of the view frustum are in same degree angles.

The view frustum consists of six planes that enclose the visible volume of the space. The *near plane* is perpendicular to the view vector. The view port lies on the near plane. Parallel to the near plane is the *far plane* that limits the viewing area in front of the viewer. The distance to the far plane depends on the application settings. The far plane defines how far the viewer can see. If the far plane is too close, objects that come closer than the far plane, may suddenly seem to "pop-up" when they become visible. This effect easily breaks the immersion. On the other hand, if the far plane is moved too far, the rendering consumes time without any significant improvements in the image. If the viewer is allowed to see infinitely far, the far plane can be removed. This way, the view frustum test is simpler but it is likely that more objects will be reported as visible. The other four planes that limit the view frustum are *left plane*, *right plane*, *top plane* and *bottom plane*.

Planes are normally positioned so that normals of the planes point away from the view frustum. It is also possible to define planes so that normals point to the inside of the view frustum. In this thesis, normals of the frustum planes are assumed to point away from the view frustum.

The view frustum needs to be generated every time the orientation or the position of the viewpoint changes. Gribb and Hartmann [Grib01] describe how to extract frustum planes from the world-view-Matrix. They derive the plane equation for left- and right-handed coordinate system. They also provide source code for DirectX [Micr03] and OpenGL [Open03]. The derivation of plane equations are omitted here, only the results are presented.

A plane in three-dimensional space is defined by the equation

$$Ax + By + Cz + d = 0\,.$$

Now let $M = \begin{matrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{43} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{matrix}$ be a projection matrix.

In the left-handed coordinate system the coefficients of the view frustum planes are calculated as follows:

Left plane: $(m_{14} + m_{11})x + (m_{24} + m_{21})y + (m_{34} + m_{31})z + (m_{44} + m_{41}) = 0$
Right plane: $(m_{14} - m_{11})x + (m_{24} - m_{21})y + (m_{34} - m_{31})z + (m_{44} - m_{41}) = 0$
Top plane: $(m_{14} - m_{12})x + (m_{24} - m_{22})y + (m_{34} - m_{32})z + (m_{44} - m_{42}) = 0$
Bottom plane: $(m_{14} + m_{12})x + (m_{24} + m_{22})y + (m_{34} + m_{32})z + (m_{44} + m_{42}) = 0$
Near plane: $(m_{13})x + (m_{23})y + (m_{33})z + (m_{43}) = 0$
Far plane: $(m_{14} - m_{13})x + (m_{24} - m_{23})y + (m_{34} - m_{33})z + (m_{44} - m_{43}) = 0$.

In the right-handed coordinate system the coefficients of the view frustum planes are calculated as follows:

Left plane: $(m_{41} + m_{11})x + (m_{42} + m_{12})y + (m_{43} + m_{13})z + (m_{44} + m_{14}) = 0$
Right plane: $(m_{41} - m_{11})x + (m_{42} - m_{12})y + (m_{43} - m_{13})z + (m_{44} - m_{14}) = 0$
Top plane: $(m_{41} - m_{21})x + (m_{42} - m_{22})y + (m_{43} - m_{23})z + (m_{44} - m_{24}) = 0$
Bottom plane: $(m_{41} + m_{21})x + (m_{42} + m_{22})y + (m_{43} + m_{23})z + (m_{44} + m_{24}) = 0$
Near plane: $(m_{41} + m_{31})x + (m_{42} + m_{32})y + (m_{43} + m_{33})z + (m_{44} + m_{34}) = 0$
Far plane: $(m_{41} - m_{31})x + (m_{42} - m_{32})y + (m_{43} - m_{33})z + (m_{44} - m_{34}) = 0$.

If the matrix M is equal to the projection matrix (i.e. $M = P$), the view frustum planes are given in view-space. If the matrix M is equal to the combined view and projection matrix (i.e. $M = V * P$), the view frustum planes are given in world-space. If the matrix M is equal to the combined world (i.e. $M = W * V * P$), view and projection matrix, the view frustum planes are given in object-space.

4.2. View Frustum Test

There are two common approaches to perform the view frustum test [Assa99]. One is to perform calculations in perspective transformed view-space. The intersection test between an AABB and the view frustum in view-space is very efficient since the test can be done by testing the intersection of two AABBs [Assa99]. This intersection can be tested with six comparison operations

[Assa99]. The drawback of the view-space method is that all BVs and the view frustum have to be transformed to the perspective coordinate system. This requires all eight vertices of an AABB to be multiplied with the view-projection matrix, which includes at least 72 multiplications per object [Assa99].

Alternatively the view frustum test can be done in world-space coordinates. The visibility test is performed by testing the intersection of a bounding volume with the six planes of the view frustum. Tests can be arranged so that if the attempts for trivial rejection or acceptance fail, only then more expensive intersection tests are applied.

If a bounding volume is fully outside one of the view frustum planes, it is then fully outside the whole frustum, and can be trivially rejected.

If a point that lies on the bounding volume is inside of all the view frustum planes, then the bounding volume intersects with the view frustum, and can be trivially accepted. If all points that lie on the bounding volume are inside the view frustum, then the bounding volume is fully inside the view frustum. Determining the partial or the full containment may be essential when the scene is processed hierarchically, see Section 4.3.

If a bounding volume is not trivially accepted or rejected, three possible cases remain:

1. If the bounding volume is a box (AABB or OBB), the corner of the view frustum may protrude through a face of the box (object G in figure 18).
2. The bounding volume contains the view frustum (object H in figure 18).
3. The bounding volume is outside the view frustum but crosses the view frustum planes outside of the view frustum (object F in figure 18).

Trivial cases involve one or more intersection tests between the view frustum planes and the bounding volume. The bounding volume is typically an AABB, OBB or a sphere. The test can be significantly accelerated if the intersection algorithm is optimized for the particular bounding volume.

Bounding boxes, AABBs and OBBs are very often used with view frustum culling. An obvious algorithm would be to test if all vertices of a box are inside or outside the view frustum plane. Each vertex is evaluated with the plane polynomial:

$$Ax + By + Cz + d\ .$$

If the value is more than 0, the vertex is outside the plane, otherwise it is on or inside. This method unnecessarily tests all eight vertices when only two needs to be tested [Hoff96b, Grap94]. Vertices that need to be tested are the farthest vertex in the direction of the plane normal (p-vertex) [Assa99] and the farthest vertex in the negative direction of the plane normal (n-vertex) [Assa99].

To identify the p- and n-vertex, the bounding box and the plane normal *N* need to be in the same coordinate system. AABBs are defined in the world coordinate system. Because OBBs are defined in their own coordinate system, the plane normal has to be transformed to the coordinate system of the OBB. Assuming that the axis vectors of an OBB are normalized, the transformation can be done by projecting the normal *N* onto axes of the OBB [Hoff96b]. The projected plane normal *N′* is calculated with three dot products:

$$\overrightarrow{N}' = ((\overrightarrow{N} \cdot \overrightarrow{X}_{axis}), (\overrightarrow{N} \cdot \overrightarrow{Y}_{axis}), (\overrightarrow{N} \cdot \overrightarrow{Z}_{axis})) .$$

To determine the p-vertex, the following cascading set of conditions is used By Hoff [Hoff96b].

```
If($x_N > 0$)                //Right
  If($y_N > 0$)              //Right, Top
        If($z_N > 0$)        //Right, Top, Front
        Else                 //Right, Top, Back
  Else                       //Right, Bottom
        If($z_N > 0$)        //Right, Bottom, Front
        Else                 //Right, Bottom, Back
Else                         //Left
  If($y_N > 0$)              //Left, Top
        If($z_N > 0$)        //Left, Top, Front
        Else                 //Left, Top, Back
  Else                       //Left, Bottom
        If($z_N > 0$)        //Left, Bottom, Front
        Else                 //Left, Bottom, Back,
```

where *N* is the plane normal in the right-handed coordinate system of the bounding box. To determine the n-vertex, the same cascading set of conditions is used but *N* is set to be –*N*. Only the evaluation of signs of the components of *N* is needed to determine n- and p-vertex. Assarsson and Möller suggest using a bitfield to store this information and to avoid conditional branches [Assa99].

The determination of the n- and p-vertex for an OBB takes 18 multiplications, 12 additions, 6 comparisons and 3 negations. For an AABB the determination of the n- and p-vertex takes only 6 comparisons and 3 negations. By using the symmetry of the bounding box, determination of the n- and p-vertex for an OBB can be reduced to 9 multiplications, 6 additions and 3 comparisons, and for AABBs 3 comparisons [Hoff96b].

To determine the intersection of a box with the plane, n- and p-vertex are first tested. If the n-vertex lies on the positive side of the plane, then the whole box is outside the plane. If the n-vertex lies on the negative side of the plane, then the p-vertex is tested. If the p-vertex also lies on the negative side of the plane, then the whole box is inside the plane. Otherwise the box intersects with the plane.

As the worst-case result the whole box-plane overlap test requires the following operations: 24 multiplications, 18 additions, 8 comparisons and 3 negations with OBB, and 6 multiplications, 6 additions, 8 comparisons and 3 negations with AABB.

Hoff's method offers significant speedup for the AABB test. Eberly [Eber00] describes even more efficient method for OBBs. The OBB and the view frustum plane are projected onto the line $\vec{C} + s\vec{N}$ where $\vec{C}$ is the center point of the OBB, and $\vec{N}$ is the normal of the plane. The symmetry provided by the box definition yields on interval of projection $[\vec{C} - r\vec{N}, \vec{C} + r\vec{N}]$. The interval is centered at $\vec{C}$ and has radius $r = a_0\left|\vec{N} \cdot \vec{A_0}\right| + a_1\left|\vec{N} \cdot \vec{A_1}\right| + a_2\left|\vec{N} \cdot \vec{A_2}\right|$, where $\vec{A_i}$ form the orthogonal basis of the OBB coordinate system, a_i is the spatial extent and $i = 0,1,2$. The view frustum plane projects to a single point $\vec{P} = \vec{C} + (d - \vec{N} \cdot \vec{C})\vec{N}$.

The OBB is outside the plane as long as the projected interval is outside in which case $\vec{N} \cdot \vec{C} + d < r$. The test is identical to sphere test except that r has to be calculated for each test of an OBB. The test requires 15 multiplications and 11 additions.

Spheres are also commonly used bounding volumes. A sphere is defined by its center point $\vec{C}$ and the radius r. If the distance of the center point $\vec{C}$ from a view frustum plane is more than the radius r, then the sphere is outside of the view frustum. This can be calculated with dot product of the normalized plane normal and the center point $\vec{C}$. The result is compared to the radius:

$$\vec{N} \cdot \vec{C} + d > r .$$

If the visibility of the bounding volume after trivial rejection and acceptance tests is still unknown, more expensive special cases have to be tested [Hoff96a].

To test if the view frustum protrudes through a box face, 12 edges of the view frustum have to be tested for intersection with six faces of the box. The test can be terminated when the first intersection is found. In the worst case, the test takes $12 * 6 = 72$ intersection tests.

If the visibility of the bounding volume is not yet determined, next test checks for containment of the view frustum inside the bounding volume. The test is done by testing if the origin of the view frustum is inside the bounding volume. For bounding boxes all faces have to be tested in the worst case.

If both previous tests fail, then the bounding volume is hidden.

Assarsson and Möller [Assa99] present a general view frustum culling algorithm for arbitrary shaped bounding volumes. Their method creates additional outer and inner view frusta. Using these new frusta, it is sufficient to test only one or few points of the bounding volume to determine the intersection of the BV and the view frustum. First a point p_{BV} inside the bounding volume is chosen, for example the center point. The larger view frustum is created by sweeping the bounding volume along the outer side of the planes of the original view frustum keeping the orientation of the bounding volume. All points passed by the point p_{BV} define the outer new view frustum. The new inner view frustum is created similarly by sweeping along the inner side of the original view frustum.

After the creation of new view frusta, the view frustum test can be done by testing the point p_{BV} against new view frusta. If the point p_{BV} lies inside the inner view frustum, the bounding volume is fully inside the original view frustum. If the point p_{BV} is outside the outer view frustum, the bounding volume is fully outside the original view frustum. If the point p_{BV} lies between the inner and outer view frustum, the bounding volume intersects with the original view frustum.

Other geometric objects can be used to bound arbitrary objects. As long as they are convex and simpler than the original object, they can be used with view frustum culling. Eberly [Eber00] describes view frustum culling algorithms for capsules, lozenges, cylinders and ellipsoids.

4.3. Hierarchical View Frustum Culling

A bounding volume accelerates per object tests if the model is significantly more complex than the bounding volume. View frustum culling can be accelerated further if tests are done hierarchically. Hierarchical view frustum culling uses some hierarchical data structure, typically one of spatial data structures described in Chapter 3.

Hierarchical view frustum culling starts at the root node and traverses toward the leaf nodes. In every visited node the corresponding bounding volume is tested. If the bounding volume is determined as outside, the corresponding subtree is also outside, and does not require further processing.

If the bounding volume is determined as fully inside, the corresponding subtree is also fully inside, and its leaf nodes can be rendered without further testing. If the bounding volume of the current node intersects with the view frustum, the visibility of the subtree is determined by descending to child nodes and testing them. When the traversal reaches leaf nodes, objects contained by leaves are first tested, and if found visible then rendered.

It is very important that all bounding volumes of child nodes are fully inside the bounding volume of the parent node. Otherwise it is not possible to cull entire subtrees.

Nodes of an axis-aligned spatial data structures, like octrees and kD-trees, correspond to AABBs, and can be culled using the algorithm in Section 4.2. Nodes of the BSP-trees are arbitrary convex volumes, and are more complex to cull. Testing an arbitrary convex volume depends on the number of vertices of the convex hull.

Each BSP-tree node contains the splitting plane. The visibility of the child nodes can be determined by testing the intersection of the splitting plane with the view frustum. If the view frustum is fully on one side of the splitting plane, then the other side of the splitting plane is fully hidden.

Alternatively for each BSP-tree node, a BV can be assigned, and the view frustum test can be done using the BV.

4.4. Optimizations

Assarsson and Möller present four optimization techniques for view frustum culling [Assa99]: the *plane-coherency test*, the *octant test*, *masking* and *TR coherency test*.

The Plane-Coherency Test

The plane-coherency test attempts to exploit temporal coherence. Let's assume that in the previous frame a BV was outside one of the frustum planes. If the camera movement is smooth it is likely that the temporal coherence exists. In the current frame, it is a high probability that the BV is outside that same plane, and the frustum test should start by testing that same plane. If the temporal coherence exists, the frustum test is accelerated, because the rejection is determined by testing only one plane.

If a BV is determined to be outside one of the frustum planes, the index of the plane is cached to the BV structure. In the next frame the testing starts with the plane pointed by the index saved during the previous frame.

The Octant Test

The octant test can be implemented for symmetric view frusta, and it is primarily used with bounding spheres. For the octant test, the view frustum is split in half along each axis subdividing the frustum into eight parts. When a bounding sphere is tested, the octant where the center of the sphere resides is identified. To test the sphere, it is sufficient to test outer three planes of the octant. If the bounding sphere is inside the three nearest plane, it must be also inside all planes of the view frustum. If the sphere is outside of any tested plane, it is totally outside of the view frustum.

The octant test can be used with arbitrary bounding volumes with one additional condition. If the radius r_S of a minimal sphere, surrounding the BV and with the center point c_S, is less than the distance d from the center of the view frustum c_{VF} to the closest plane of the c_{VF}, then it is sufficient to test the intersection against three outer planes of the view frustum octant O_{VF} that the sphere c_S lies in.

Masking

Masking can be used with hierarchical view frustum culling. If the BV of a node is completely inside one of the planes of the view frustum, then all child nodes of the node are also inside that same plane. The plane can be eliminated (masked off) from further testing in the subtree of the node.

The mask can be implemented as a bitfield, where a bit for each frustum plane indicated whether the parent is inside that plane. The mask is passed from parent node to child nodes. Before the plane test, the mask is checked if that plane is masked off or not.

The TR Coherency Test

TR coherency (translation and rotation coherency) attempts to exploit temporal coherence in environments where the navigation is limited to rotation around one axis, or translation. For objects that have not moved since the last frame the following applies.

- If only view frustum rotations have been done around one of the axis only since the last frame, the tested BV can be classified as rejected if the distance to the plane that determined rejection in the previous frame has increased.

- If the view frustum has done only a pure translation since the previous frame, the distances from all BVs to the same view frustum plane have either increased or decreased by the same amount Δd. If only a translation has been done since the last view frustum culling invocation, the distance Δd_i is precomputed for each view frustum plane i by projecting the translation on the normal of the planes. For each BV and view frustum plane to be tested, the distance Δd_i is compared to corresponding distance in the previous frame.

Assarsson and Möller also provide test results using spheres, AABBs or OBBs, and different combinations of algorithms. Three different scenes were used with predefined camera path, random user navigation and pure rotation and pure translation navigations.

4.5. Summary

View frustum culling is a very efficient and widely used conservative method to cull most of the hidden geometry of the scene. View frustum culling works well in environments where the viewer is surrounded by objects of the scene and the large part of the scene is not visible. Games and virtual environments typically have these kinds of scenes. CAD applications usually do not benefit as much from view frustum culling as games, because most of the scene is inside the view frustum.

This chapter has reviewed algorithms to efficiently calculate the intersection between typical bounding volumes and the view frustum. The AABB intersection test is especially important because it can be used with octrees and kD-trees whose cells correspond to AABBs. For single objects the OBB intersection test may be more efficient than the AABB test because OBBs can potentially provide tighter bounding of an object.

5. Occlusion Culling

Modern academic studies of the hidden surface removal have strongly concentrated on occlusion culling. While view frustum culling and back face culling are relatively straightforward operations, occlusion culling is a more complex and not so precisely defined task. Although many approaches to solve occlusion culling are presented, common operations exist to almost all occlusion culling algorithms. This chapter describes common characteristics of occlusion culling algorithms and represents well known algorithms.

5.1. Fundamental Concepts

The input set of an occlusion culling algorithm is the output set of view frustum culling algorithm, objects that are partially or totally inside the view frustum. The task of an occlusion culling algorithm is to separate actually visible objects from objects that are hidden by other objects.

The term *occluder* refers to an object that obstructs (or occludes) other object(s). In figure 19, objects A and B are occluders. *Occlusion power* is a characteristic that is used to describe the ability of an occluder to occlude other objects. An *occluder set* is a set of objects that the occlusion algorithm chooses as occluders. The occluder set is likely to change if the viewer's position or orientation changes. In figure 19, objects A and B form an occluder set. The term *occludee* refers to an object that is occluded by an occluder or a set of occluders. Bounding volumes are typically used as occludees during the occlusion tests.

A group of occluders can be merged into one big occluder. This operation is called the *occluder fusion*. The occlusion power of the resulting fused occluder is always greater or equal to the sum of occlusion powers of the merged occluders. The fusion of occluders can significantly improve the performance of the occlusion culling algorithm. In figure 19, if occluders A and B are fused they can occlude the object C. Alone neither of them can fully occlude the object C.

An object that is actually not visible in the scene can also act as an occluder. The term *virtual occluder* is used for this kind of occluders. If the visibility in a scene is known to be *n* meters, a virtual occluder can be constructed in the distance of *n* meters, and be used to occlude the hidden parts of the scene. In figure 19, occluders A and B can be replaced with the virtual occluder F.

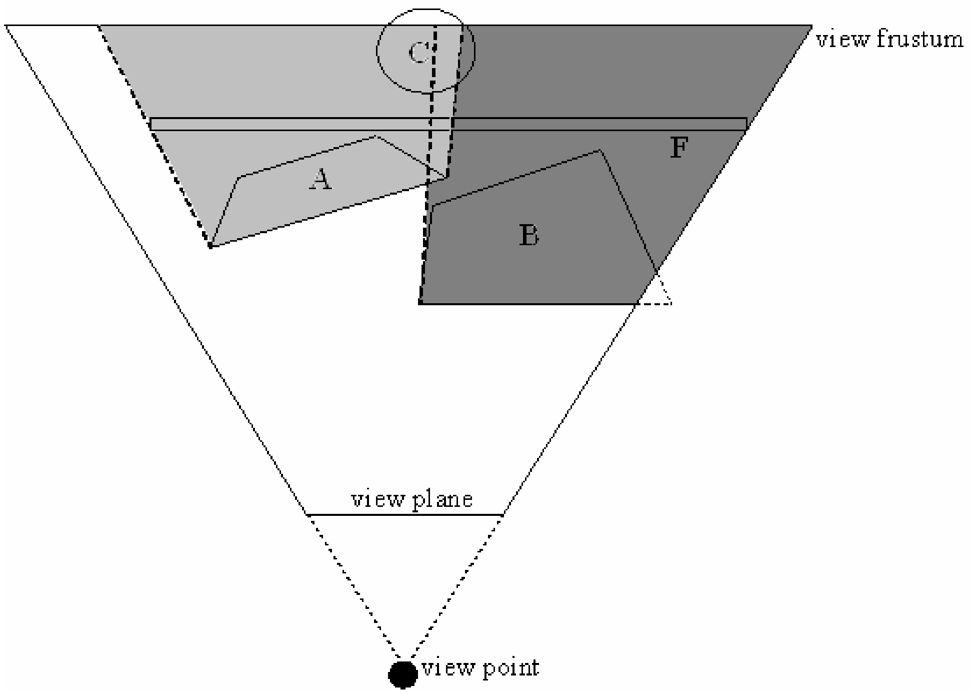

Figure 19. Object C (occludee) is occluded by occluders A and B. Object F is a virtual occluder which can be used instead of occluders A and B.

5.2. Occluder Selection

Occluder selection is an operation that is typical to most of the occlusion algorithms. An occlusion culling algorithm selects a subset of potential occluders as the occluder set. Potential occluders can be pre-processed for a static scene, or they can be selected dynamically each time they are needed. The accurate occluder selection is very important task. If a set of bad occluders is selected, performance of the application can decrease compared to situation where occlusion culling is not used.

Even though occluder selection is often very hard to describe generally and yet accurately, there exist some classification criteria that can be used when evaluating an occluder [Aila01]:

- Size: Small objects usually do not serve as good occluders unless the viewer is very close to them. The size of a potential occluder is dependent on the distance to the viewer. An object that is small compared to other objects might still be a good occluder if it is very near the viewer. This leads to the conclusion that good occluders often reside near the viewer, especially when perspective projection is used.

- Redundancy: Some objects, for instance a clock on the wall, provide redundant occlusion and should be removed from the database.
- Rendering complexity: Objects with high polygon count or rendering complexity are not preferred since scan-converting them may take considerable time and affect the overall frame rate.

Many occlusion culling algorithms [Bitt98, Coor97, Huds97] use the *solid angle* metric to estimate a potential occluder. The solid angle Ω subtended by a surface S is defined as the surface area Ω of a unit sphere covered by the surface's projection onto the sphere. This can be written as

$$\Omega = \iint_S \frac{\vec{n} \cdot da}{r^2},$$

where $\vec{n}$ is a unit vector from the origin, da is the differential area of a surface patch, and r is the distance from the origin to the patch.

The solid angle measures the size of an object, as seen from a point in space. Its value is determined by the size of the object's projection on the unit sphere centered around the view point [Kolt00].

A popular approximation of the solid angle is the area-angle by Coorg and Teller [Coor97]. The area-angle is the quantity

$$\frac{-A(\vec{N} \cdot \vec{V})}{\|\vec{D}\|^2}$$

where A represents the area of the occluder, $\vec{N}$ represents the normal of the occluder, $\vec{V}$ represents the viewing direction, and $\vec{D}$ represents the vector from the viewpoint to the center of the occluder. The area-angle can be also written in the following form:

$$\frac{A * \cos\theta}{r^2},$$

where θ is the angle between the normal of the surface and the directional vector from the surface to the viewpoint, and A is the area of the surface [Kolt00].

The area-angle captures properties of the subtended solid angle of the polygon:

- Larger polygons have an larger area-angle.
- The area-angle falls as a square of the distance from the viewpoint, as does the subtended angle.
- The maximum area-angle occurs when the viewing direction $\vec{D}$ is "head-on" with the occluder, and falls with the dot product as the occluder is viewed obliquely.

The area-angle differs from the solid-angle by not considering the actual shape of the occluder. It is much simpler to compute and serves as a useful heuristic to identify large occluders near the viewpoint [Coor97]. Figure 20 illustrates the concept of area-angle.

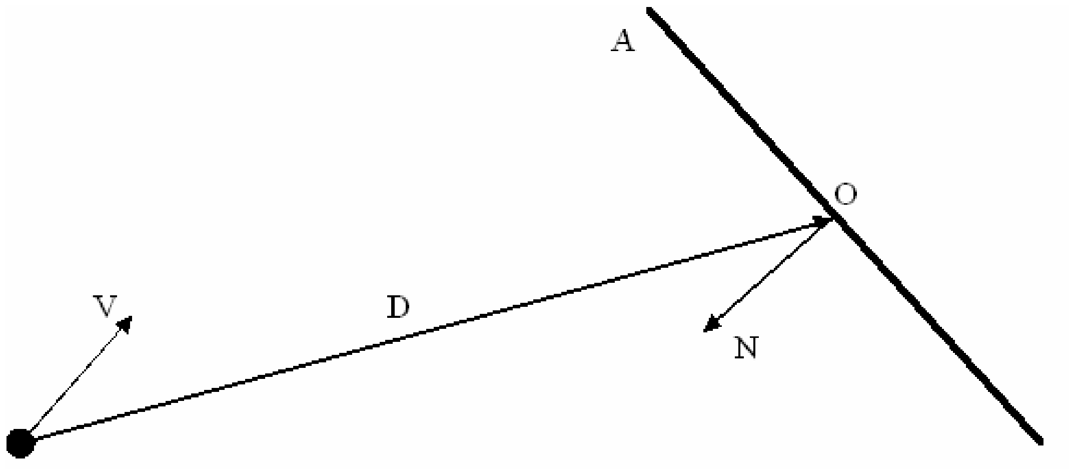

Figure 20. Parameters in the area-angle metric.

In some cases the solid angle does not give a correct evaluation of the occluder. Koltun [Kolt00] proposes an object-space method that correctly measures the potential occlusion contribution of an object in both from-region and from–point cases.

If the viewpoint is treated as a light source, the shadow volume (umbra) casted by an object corresponds to the volume the object is capable of occluding. Koltun's idea is to compare this shadow volume to viewable volume. Koltun describes an accurate method in 2D and an approximate method in 3D.

The approximate method places two planes behind the potential occluder. Planes are parallel to each other and the distance between them is a small constant δ. The potential occluder is projected to both planes, and the areas are computed. The proposed measure M is defined as

$$M = \frac{A_2}{A_1},$$

where A_1 is the area of the closer projection and A_2 is the area of the farther projection. This ratio yields to an estimation of the growth rate of the umbra, which reflects the umbra's size. This ratio is also proportional to the area-angle of the occluder. Koltun's method can be used to estimate the whole object whereas the area-angle gives an estimation of a single polygon.

Koltun's method can be hardware assisted using the stencil buffer [Micr03, Open03].

5.3. Hardware Occlusion Queries

For several years the Z-buffer [Catm74] algorithm has been the standard method to solve the visibility in hardware. Practically all graphics chips implement the Z-buffer algorithm. As it has been noted in this thesis and similar studies, Z-buffer is not alone fast enough to determine visibility in complex scenes. This problem is also issued by hardware manufacturers who have introduced their hardware occlusion culling systems in their recent generations of graphics chips [More00, NVID02].

ATI's Hyper Z Technology [More00] is a collection of algorithms implemented in hardware that try to reduce the memory bandwidth inside the graphics chip. Hyper Z technology contains Fast Z Clear, Z Compression and Hierarchical Z algorithms. Hierarchical Z algorithm is a hierarchical implementation of ordinary Z-buffer. Hierarchical Z is able to discard hidden samples before texturing [More00]. Fast Z Clear allows to clear the depth buffer without actually writing out depth values for all pixels in the depth buffer. Fast Z Clear is essentially free [Rigu02]. Z compression provides lossless compression of the depth buffer. Compression rate can be up to 24:1 [Rigu02].

NVIDIA has a similar hierarchical occlusion culling algorithm implemented as a part of their Lightspeed Memory Architecture [nvid02]. Their visibility subsystem is called Z-Occlusion Culling.

NVIDIA's and ATI's methods do not require that objects are rendered in front-to-back order but this way hierarchical z-testing is most efficient [Rigu02].

Along with the Hyper Z and similar methods, hardware manufacturers have provided support for occlusion queries through common graphics APIs. The rest of this section reviews occlusion queries supported by DirectX 9 [Micr03] and OpenGL [Open03].

The OpenGL extension HP_OCCLUSION_TEST [GLHP] provides a mechanism to determine if the rendered geometry could have or could not have modified the depth buffer. If a set of geometry could modify the depth buffer, some or all of the geometry will be visible if it is rendered. If the geometry could not have modified the depth buffer, it would be hidden if rendered, and therefore it can be culled. The return value of the visibility query is boolean, true if geometry is visible and false if hidden. Objects that are tested with HP_OCCLUSION_TEST are typically bounding volumes.

The extension operates independently of the current rendering state. When HP_OCCLUSION_TEST is enabled, samples are generated and the depth and/or color buffer may be updated. To prevent updating the depth/color buffers, the application must disable updates to these buffers. As a side effect of reading the occlusion result the internal result state is cleared, setting it up for a new occlusion test.

The test uses a “stop-and-wait” model for multiple tests which can easily stall the rendering pipeline and eliminate the parallelilism between main processor and graphics chip.

Staneker uses HP_OCCLUSION_TEST in OpenSG Plus toolkit [Stan02, Stan03].

The OpenGL extension NV_occlusion_query [GLNV02] is very similar to HP_OCCLUSION_TEST but it also solves the main problems with latter mechanism.

The extension provides an interface to issue several queries and requesting results later. The application can use CPU time to do other tasks before requesting results. Also the results for the first query might already be ready after the last query is issued thus keeping the parallelism between CPU and GPU intact.

NV_occlusion_query returns the amount of samples that pass the depth test. The sample count might be useful if non-conservative methods are used, or it may indicate the potential visibility of an object in next frame.

Hillesland *et al.* use NV_occlusion_query in their algorithm [Hill02].

Based on NV_occlusion_query OpenGL ARB approved ARB_occlusion_query [GLAR03] as a part of the OpenGL 1.5 core. In addition to functionality of NV_occlusion_query, ARB_occlusion_query allows a target parameter to define what kind of queries are made. At the moment SAMPLE_PASSED_ARB is the only valid value to be set as the target. SAMPLE_PASSED_ARB parameter requests the same result as the NV_occlusion_query; the number of samples passed the depth and stencil tests.

The latest version of Microsoft’s DirectX [Micr03] graphics API also provides visibility query mechanism which is practically identical to NV_occlusion_query OpenGL extension. Occlusion query is constructed using IDirect3DDevice9::CreateQuery method, which returns a pointer to interface IDirect3DQuery9 that represents the query. Through this interface queries can be issued and results requested. The result of a query is the count of visible pixel. DirectX 9 occlusion query is also asyncronous which allows other tasks to be completed before the results are requested.

5.4. Object-Space Methods

Object-space algorithms are those visibility algorithms that solve the visibility in three-dimensional continuous space where the scene is defined. Object-space algorithms usually require the scene to be constructed in specific way, typically using convex polygons.

Real-Time Occlusion Culling for Models with Large Occluders

Coorg and Teller present a conservative object-space occlusion culling method [Coor97] that is designed to accelerate visibility determination in urban and architectural environments where large occluders typically exist.

The algorithm exploits spatial coherence by organizing the scene in the kD-tree. As a pre-processing stage, polygons of the input model are organized into a kD-tree. The kD-tree subdivision is at the center of each kD-tree node. The input model is assumed to be a static set of convex polygons. During the pre-processing a small set of polygons is selected as potential occluders. At runtime the actual occluders are chosen dynamically from the set of occluders.

At runtime, the algorithm recursively traverses the kD-tree performing the conservative visibility test for each encountered node. The view frustum test is performed before the actual occlusion test. A small set of occluders is selected from the pre-processed set of potential occluders using the area-angle metric.

The actual occlusion test is performed using *supporting* and *separating planes*. A separating plane of two convex polyhedral objects is a plane formed by an edge of one object and a vertex of the other such that the objects lie on opposite sides of the plane. A supporting plane is analogous except that both objects lie on the same side of the plane.

The interaction between an occluder and a potential occludee can be described with supporting and separating planes, as depicted in figure 21. In figure 21 A is the occluder and T is the occludee (typically a bounding volume or a cell of the spatial data structure). The occlusion occurs only in the region where the viewpoint lies that half-space of A where T does not lie. This region can be subdivided into three separate regions. In the region 1, T is not occluded by A; in the region 2, T is partially occluded by A; and in the region 3, T is completely occluded by A where the full occlusion occurs.

The algorithm supports occluder fusion in the cases where a set of occluders $A_1,...,A_n$ jointly occlude T. This occurs when:

- $A_1,...,A_n$ partially occlude T, and none fully occlude T.
- If two occluders A_i and A_j share an edge E, they lie on opposite sides of E as seen from the viewpoint; and
- The signed distances of the viewpoint from all planes, other than those supporting common edges, are positive.

The algorithm exploits temporal coherence by caching supporting and separating planes in kD-tree nodes, see Section 3.7.

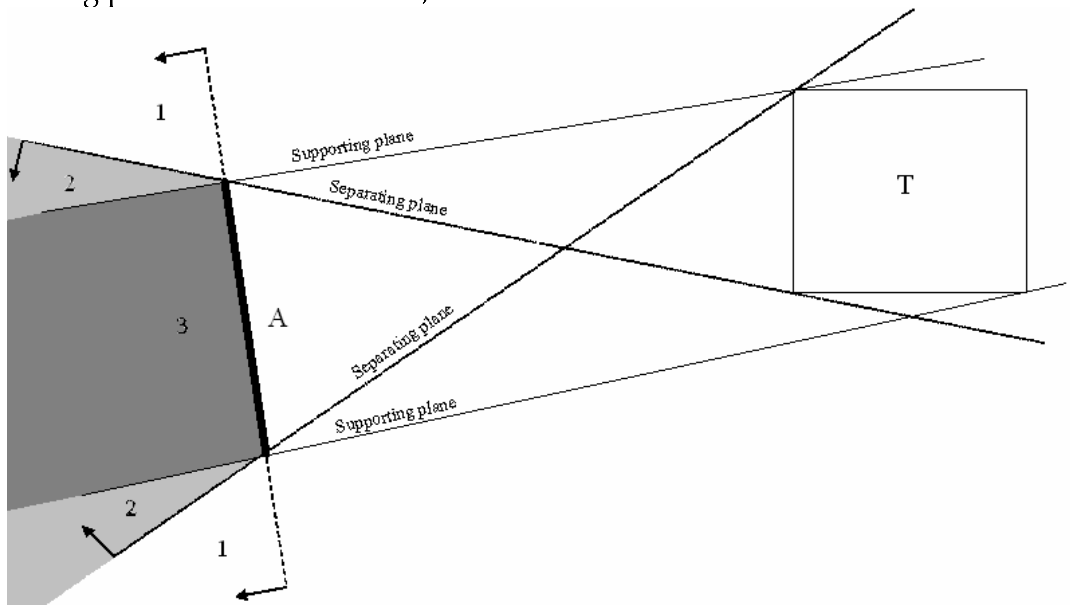

Figure 21. The occlusion of the occluder A determined by supporting and separating planes. The object T (occludee) is fully occluded only when the viewpoint is in the area 3.

Accelerated Occlusion Culling using Shadow Frusta

Hudson *et al.* present a conservative object-space occlusion culling method which uses shadow volumes to detect occluded parts of the scene [Huds97]. Shadow volumes are generated from the viewpoint using occluders as near planes of frusta.

The algorithm uses spatial subdivision to organize the scene in regions. Each region stores a list of potentially good occluders. Potential occluders that are not likely to be good when viewed from a given set of viewpoints are discarded during the pre-processing. Following criteria are used to evaluate occluders:

- Solid angle: Measures the fraction of the visual field that it occupies (Their test implementation actually uses the area-angle by Coorg and Teller [Coor97]).
- Depth Complexity: Pre-processing stage selects some random points where the shadow frusta are generated for potential occluders. Occlusion power of each potential occluder is then evaluated. The average of several samples gives an estimate of the occlusion power of an occluder.
- Coherence: The algorithm keeps track of each occluder's occlusion power per frame. It is likely that an occluder performs similarly in successive frames.

For each frame the algorithm first finds the region of the spatial data structure where the viewpoint lies. View frustum culling is first used to narrow the list of potential occluders.

For each occluder the shadow frustum is constructed. The apex of the frustum is at the viewpoint and the near plane is the plane that passes through the farthest point of the occluder's silhouette and whose normal points in the direction from that point towards the viewpoint. Each side of the frustum is a plane containing the viewpoint and an edge of the object's silhouette. Any geometry inside the shadow frusta is occluded.

The general algorithm for the interference detection between a shadow frustum and a potential occludee is performed by projecting the silhouette of the occluder and the occludee onto the image plane. Also the specialized overlap test of AABBs and OBBs are presented. The overall running time is reported to be linear in the number of faces of the shadow frustum.

Hierarchical Visibility Culling with Occlusion Trees

Bittner *et al.* [Bitt98] further developed the idea of shadow frusta. They use a data structure called occlusion tree to merge all generated occlusion volumes. The occlusion tree is a BSP-tree similar to shadow volume BSP-tree by Chin and Feiner [Chin89]. Figure 22 illustrates the idea of the occlusion tree in 2D. The algorithm is designed for environments where large occluders exist, typically architectural and indoor environments. The only requirement for the scene is that the occluders are convex polygons. In the preprocessing stage the algorithm does not create occluder database like previous methods [Coorg97, Huds97]. The knowledge of the model is used to select potential occluders. Detail objects such as flowers and chairs in a room are considered non-occluding. All other polygons are marked as potential occluders.

The scene is decomposed into a kD-tree where the splitting plane of a node is chosen so that it splits the minimum number of objects. The objects that intersect with the splitting plane are associated with both nodes without actually splitting them.

For each frame the kD-tree structure is traversed in front-to-back order. View frustum culling is first applied. Occlusion culling is applied to nodes that reside inside the view frustum.

The algorithm dynamically selects a small set of occluders for each frame. The area-angle is used to estimate the quality of an occluder. Potential occluders are selected from leaf node cells whose centers are located within a certain distance λ of the viewpoint. For each potential occluder the area-angle is computed. Area-angle values are used to select k (6-256 in [Bitt98]) occluders with the largest area-angle.

Selected occluders are used in building the occlusion tree. Each node in the occlusion tree is associated with a (shadow) plane passing through the viewpoint and an edge of the occluder. Normals of the (shadow) planes are oriented so that the shadow volume and the occluder lie in the negative half-space of the plane. Each leaf node corresponds to a semi-infinite polyhedral cell (frustum). Leaf nodes are classified as in- or out-leave. In-leaves lie inside shadow frusta. Out-leaves are not occluded by the occluder. The generated occlusion (shadow) volume is a union of all cells corresponding to in-leaves.

The occlusion tree is used to test the visibility of cells of the kD-tree. The visibility of a cell can be determined by testing the visibility of its faces. At most three rectangular polygons have to be tested to determine the visibility of a cell.

The visibility test is done by filtering the face polygons of a cell down the occlusion tree. The filtering starts from the root node and traverses towards the leaves in front-to-back order. In each node the position of a polygon with respect to node's shadow plane is determined. If the polygon is located completely on the back or front side of the node's shadow plane, it is filtered down to the back or front child of the node respectively. Otherwise the polygon is split by the shadow plane and the resulting fragments are filtered down to the both children of the node. When a fragment of a polygon reaches a leaf node, its visibility status can be classified. For out-leaves the fragment is fully visible. For fragments that reach an in-leaf the visibility has to be further examined. It may be possible that the fragment is in front of the occluder. If the fragment is completely in front of the occluder, it is completely visible. Similarly if the fragment is fully behind the occluder then the fragment is fully hidden. Otherwise the fragment is partially visible.

If all fragments of a polygon are invisible, then the polygon is invisible and thus occluded. Similarly if all fragments are visible, then the polygon is fully visible. Otherwise the polygon is partially visible. The algorithm reports conservatively all partially visible polygons as not occluded.

If all faces of a cell are determined as hidden, the cell is occluded. If a leaf cell is determined as visible or partially visible, its contents are rendered using the Z-buffer.

Bittner *et al.* also represented a modified occlusion tree. Motivation for the modified occlusion tree is to reduce the number of polygon splits during the construction of the occlusion tree. The modified occlusion tree is based on the observation that some of the shadow planes can be removed without changing the information about the occlusion volume.

The algorithm also exploits spatial and temporal coherence using methods by Bittner described in Section 3.7.

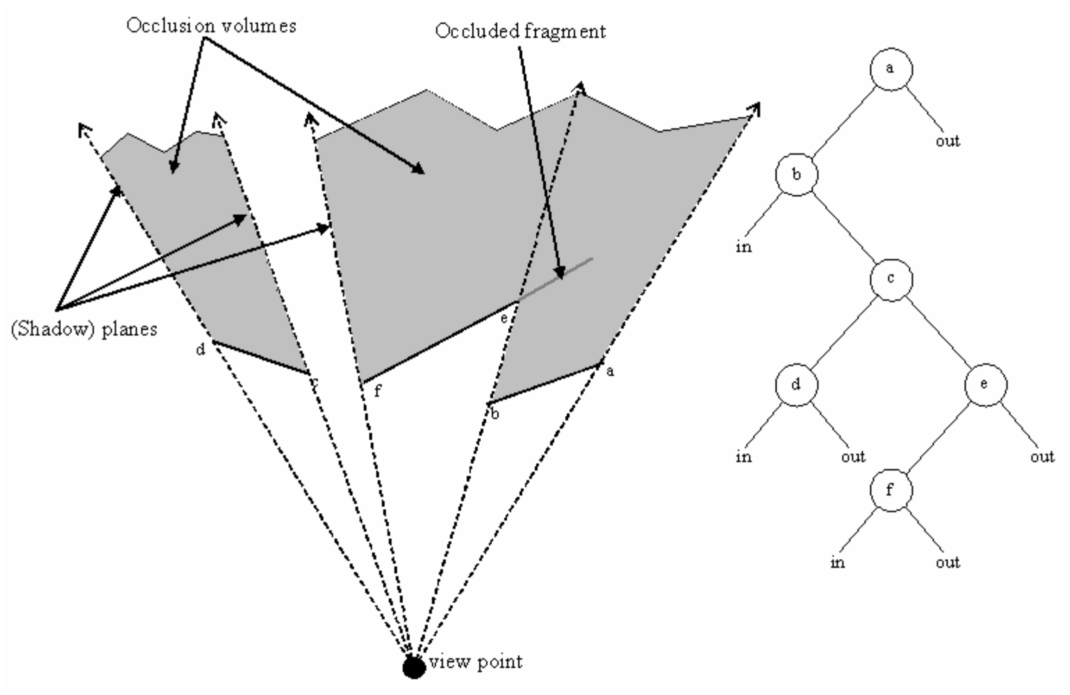

Figure 22. Gray areas are occlusion volumes formed by three occluders, and the corresponding occlusion tree diagram.

5.5. Image-Space Methods

Image-space algorithms solve the visibility in two-dimensional discrete space. Image-space algorithms typically work well with any kind of scenes and do not assume any particular rendering primitives to be used.

Unlike object-space algorithms image-space algorithms are highly robust since no computational geometry is involved. Image-space algorithms also provide occluder fusion.

Image-space algorithms do not utilize temporal coherence but they can be accompanied by additional algorithms that utilize temporal coherence, like the ones described in Section 3.7.

Hierarchical Z-Buffer Visibility

The hierarchical Z-Buffer algorithm by Greene *et al.* [Gree93] is an extension to the traditional Z-Buffer algorithm [Catm74]. The hierarchical Z-Buffer uses two hierarchical data structures; an octree to decompose the scene in object-space and the hierarchical Z-Buffer called Z-pyramid in the view-space.

The algorithm traverses the octree structure in front-to-back order and in each node the visibility of the corresponding cell is tested. First view frustum culling is performed.

The occlusion test is performed using a hierarchical Z-pyramid data structure. The Z-pyramid contains several levels of Z-buffers that differ by precision. Figure 23 illustrates a three-level Z-pyramid. The finest level of the Z-pyramid corresponds to the contents of the Z-buffer. Each coarser level in the Z-pyramid is half of the resolution of its ancestor. A z-value of the coarser level is the maximum value of corresponding four values at the finer level.

During the visibility test the faces of the node are scan converted into the frame buffer and tested hierarchically against the Z-pyramid. The first test is done using the coarsest level of the Z-pyramid. If the node is not determined to be hidden, the next level of the Z-pyramid is used for testing. The visibility of a node is determined at the finest level of z-pyramid but the non-visibility can be determined at an earlier level. If the leaf nodes are reached and determined as visible, the contained primitives are rendered and the Z-pyramid is updated.

The hierarchical Z-Buffer algorithm exploits temporal coherence by using the temporal coherence list described in Section 3.7.

5	3	5	7	6	5	5	9
4	2	5	9	2	4	3	9
5	3	7	5	1	9	5	5
2	5	2	5	2	8	8	2
2	2	5	8	4	5	5	2
7	4	1	5	3	8	2	3
2	6	8	6	3	2	9	4
5	6	7	4	3	5	3	8

Original Z-buffer

5	9	6	9
5	7	9	8
7	8	8	5
6	8	5	9

1/2 resolution

9	9
8	9

1/4 resolution

Figure 23. Hierarchical Z-buffer. Z-pyramid is formed by original Z-buffer and two coarser levels.

Hierarchical Polygon Tiling with Coverage Masks

Hierarchical polygon tiling with coverage masks by Greene [Gree96] uses recursive subdivision of image-space to resolve visibility. The image-space is hierarchically subdivided using triage coverage masks to represent the visibility states of the nodes. The hierarchy is called *coverage pyramid*. A triage mask defines the state of an image-space area as covered (C), vacant (V) or active (A). The state C indicates that the area is fully covered and thus occludes all geometry that is projected to the corresponding area. The state V indicates that the area is fully free. Partially covered areas are marked as A.

The algorithm requires the strict front-to-back order, which is achieved by organizing the scene into a BSP-tree or an octree of BSP-trees. The hierarchical culling of octree nodes can be used and each BSP-tree can be efficiently traversed in strict front-to-back order. This way there is no need to maintain depth information while rendering the polygons. First the bounding box of the polygon is tested. If the bounding box test does not determine the polygon as hidden, the polygon is tiled into the smallest enclosing cell in the coverage pyramid. When a polygon is tiled into a coverage pyramid node, the triage mask is computed for the polygon. This is done by composing together triage masks for each edges of the polygon. The triage mask of the polygon is then combined with the node. Three types of areas are identified: where the polygon is fully visible, fully hidden, or active. Fully hidden areas are ignored and the active areas are recursively subdivided. Cells where the polygon is fully visible are marked as visible and rendered. If the visibility status of a node is changed, the information is propagated upward the coverage pyramid.

Visibility Culling using Hierarchical Occlusion Maps

The visibility culling using hierarchical occlusion maps by Zhang *et al.* [Zhan97] has some similarities to Greene's hierarchical Z-buffer [Gree93]. The scene is decomposed into a bounding volume hierarchy which is traversed during the process of visibility determination. Along with the scene hierarchy is an occluder database which is constructed in the preprocessing stage. Occluders are selected from the occluder database at runtime. After view frustum culling, the occlusion culling is performed in two separate phases: the *coverage test* and *depth test*.

The coverage test determines if a primitive is covered by occluders from the current viewpoint. The coverage test is implemented using h*ierarchical occlusion maps* (HOM). The HOM is a pyramid of monochromatic frame buffers (figure

24). The lowest level of the hierarchy (level 0) has the same resolution as the final image. Occluders are first rendered into an image, which forms the lowest level of the hierarchy. Starting at the level 0 map, a hierarchy of occlusion maps is constructed by recursively applying the average operation to rectangular blocks of pixels. The result is a pyramid of occlusion maps. Pixel values in the hierarchy indicate the coverage opacity percentage of underlying pixels. If the hardware supports mipmapping, it can be used to generate the HOM.

The algorithm uses the hierarchy of occlusion maps to accelerate the overlap test. It begins the test at the level of hierarchy where the size of a pixel in the occlusion map is approximately the same size as the bounding rectangle of the tested primitive. If the coverage test shows that the object is covered by occluders, the depth test is performed. Otherwise the object is determined as visible.

The depth test determines if the primitive is behind occluders. The depth test uses a data structure called d*epth estimation buffer* (DEB). It is a software buffer that conservatively estimates the depth of occluders (figure 25). DEB is first initialized to the nearest possible value (as opposite to Z-buffer). A DEB update occurs if the current value is greater than the value stored in DEB. DEB has usually a lower resolution than the resolution of the frame buffer.

For each frame occluders are first rendered to HOM and DEB. Occluders are selected from the occluder database.

The algorithm considers each object in the occluder database lying in the viewing frustum. The distance between the viewer and the center of an object's bounding volume is used as an estimate of the distance from the viewer to the object. The algorithm sorts these distances, and selects the nearest objects as occluders until their combined polygon count exceeds a predefined threshold.

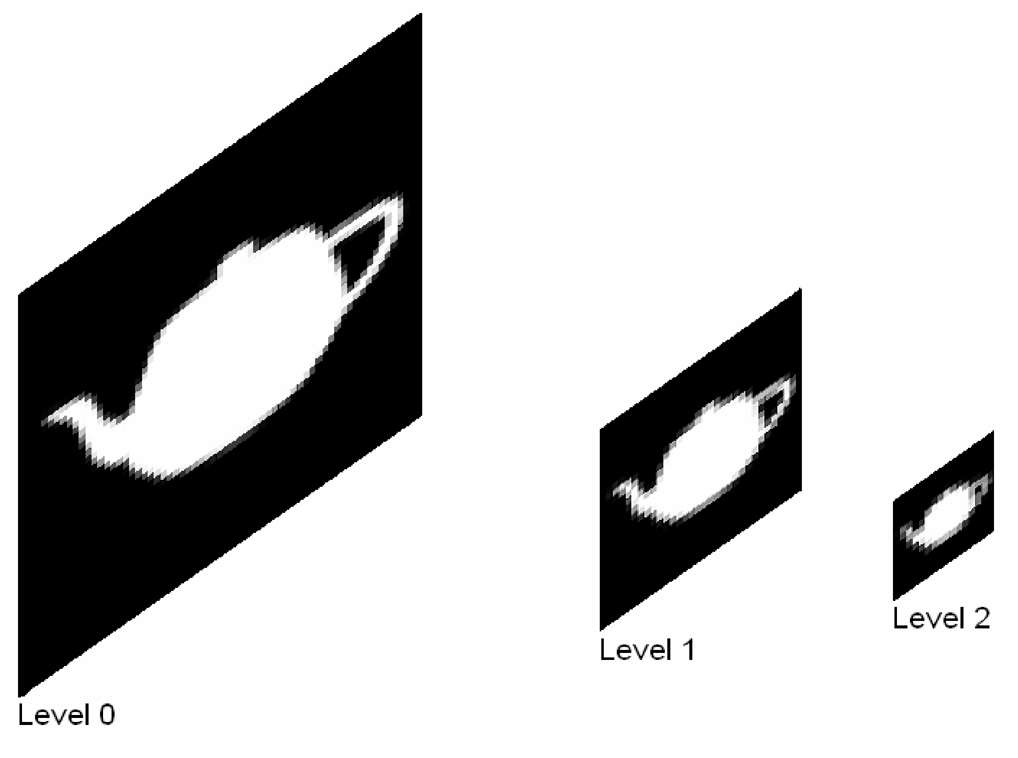

Figure 24. Three level hierarchical occlusion map (HOM).

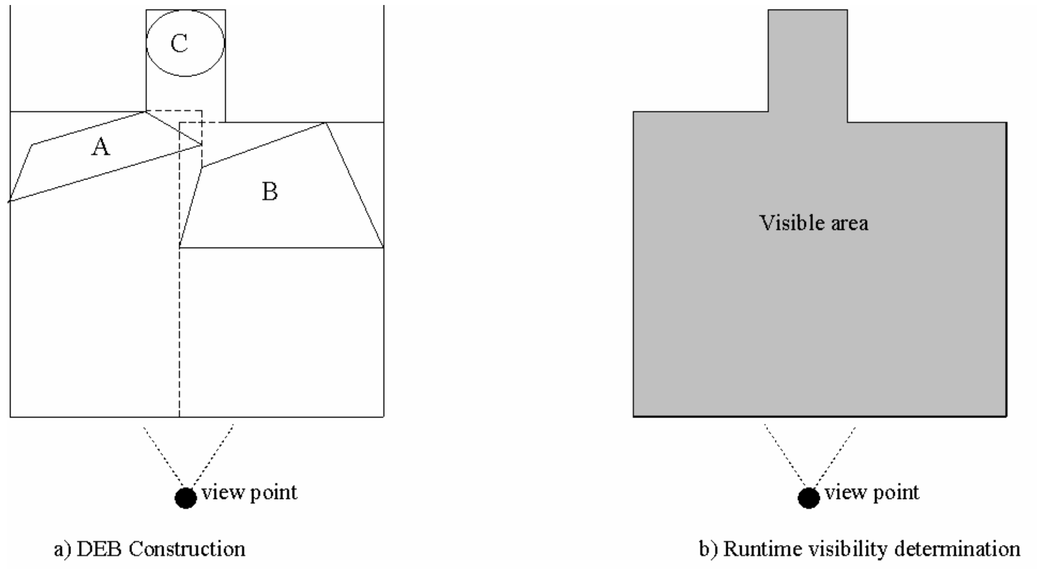

Figure 25. Conservative depth estimation of the DEB.

Incremental Occlusion Maps

Hierarchical occlusion maps algorithm by Zhang *et al.* [Zhan97] renders all occluders to HOM once per frame. After the rendering of occluders, the frame buffer is downloaded to the main memory where occlusion tests are done. The algorithm could be modified to be incremental but it would cause the frame buffer to be downloaded to the main memory several times during the frame. Using the current hardware, this would cause significant overhead. Incremental occlusion maps (IOM) [Aila01] is an incremental variant of the HOM algorithm. The IOM pipeline consists purely of per-object operations. The software implementation makes it possible to incrementally add more occluders when needed. This feature also gives a possibility to evaluate the occlusion power of an occluder.

The coverage test is done using a coverage buffer which is a three-level hierarchy of buffers. The resolution of the finest level of the hierarchy is same as the resolution of the frame buffer. The resolution of second level buffer is $\frac{1}{8x8} *$ resolution of the finest level. The resolution of the third level is $\frac{1}{32x64} *$ resolution of the finest level.

For each occluder the silhouette is filled to a cache. Occluder fusion is done by "ORing" the cache and the coverage buffer. If a block in the coverage buffer changes during the occluder fusion, the corresponding depth value is updated.

The depth estimation buffer (DEB) in IOM is hierarchical whereas the DEB in HOM is flat. Hierarchical DEB results in faster depth queries for objects with the large screen coverage. DEB is updated if the silhouette at least partially covers the block and if the silhouette actually contributes to the corresponding coverage buffer block. DEB updates in IOM pipeline are less conservative than in HOM because HOM algorithm updates the depth values of all blocks that are at least partially covered by the axis-aligned rectangle.

Incremental Occlusion Maps is part of the Hybrid's visibility library called dPVS [Aila01].

OpenGL-assisted occlusion culling for large polygonal models

Both view frustum culling and occlusion culling are performed using OpenGL functions in the algorithm by Bartz *et al.* [Bart99]. The hardware acceleration is expected also in low-end graphics systems because only common OpenGL features are used.

The algorithm exploits spatial coherence by subdividing the static parts of the scene in a data structure called the sloppy n-ary space-partitioning tree (snSP-tree). The snSP-tree is a hierarchical data structure which allows space partitioning of sibling nodes to be overlapping. Also the number of child nodes of a node is not predefined. Sibling nodes are allowed to overlap to prevent possible object splitting. The number of child nodes is determined by a cost function. Dynamic objects are represented by leaf nodes only.

View frustum culling is implemented using the OpenGL selection mode [Neid03]. The selection mode is designed to identify geometric objects rendered into a specific area of the screen (in this case, the whole screen). The geometry of a bounding volume is rendered without contributing to the frame buffer. If the bounding volume intersects the view frustum, the hit buffer of the selection mode contains the information about the bounding volume. If the bounding volume resides entirely inside the view frustum, all corresponding subnodes are marked as potentially visible. Otherwise subnodes are recursively tested. Even if no contribution in the hit buffer is found, it may be possible that the bounding volume fully contains the view frustum or the bounding volume is between the viewpoint and the near plane. These situations can be tested on software. If neither of these cases is true, the bounding volume is marked as hidden.

Occlusion culling is implemented using a virtual occlusion buffer that is mapped onto the OpenGL frame buffer. Bertz *et al.* used the stencil buffer to implement the virtual occlusion buffer.

The occlusion test for an object-space hierarchy node is performed by sending the geometry of node's bounding volume to the OpenGL pipeline. The pipeline uses the Z-buffer [Catm74] test while scan converting the geometry, and redirects the output into the virtual occlusion buffer. If the bounding volume does not contribute to the virtual occlusion buffer, it is determined as occluded. Results of the test are retrieved by reading the virtual occlusion buffer from the hardware.

The reading of the virtual occlusion buffer is estimated to take about 90% of the total time of the occlusion stage. To reduce the overall time of the occlusion stage, the occlusion test is performed progressively reading spans of pixels from the virtual occlusion buffer using double interleaving scheme. This progressive sampling introduces non-conservatism; sometimes a bounding volume is determined as occluded, when it is actually only partially occluded. To improve the performance of occlusion culling, Bertz *et al.* [Bart98] propose an extension for visibility queries within OpenGL.

Real-time occlusion culling with a lazy occlusion grid

Hey *et al.* [Hey01] present a conservative image-space occlusion culling algorithm that uses low resolution grid on top of the Z-buffer [Catm74]. The resolution of the grid is system dependent. They choose to use a 2D grid structure instead of a hierarchical representation like the HOM [Zhan97]. The grid is updated in lazy manner; results are calculated only when they are requested. The number of expensive pixel level occlusion queries is reported to be significantly fewer than by updating in busy manner. The algorithm can use hardware accelerated per pixel queries if the suitable hardware is available. The algorithm uses a hierarchical subdivision of the scene that provides approximate front-to-back traversal.

After view frustum culling the occlusion culling stage is performed using a low resolution grid. Every cell in the grid stores an occlusion state flag. If the whole area of a cell is covered by already drawn geometry, the cell is marked as occluded. Otherwise the state of the cell can completely be free or partially free. The occluded cell allows fast rejection of geometry. The grid is based upon a conventional Z-buffer or an occlusion-buffer where a single bit per pixel shows if the pixel is free or occluded.

Occlusion culling using lazy updates of the grid has following steps:

1. Test the visibility of bounding volume of the object.
2. If the bounding volume is potentially visible, render the object using Z-buffer. If the bounding volume is occluded, the object is culled.
3. If the object is determined as potentially visible, all cells that intersect with the object's image area are marked as outdated so that next time the occlusion state of the outdated cell is queried, the query will be updated on pixel level.

The two variants of the grid are represented; occlusion-buffer variant and Z-buffer variant. In the Z-buffer variant each cell stores the farthest Z-value of its pixels and the state flag. The Z-buffer variant is intended for systems that support pixel-level occlusion queries on hardware or fast reading access to the Z-buffer.

In the occlusion-buffer variant each cell stores a single bit per pixel to indicate if the pixel is occluded of not. The occlusion-buffer variant is for systems that do not support previously mentioned features. The occlusion-buffer variant can also be used with the conventional Z-buffer.

When an occlusion query for a bounding volume is made, the cells that intersect with the image area of the bounding volume are identified. If all cells are marked as occluded, then the bounding volume is occluded. If the bounding volume intersects with cells that are partially visible, pixel-level queries have to be issued for intersecting pixels.

Fast and simple occlusion culling using hardware-based depth queries

Hillesland *et al.* [Hill02] use hardware occlusion queries to conservatively cull hidden parts of the scene. Occlusion culling is implemented using OpenGL extension NV_occlusion_query [GLNV02].

The algorithm exploits spatial coherence by using spatial subdivision of the scene. The uniform grid and the nested grid versions are presented. The scene decomposition enables the approximate front-to-back traversal of the scene which is essential to incrementally add new occluders as the rendering of the frame proceeds.

The set of visible objects of the scene is first limited with view frustum culling. Cells of the grid are processed in slabs and in front-to-back order. A slab is a pair of parallel planes [Kay86]. The algorithm uses slabs that are equivalent to rasterized planes approximately orthogonal to the view vector. The visibility of cells is first tested with NV_occlusion_query and if determined as visible rendered.

The efficiency of the NV_occlusion_query depends on how full the graphics pipeline can be kept. The result of one query is available after it is transformed and rasterized. If only few queries can be issued, it may cause the pipeline to stall because the application has to wait for the results. It is likely that when multiple queries are issued results of the first ones are available when the last query is sent to the hardware.

The following algorithm tries to keep the pipeline as full as possible:

```
For each slab where a slab is a collection of cells:
1. Get next n cells within the slab where n is the maximum number of
   occlusion queries that may be in the pipeline at one time.
2.For $i = 1$ to $n$
    • Render $C_i$ query geometry (z and color writes off)
3.For $i = 1$ to $n$
    • Get Results of query for $C_i$ query geometry
    • If $C_i$ is available:
          Render the model geometry associated with cell $C_i$
```

The threshold of the number of triangles per cell is counted. If the number of triangles in a cell is below the threshold the geometry inside the cell is rendered without testing it first.

The efficiency of the occlusion is also presented comparing the number of triangles rendered using only view frustum culling, view frustum culling and occlusion culling and the actual number of visible triangles evaluated using the item buffer [Neid03]. For large models the results show significant culling of hidden geometry using occlusion culling compared to view frustum culling alone. The conservativity of the algorithm leads also to higher triangle count compared to actually visible triangles.

5.6. Summary

Object-space algorithms typically assume that the scene contains a relatively small set of large occluders that occlude most of the scene. This limits applicable environments mainly to indoor and architectural environments. Object-space algorithms are also limited to polygonal data, often to convex polygons.

Object-space algorithms also provide occluder fusion poorly, or not at all, which usually increases the conservativity of the algorithm. The advantage of object-space algorithms is the potential ability to exploit spatial and temporal coherence [Aila01].

Image-space algorithms are commonly considered as more robust than object-space algorithms, since no computational geometry is involved. Typically image-space algorithms do not have any restrictions on scene structure.

Image-space algorithms also inherently support occluder fusion on the view plane which is not very strongly present in object-space algorithms.

Software implementations of image-space algorithms [Gree93, Gree96] are typically too slow to have practical meaning. An exception is IOM implementation in dPVS by Hybrid [Aila01]. On the other hand, software implementations allow fast access to occlusion presentation, allowing the incremental addition of occluders.

Some of the image-space algorithms [Bart99, Hey01, Zhan97] try to use common hardware features to accelerate occlusion culling. The problem with these algorithms is that they all need to download the frame buffer to obtain the occlusion information. The downloading stalls the rendering pipeline for a long period if time. In method by Bartz *et al.* [Bart99] the downloading is reported to take up to 90% of the time used in occlusion culling. The incremental addition of occluders cannot be used efficiently with algorithms that need to download the occlusion information to main memory. To be efficient, these algorithms have to be able to select all significant occluders at once.

The latest hardware [Micr03, Open03] supports occlusion queries which allow the hardware acceleration of occlusion culling and incremental addition of occluders. In fact, algorithms based on occlusion queries do not have to perform the occluder selection at all [Hill02].

6. Conclusion

There are many approaches to solve the visibility problem in computer graphics. Bittner [Bitt02] distinguishes three different approaches to solve the visibility: from-region, from-point and along-line.

Methods that solve the visibility from a region usually need a time-consuming preprocessing stage but are practically free at runtime. From-region methods estimate the set of visible objects from a region. At runtime the region where the viewpoint resides is located and the PVS [Aire90] information of that region is used to render the scene. From-region methods are suitable for environments that are mostly static, where the depth complexity is significant or large occluders exist. Indoor environments, like Quake [Abra97], are typically used with from-region methods, like PVS. From-region methods usually do not work well in environments that are highly dynamic or the occlusion of the scene is based on the large number of small objects, for example a forest scene.

Methods that solve the visibility along a line like ray tracing and ray casting are typically used with some global illumination method. They are usually not used in real-time graphics, but in realistic rendering.

Methods that solve the visibility from point are typically used along with from-region algorithms in real-time graphics. They usually require less preprocessing than the from-region methods, and are therefore potentially suitable for dynamic scenes.

Visibility algorithms can be classified into two categories, exact and conservative algorithms. Exact algorithms guarantee that every pixel in the final image is calculated correctly. The Z-buffer [Catm74] algorithm is the most used exact visibility algorithm in real-time graphics. Algorithms that solve the visibility along line are also often exact, for instance a ray tracing algorithm that shoots a ray for each pixel.

Conservative methods are used to accelerate the rendering and to decrease the workload of the exact visibility algorithm. Conservative methods overestimate the set of visible objects, and cull the invisible geometry away before they are rendered. Conservative methods typically first try to determine non-visibility, and if this fails the visibility is a conclusion of the test. Most of the from-region and from-point algorithms are conservative. On the other hand, most of the exact visibility algorithms solve the visibility from-point or along line.

Games have traditionally used from-region visibility algorithms with preprocessed PVS information [Abra97]. The lack of processing power and appropriate graphics hardware has forced games to have mostly static environments where dynamic objects are not part of the spatial database. This of course reduces the immersion.

This thesis concentrated on the subset of the visibility algorithms, conservative from-point methods. Conservative from-point methods can be also separated in two classes of algorithms, view frustum culling and occlusion culling algorithms.

Chapter 3 reviewed often used spatial data structures, grids, octrees, kD-trees and BSP-trees. Most of the popular spatial data structures are hierarchical. Hierarchical subdivision of the scene space allows logarithmic search of the visible objects and allows output-sensitivity. Output-sensitivity is a requirement for an efficient visibility algorithm. A spatial data structure can be used to describe the whole scene, like in Quake [Abra97], or it can be used as an auxiliary database along with a scene graph.

How the spatial data structure is built determines whether the spatial data structure can be updated at runtime. If the scene contains multiple dynamic objects the spatial data structure needs to be updated when a dynamic object changes its position. If the spatial data structure maps objects on object level, it can be updated at runtime [Aila01, Bata01, Suda96, Shag03]. If the scene is fully static, it has traditionally been more efficient to map objects on primitive level in the spatial data structure. BSP-trees have typically been used to map every polygon rather than objects [Abra97].

Modern graphics hardware uses commonly vertex buffers [Open03, Micr03] to store the rendered geometry of the scene. Vertex buffers can reside on the main memory, in the AGP-memory or in the memory of the video cards [Rigu02]. Where vertex buffers reside is not always entirely up to the programmer. Current graphics APIs [Open03, Micr03] usually manage resources. Vertex buffers that are marked as read-only are managed by the API entirely, and are the fastest to render. If the spatial data structure maps every primitive (triangle) of the scene, vertex buffers have to be constructed for every frame and uploaded to the video memory. If the mapping is done on the object level, vertex buffers can be often constructed at the startup. In some cases looser object level mapping can yield to faster rendering due to the hardware features of the modern graphics systems.

All hierarchical data structures reviewed in Chapter 3 provide logarithmic search property. They differ in how and when the subdivision is done. The simplest is the octree. Only the decision of halting the division process has to be made. Octree subdivision uses always the center point of the cube for subdivision. A more adaptive subdivision is allowed in the kD-tree data structure. Partition is still always axis-aligned in the kD-tree. The most flexible partition scheme is in the BSP-trees. The space is divided into two subspaces with an arbitrary oriented plane. BSP-trees allow every polygon of the scene to be mapped and thus allows exact front-to-back traversal. Octrees and kD-trees are typically used in object level mapping [Aila01].

The rendering process can be accelerated by exploiting various coherences. Chapter 3 reviewed various methods to exploit spatial and temporal coherences. Also different approaches to update and manage dynamic scenes are reviewed.

Chapter 4 reviewed view frustum culling techniques. View frustum culling can be considered as a basic method to conservatively cull hidden geometry. Practically all virtual environment applications benefit from view frustum culling. If the whole scene is in the view frustum, then view frustum culling may slow down the rendering process. Usually this is not the case in games and virtual environments. Also several optimizations for the view frustum culling is reviewed in Chapter 3.

Whereas the backface culling and view frustum culling are very precisely defined tasks, occlusion culling is more vaguely defined. In many cases the intimate knowledge of the scene objects is required to select good occluders [Bitt98].

There are currently only few commercial applications that use occlusion culling [Aila01]. In theory from-region methods could be replaced with an occlusion culling algorithm. This would avoid time-consuming preprocess stage. If the occlusion culling algorithm does not need significant preprocessing also dynamic scenes can be supported.

Occlusion culling algorithms are traditionally classified into two categories, object-space algorithms and image-space algorithms, depending on the space in which the occlusion test is performed.

Object-space algorithms do not typically provide good occluder fusion. Occluder fusion is a very important property since rarely one occluder occludes objects fully. Object-space methods are also limited to convex polygonal data, where large occluders exist, such as indoor environments. They are hard to accelerate using the current hardware, and are more prone to computational errors compared to image-space algorithms.

Almost all features that object-space algorithms are lacking are present in image-space algorithms. They inherently support occluder fusion and do not restrict the rendering primitives because the occlusion is determined in the discrete image-space. Image-space methods are also commonly described more robust than object-space methods [Aila01].

Some of the image-space algorithms [Bart99, Hey01, Zhan97] reviewed in this thesis attempt to benefit from graphics hardware by moving some of the operations to the hardware. The common problem with all these methods is the significant performance loss when the frame buffer is downloaded from the video card. Bartz *et al.* [Bart99] report that the downloading takes up to 90% of the time needed for the algorithm. Their method tries to reduce the time by reading the frame buffer in spans which causes non-conservatism. It is no surprise that the only well known commercial occlusion culling system, dPVS [Aila01], is implemented purely in software.

The latest generation of graphics hardware and graphics APIs [Micr03, Open03] has introduced occlusion queries. Occlusion queries are now supported also at the entry-level graphics cards. Chapter 6 reviewed currently available occlusion queries. Through these occlusion queries practical applications using the occlusion culling may become more common.

Some of the major problems with occlusion culling are occluder selection and occluder evaluation. Existing occluder selection methods are reviewed in Chapter 6. As Aila and Miettinen [Aila01] point out, the occluder evaluation and the feedback from the algorithm are very important in efficient occlusion culling. Occlusion queries of the current generation of graphics hardware do not provide much information that can be used for evaluation of occluders.

Future Work

If the hardware occlusion queries achieve the same kind of standard status in the field of conservative visibility determination methods as the Z-buffer algorithm in the field of exact visibility determination algorithms, the focus of visibility studies will probably move away from the actual occlusion test. As this study clearly shows operations such as occluder selection and occluder evaluation are far from trivial and no exact algorithms exist. Also techniques related to the management of dynamic scenes could be a good choice for a research. Methods to use temporal bounding volumes [Suda96] are also very general and no exact algorithm is provided either in Sudarsky's PhD Thesis or in other research papers that study the use of TBVs [Aila01, Bata01].

References

[Abra97] Michael Abrash, *Michael Abrash's Graphics Programming Black Book, Special Edition*. Coriolis Group Books, 1997.

[Aila01] Timo Aila, Ville Miettinen. dPVS Reference Manual Version 2.10. Hybrid Holding Ltd. October 2001.
Available at http://www.hybrid.fi/dpvs_download.html.

[Aire90] John M. Airey, John H. Rohlf, and Frederick P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. In: *Proceedings of Symposium on Interactive 3D Graphics*, pp. 41–50, ACM SIGGRAPH. March 1990.

[Assa99] Ulf Assarsson, Tomas Möller,
Optimized view frustum culling algorithms. Technical report, March 2000. *http://www.ce.chalmers.se/staff/uffe/vfc.pdf*.

[Bart98] Dirk Bartz, Michael Meißner, Tobias Hüttner. Extending graphics hardware for occlusion queries in OpenGL. In: *Proceedings of the 1998 Workshop on Graphics Hardware,* Lisbon, Portugal, pp. 97–104. 1998.

[Bart99] Dirk Bartz, Michael Meißner, Tobias Hüttner. OpenGL-assisted occlusion culling for large polygonal models. *Computers and Graphics* **23**, 5, pp. 667–679. October 1999.

[Bata01] Harlan Costa Batagelo, Shin-Ting Wu. Dynamic scene occlusion culling using a regular grid. In: *conference proceedings of Sibgrapi 2002*, pp. 434. *http://www.dca.fee.unicamp.br/~ting/Publications/P2001-2005/batagelo-wu-2002-3dvis.pdf*.

[Bent75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. In *Communications of the ACM* **18**, 9, pp. 509–517. September 1975.

[Bitt98] Jiri Bittner, Vlastimil Havran, Pavel Slavík. Hierarchical visibility culling with occlusion trees. In: *Proceedings of Computer Graphics International '98 (CGI'98*), pp. 207–219, IEEE. 1998.

[Bitt01] Jiri Bittner and Vlastimil Havran. Exploiting temporal and spatial coherence in hierarchical visibility algorithms. In: *Proceedings of Spring Conference on Computer Graphics (SCCG'01*), pp. 13–220, IEEE Computer Society, Budmerice, Slovakia. 2001.

[Bitt02] Jiri Bittner. Hierarchical Techniques for Visibility Computations. Ph.D. Dissertation. Department of Computer Science and Engineering. Czech Technical University in Prague, December 2002.

[Catm74] Edwin Catmull. A Subdivision algorithm for computer display of curved surfaces. PhD Thesis. Dept. of Computer Science, University of Utah, Salt Lake City, Utah, 1974.

[Chin89] Norman Chin and Steven Feiner. Near real-time shadow generation using BSP trees. *Computer Graphics (Proceedings of SIGGRAPH '89)*, pp. 99–106. 1989.

[Coor97] Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. *In Proceedings of the Symposium on Interactive 3D Graphics*, pp. 83–90, ACM Press. April 1997.

[Diet01] Sim Dietrich. Optimizing for hardware transform and lighting. 2001. A presentation at Xtreme Game Developers Conference. http://developer.nvidia.com/attach/1677.

[Dura00] Frédo Durand, George Drettakis, Joelle Thollot, and Claude Puech. Conservative visibility preprocessing using extended projections. *Computer Graphics (SIGGRAPH '00 Proceedings)*, pp. 239–248, 2000.

[Dura96] Frédo Durand, George Drettakis, and Claude Puech. The 3D visibility complex: a new approach to the problems of accurate visibility. In: *Proceedings of Eurographics Workshop on Rendering*, pp. 245–256, Eurographics, Springer Wein, June 1996.

[Dura97] Frédo Durand, George Drettakis, and Claude Puech. The visibility skeleton: a powerful and efficient multi-purpose global visibility tool. *Computer Graphics (SIGGRAPH '97 Proceedings)*, pp. 89–100, 1997.

[Eber00] David H. Eberly. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*. Morgan Kaufmann, September 2000.

[Fole90] James D. Foley, Andries van Dam, Steven. K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice, Second Edition*. Addison-Wesley Publishing Co., 1990.

[Fuch80] Henry Fuchs, Zvi M. Kedem, and Bruce Naylor. On visible surface generation by a priori tree structures. *Computer Graphics* **14**, 3, pp. 124-133, 1980.

[Funk93] Thomas A. Funkhouser, Carlo H. Sequin, Seth J. Teller. Database and display algorithms for Interactive visualization of architectural models. PhD thesis, CS Division, UC Berkeley, 1993.

[GLHP] GL_HP_occlusion_test extension specification. Silicon Graphics Inc. *http://oss.sgi.com/projects/ogl-sample/registry/HP/occlusion_test.txt*

[GLNV02] GL_NV_occlusion_query extension specification. Silicon Graphics Inc. February 2002. *http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion_query.txt*.

[GLAR03] GL_ARB_occlusion_query extension specification. Silicon Graphics Inc. June 2003. http://oss.sgi.com/projects/ogl-sample/registry/ARB/occlusion_query.txt.

[Grap94] Paul S. Heckbert (Editor). *Graphics Gems IV, Book and Disk edition*. Morgan Kaufmann, January 1994.

[Gree93] Ned Greene, Michael Kass, Gavin Miller. Hierarchical Z-Buffer visibility. *Computer Graphics (SIGGRAPH '93 Proceedings)*, pp. 231–238, 1993.

[Gree96] Ned Greene. Hierarchical polygon tiling with coverage masks. *In H. Rushmeier, Ed., Computer Graphics (SIGGRAPH '96 Proceedings), pp. 65–74*, New Orleans, Louisiana, 04-09 August 1996.

[Grib01] Gil Gribb, Klaus Hartmann. Fast Extraction of Viewing Frustum Planes from the World-View-Projection Matrix. 2001. Available at http://www2.ravensoft.com/users/ggribb/plane%20extraction.pdf.

[Java03] Java 3D API. Sun Microsystems Inc. May 2003. Available at http://java.sun.com/products/java-media/3D/.

[Havr98] Vlastimil Havran, Jirí Bittner, Jirí Zára. Ray tracing with rope trees. *In Proceedings of 13th Spring Conference on Computer Graphics*, pp. 130–139, Budmerice. 1998.

[Hey01] Heinrich Hey, Robert F. Tobler, Werner Purgathofer. Real-time occlusion culling with a lazy occlusion grid. *In Rendering Techniques (Proceedings of Eurographics Workshop on Rendering '00*), pp. 217–222, 2001.

[Hill02] Karl Hillesland, Brian Salomon, Anselmo Lastra, and Dinesh Manocha. Fast and simple occlusion culling using hardware-based depth queries. Technical Report TR02-039, UNC Chapel Hill, 2002.

[Hoff96a] Kenneth E. Hoff, A "fast" method for culling of oriented-bounding boxes (OBBs) against a perspective viewing frustum in large "walkthrough" models. Technical report, University of Carolina, 1996. http://www.cs.unc.edu/~hoff/research/index.html.

[Hoff96b] Kenneth E. Hoff, A Faster Overlap Test for a Plane and a Bounding Box. Technical report, University of Carolina, 1996. http://www.cs.unc.edu/~hoff/research/index.html.

[Hoff97] Kenneth E. Hoff, Fast AABB/View Frustum Overlap Test. Technical report, University of Carolina, 1997. http://www.cs.unc.edu/~hoff/research/index.html.

[Huds97] Tom Hudson, Dinesh Manocha, Jonathan Cohen, Ming C. Lin, Kenneth E. Hoff III, and Hansong Zhang. Accelerated occlusion culling using shadow frusta. *In Proceedings of ACM Symposium on Computational Geometry*, pp. 1–10, 1997.

[Kay86] Timothy L. Kay, James Kajiya. Ray tracing complex scenes. *Computer Graphics* **20**, 4, August 1986, pp.269-278.

[Kolt00] Vladlen Koltun and Daniel Cohen-Or. Selecting effective occluders for visibility culling. *Eurographics 2000 (short presentations track*), pp. 165-169. 2000.

[Lars03] Bent Dalgaard Larsen. Real-time terrain rendering using smooth hardware optimized level of detail. Technical University of Denmark, Department of Informatics and Mathematical Modelling. 2003. Available at http://www.imm.dtu.dk/pubdb/views/edoc_download.php/1425/pdf/imm1425.pdf.

[Lueb95] David Luebke and Chris Georges. Portals and mirrors: simple, fast evaluation of potentially visible sets. In: *Proceedings of Symposium on Interactive 3D Graphics '95*, pp. 105–106, ACM SIGGRAPH, April 1995.

[Micr03] DirectX 9 SDK.
Microsoft Corporation, 2003. *http://msdn.microsoft.com/directx*.

[More00] Steve Morein. ATI Radeon HyperZ Technology.
In SIGGRAPH Eurographics Graphics Hardware Workshop, Hot3D Session, 2000. Available at *http://www.merl.com/hwws00/presentations/ATIHot3D.pdf*.

[Möll99] Tomas Möller, Eric Haines, *Real-time Rendering*. A K Peters Ltd., 1999.

[Nayl93] Bruce F. Naylor. Constructing Good Partitioning Trees. In: Proceedings of Graphics Interface '93, pp. 181--191. Available at http://www.graphicsinterface.org/pre1996/93-Naylor.pdf.

[Nayl98] Bruce F. Naylor. A Tutorial on binary space partitioning trees. Proceedings of Computer Games Developer Conference, pp 433-457, 1998.

[Neid03] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide: The Official Guide to Learning OpenGL 1.4, Fourth Edition*. Addison-Wesley, November 2003.

[NVID02] NVIDIA Lightspeed Memory Architecture II. Technical brief. NVIDIA, January 2002.
Available at *http://www.nvidia.com/object/techbrief_lmaii.html*.

[Open03] The OpenGL Graphics System:
A Specification (Version 1.5). Silicon Graphics Inc., 2003. *http://www.opengl.org/developers/documentation/OpenGL15.html*

[Paja98] Renato Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In: *Proceedings IEEE Visualization '98*, pp. 19-26 & 515, 1998.

[Rigu02] Guennadi Riguer. Performance Optimization Techniques for ATI Graphics Hardware with DirectX® 9.0. December 2002. Available at *http://www.ati.com/developer/dx9/ATI-DX9_Optimization.pdf*.

[Same90a] Hanan Samet. *Design and Analysis of Spatial Data Structures*. Addison–Wesley, Redding, MA, 1990.

[Same90b] Hanan Samet. *Applications of Spatial Data Structures*. Addison–Wesley, Reading, MA, 1990.

[Shag03] Joshua Shagam, Joseph Pfeiffer, Jr. Dynamic irregular octrees. Technical report, New Mexico State University, 2003.

[Stan02] Dirk Staneker, A first step towards occlusion culling in OpenSG PLUS. In: *1. OpenSG Symposium Darmstadt*, 2002. Also available in *http://www.opensg.org/OpenSGPLUS/symposium/Papers2002/Staneker_OcclusionCulling.pdf*.

[Stan03] Dirk Staneker, An occlusion culling toolkit for OpenSG PLUS. *In Proceedings of OpenSG 2003 Symposium*, 2003. Also available in *http://www.eg.org/EG/DL/PE/OPENSG03/09staneker.pdf*.

[Suda96] Oded Sudarsky, Craig Gotsman. Output-sensitive visibility algorithms for dynamic scenes with applications to virtual reality. *Computer Graphics Forum*, 15, 3 (Proceedings of Eurographics), August 1996.
Available at http://www.cs.technion.ac.il/~sudar/cv_eng.html.

[Suda98] Oded Sudarsky. Dynamic scene occlusion culling. PhD thesis, Technion–Israel Institute of Technology, January 1998. Available at http://www.cs.technion.ac.il/~sudar/cv_eng.html.

[Suth74] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A characterization of ten hidden-surface algorithms. In: *ACM Computing Surveys*, **6**, 1, pp. 1–55, March 1974.

[VRML03] VRML 97/X3D Specifications. Web3D Consertium. June 2003. *http://www.web3d.org/x3d.html*.

[Watt99] Alan H. Watt. *3D Computer Graphics (3rd Edition)*. Addison-Wesley, December 1999.

[Zhan97] Hansong Zhang, Dinesh Manocha, Tom Hudson, and Kenneth E. Hoff III. Visibility culling using hierarchical occlusion maps. *Computer Graphics (Proceedings of SIGGRAPH '97)*, pp. 77–88, 1997.

[Zhao01] Zhao YouBing, Zhou Ji , Shi JiaoYing , Pan ZhiGeng. A fast Algorithm for large scale terrain walkthrough. In: *International Conference on CAD&Graphics*. 2001, China.