

# Dynamic Occlusion Culling

A. Julian Mayer\*  
TU Wien

## Abstract

Visibility computation is one of the fundamental problems in 3D computer graphics, and due to ever-increasing data-sets its relevance is likely to increase. One of the harder problems in visibility is occlusion culling, the process of removing objects that are hidden by other objects from the viewpoint. This paper gives an overview of occlusion culling algorithms and the data structures they operate upon. Special emphasis is given to dynamic occlusion, which enables real-time modification of the scene, and the supporting dynamic data structures.

**CR Categories:** I.3.7 [Computer Graphics]: Picture/Image Generation—Viewing algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Hidden line/surface removal; E.1 [Data]: Data Structures—Trees

**Keywords:** visibility, occlusion culling, data structures, dynamic occlusion, dynamic visibility, dynamic data structures

## 1 Introduction

In 3-dimensional rendering typically only a small subset of the scene is visible from a given viewport at any time. The determination of this visible subset is part of the visibility problem, and has two solutions: visible surface determination and hidden surface determination. While visible surface determination tries to determine the visible-set without touching all the geometry, hidden surface determination starts with the full scene and removes invisible parts until it ends up with approximately the visible-set. Hidden surface determination typically contains these stages:

- **View frustum culling:**

The viewing frustum is a geometric representation of the virtual camera's view volume. View frustum culling removes objects that are completely outside of the viewing frustum since they naturally can not be possibly visible. View frustum culling is a very important step because it removes a very significant portion of the geometry, but it has to be manually handled by the developer at the geometry level. The algorithm used is typically dependent on the spatial data structure used. Objects that are only partially in the viewing frustum can be clipped against it, though this is normally handled automatically at a lower level.

- **Occlusion culling:**

Occlusion culling is the process of removing objects that are hidden by other objects from the viewpoint. Occlusion

culling is "global" as it involves interrelationship among polygons and is thus far more complex than backface and view-frustum culling. It has historically necessitated expensive preprocessing steps, which largely prevented dynamic scenes. This article focuses on the algorithms used for occlusion culling and the structures they operate upon.

- **Backface culling:**

Objects are usually represented by their hollow shell, which is composed of polygons that are defined in a special order (clockwise or counter-clockwise) so that their front-side can be distinguished from their back-side. There is normally no reason to draw polygons that are facing the virtual camera with their back-side. Culling those polygons is also the cause for objects becoming invisible if the virtual camera (accidentally) gets inside them. Backface culling can remove about 50 percent of all geometry and nowadays does not need developer-intervention since it happens at the graphics-library level.

- **Exact Visibility Determination:**

Exact Visibility Determination resolves a single or multiple contributing primitives for each output image pixel - the closest object at any given screen pixel should be drawn [Aila 2000]. This is today handled in hardware by the z-buffer, in the past developers were forced to do it themselves, e.g. using the Painters Algorithm.

- **Contribution culling:**

Contribution culling is the process of removing objects that would not contribute significantly to the image, due to their high distance or small size. This is usually measured with their screen projection. Contribution culling can also happen as part of the occlusion culling step, e.g. when using occlusion queries, which return the exact number of pixels an object would take up on screen. "Whereas the other four sub-categories leave the output image intact, contribution culling introduces artifacts. The term aggressive culling is sometimes used to refer to such non-conservative methods." [Aila 2000] It is also common to discard far-away objects regardless of their size and draw e.g. fog instead, though advances in graphics hardware have made this measure less necessary. Another common technique is to replace objects by representations with a lower level of detail (LoD) when they are far away.

The problem of the 3D-visibility from a specific point (camera) described is only a part of visibility problem, there are other parts like the visibility from a region [Bittner and Wonka 2003], which is needed for pre-computing a potentially visible set (see section 4.2).

"The goal of visibility culling is to bring the cost of rendering a large scene down to the complexity of the visible portion of the scene and mostly independent of the overall size. Ideally, visibility techniques should be output sensitive: the running time should be proportional to the size of the visible set. Several complex scenes are 'densely occluded environments', where the visible set is only a small subset of the overall environment complexity. This makes output-sensitive algorithms very desirable. Visibility is not an easy problem since a small change in the viewpoint might cause large changes in the visibility." [Cohen-Or et al. 2000]

---

\*e-mail: julianmayer@mac.com

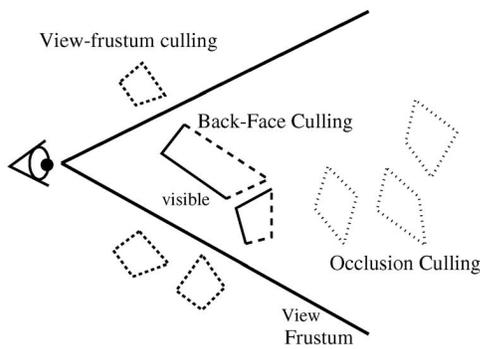


Figure 1: A sample scene that explains view-frustum-, back-face- and occlusion-culling [Cohen-Or et al. 2000].

The problem of dynamic occlusion culling is the focal point of this paper and can be divided into two smaller problems. First, dynamically updating a spatial data structure with minimal overhead, when scene updates happen. Second, doing dynamic occlusion culling on top of this structure. This involves amongst others the independence from any pre-computed information that would be invalidated during scene updates. Consequently this paper has the two major chapters "Spatial Data Structures" and "Occlusion Culling Algorithms", analogous to these two problems.

## 2 Related Work

Visibility is a well covered field [Cohen-Or et al. 2000] [Bittner and Wonka 2003] [Hadwiger and Varga 1997].

Spatial data structures are covered in books [Samet 1990b] [Samet 1990a] and also their relationship to occlusion-culling [Helin 2003] and its performance [Meißner et al. 1999] is evaluated.

Heinrich Hey and Werner Purgathofer give an overview over various occlusion culling methods [Hey and Purgathofer 2001] while others explore certain methods in depth [Bittner et al. 1998] [Zhang et al. 1997] [Hua et al. 2002] [Ho and Wang 1999].

David Luebke and Chris Georges documented the possibility of using portals to speed-up rendering [Luebke and Georges 1995].

Pre-computed occlusion information has long been the method of choice for speeding up rendering and so their generation is well-understood [Laine 2005] [Koltun et al. 2000]. S. Nirenstein and E. Blake proposed an interesting way to speed-up this expensive operation using hardware-acceleration [Nirenstein and Blake 2004].

Occlusion queries may be the method of choice for occlusion culling in the future since they are part of consumer graphics hardware since a few years now. However, due to their nature it is immediately obvious that a straightforward and simple approach will not bring the desired performance-benefits. Therefore a large number of algorithms and techniques has been developed that use occlusion queries to get the desired acceleration [Bittner et al. 2004] [Hillesland et al. 2002] [Staneker et al. 2003] [Guthe et al. 2006] [Kovalcik and Sochor 2005] [Staneker et al. 2006] [Sekulic 2004]. Maybe a combination of some of these ideas will bring the best results.

While occlusion queries provide the possibility for dynamic scene updates (in opposition to pre-computed approaches like the PVS), these can only happen if the underlying spatial data structure supports dynamic updates. These dynamic structures are notably cov-

ered by Joshua Shagam and Joseph J. Pfeiffer, Jr. [Shagam 2003] [Pfeiffer et al. 2003].

Dynamic occlusion of course does not have to depend on occlusion queries, H. Batagelo and S. Wu present an output-sensitive occlusion culling algorithm for densely occluded dynamic scenes [Batagelo and Wu 2001]. Oded Sudarsky and Craig Gotsman generalize existing occlusion culling algorithms, intended for static scenes, to handle dynamic scenes while also having a look at dynamically updating an octree-structure [Sudarsky and Gotsman 1999].

## 3 Spatial Data Structures

### 3.1 Overview & Features

There are numerous ways to represent and store the geometrical shape of fictional objects or approximations of real-world objects in a computer-system, depending on the desired application: point-clouds, edge-lists, volume-data (CSG, ) or mathematical curves and surfaces. Here we will focus on the representation through the triangles and polygons that approximate the surface of an object, since this is the predominant solution for realtime and non-realtime graphics alike. It is also versatile, applicable to a wide array of use-cases and its use is also supported and accelerated through current 3D-graphics-hardware. So, we are mostly covering the structures below in the context of storing polygons, even if many of them are also suited to store other objects or primitives.

It is obvious that storing the polygons that make up a scene in a simple (linear) list is unsuited to any but the simplest task, because there is no efficient way to get the spatial information that is necessary not only for the various culling stages in the graphics pipeline, but also for things like collision-detection or polygon-sorting.

Since there are many different possible structures their various features have to be compared to the unique needs in the desired application. Typical properties are the space complexity (in memory and on disk), the time complexity for generation, dynamic updates and the various spatial queries, as well as code complexity necessary to support its use. Because of their unique characteristics only hierarchical subdivision structures are suitable for most applications: if one item is invisible, then all of its children are invisible as well. Given similar characteristics it may be desired to choose the structure that is easier to understand or has a more familiar mental model.

"Instead of having all the geometry in one, unbounded space, one might want to subdivide the space into smaller segments or voxels as they are called [...]. It should be noted that the [spatial] subdivision can always be done to the plain geometry (splitting the polygons) or e.g., bounding boxes (adding the pointer of the object to each voxel it touches), depending on the application and requirements. [...] One should also note that it is possible and often very useful to use multiple and concurrent spatial subdivisions for a scene. E.g., the static geometry could be placed inside a BSP-tree while a loose octree could be used to store the dynamic objects." [Helin 2003] It should also be noted that it is possible to combine spatial data structures, e.g. an octree can stop subdividing when a certain threshold is reached and store the content data in a small BSP-tree, at each leaf-node.

## 3.2 Non-Hierarchical Grids

### 3.2.1 Uniform Grids

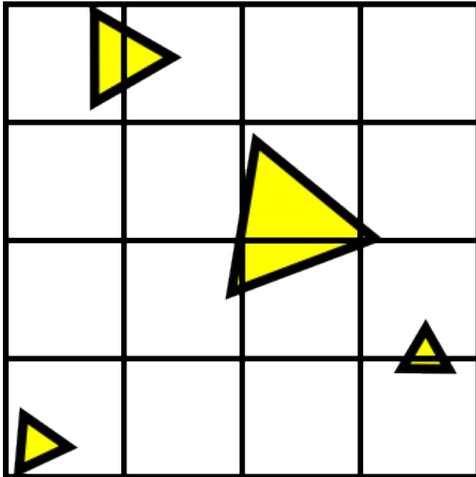


Figure 2: An uniform grid (2-dimensional).

"We place a regular grid over the scene and assign the geometry to the grids voxel that encloses its center point. Afterwards we can find all the geometry in voxel  $n$  by indexing the grid. One could also clip the polygons to fit perfectly inside the voxels, but this would increase the amount of polygons. Then again, one could assign the polygons to every voxel they intersect. [...] Regular grids are suitable for dynamic scenes when used with bounding volumes for the objects, because relocating geometry is easy: just remove it from the current voxel and place it into the new one. Regular grids work best when the geometry is uniformly distributed in the scene." [Helin 2003] "It involves the subdivision of initial cell into equally sized elementary cells formed by splitting planes that are axis-aligned regardless of the distribution of objects in a scene. [...] The  $n$ -dimensional grid resembles the subdivision of a two-dimensional screen into pixels. A list of objects that are partially or fully contained in the cell is assigned to each parallelepiped cell (also called a voxel in IE3 space). [...] Since the uniform grid is created regardless of the occupancy of objects in the voxels, it typically forms many more voxels than the octree or the BSP tree, and therefore it demands necessary storage space." [Havran 2000]

The possibility of using a uniform grid for dynamic scene occlusion culling is explored by H. Batagelo and S. Wu [Batagelo and Wu 2001], but such a partitioning suffers from performance and/or scalability problems when used for extremely large environments or when there is a high variability in object density [Pfeiffer et al. 2003]. It should be noted that the time complexity for creating an uniform grid is very low, I presume it has order of  $n$ , where  $n$  is the number of geometric objects to be placed.

### 3.2.2 Non-Uniform Grids

"In the non-uniform grid the splitting planes are also axis-aligned, but along the axis they can be positioned arbitrarily. Non-uniform grids can have a better fit of voxels to the scene geometry, since for sparsely occupied scenes they put more splitting planes to the spatial regions with higher object occupancy. The positioning of the planes is performed according to the histogram of objects along the coordinate axes." [Havran 2000]

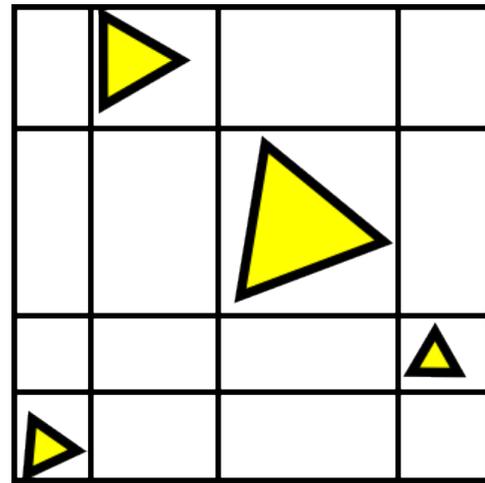


Figure 3: A non-uniform grid (2-dimensional).

The non-uniform grid trades higher cost for generating the grid for better efficiency fitting the geometry in the grid. It should be noted that algorithms that work with an uniform grid may work much slower, or not at all with a non-uniform grid [Havran 2000]. It should also be noted that it is possible to fit a 3-dimensional scene within a 2-dimensional grid, discarding one of the coordinates during placement in the grid. This can be useful when the density of objects is much lower in one dimension (e.g. many objects placed along a 2d-plane).

## 3.3 Hierarchical Grids

### 3.3.1 Recursive Grids

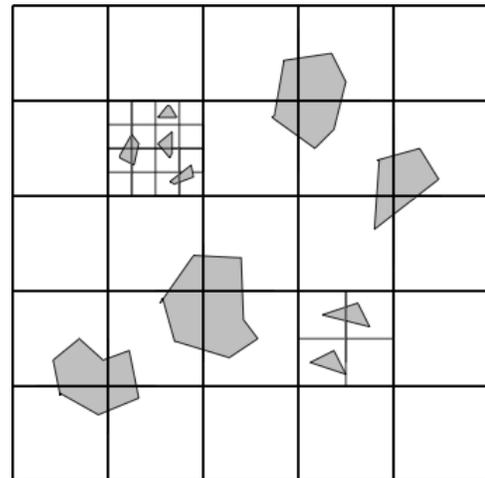


Figure 4: A recursive grid (2-dimensional) [Helin 2003].

"Recursive grids [...] simply bring the concept of recursiveness into uniform grids. The principle is as follows: construct a uniform grid over the set of objects, assign the objects to all voxels where they belong. Then recursively descend; that is, for each voxel that contains more objects than a given threshold, construct a grid again. The construction of grids is terminated when the number of objects

referenced in the voxel is smaller than a threshold or some maximum depth for grids is hit. This maximum depth is usually set to two or three. The principle is thus similar to the BSP tree or the octree, but at one step a cell is subdivided into more than two or eight child cells.” [Havran 2000]

### 3.3.2 Hierarchies of Uniform Grids

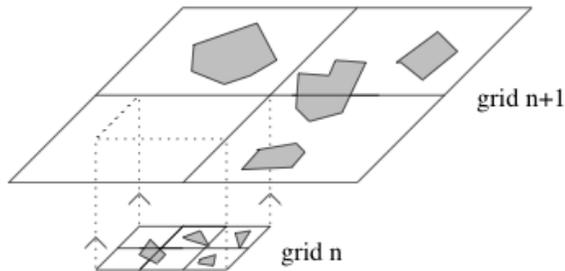


Figure 5: ”A hierarchical uniform grid. Here the all objects are on the same level in space with grid n+1, but to emphasize the fact that the grids are not spatially connected they were drawn separately.” [Helin 2003].

”Hierarchical uniform grid [...] has a different kind of approach to the spatial subdivision problem. Instead of starting by subdividing the whole scene we first group the scenes objects by their size. Next we go through all the groups and put neighboring objects of similar size into clusters. Then we use regular grids (where the voxel size depends on the size of the objects inside the cluster) to subdivide the clusters. [...] Hierarchical uniform grids have flatter hierarchy than e.g., octrees, but they are not very suitable for dynamic scenes either, unless every cluster in the scene moves as one (scaling or adding more objects might require a new subdivision or at least new uniform grids).” [Helin 2003]

### 3.4 Octrees and Quadrees

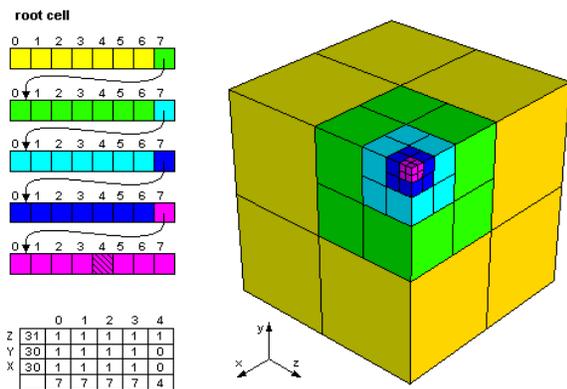


Figure 6: An octree [Stolte 2007].

Octrees and Quadrees are hierarchical subdivision structures. The initial voxel is set to the enclose all of the scenes geometry, and

is recursively subdivided into eight (octree) or four (quadtree) sub-voxels until a threshold value is met.[Helin 2003]. Typically an octree is used for dividing 3-dimensional space and a quadtree for dividing 2-dimensional space, but the quadtree can also be used in 3-dimensional space when discarding one dimension (see section 3.2.2). ”Every polygon is assigned to the smallest voxel that encloses the polygon completely. [...] Octrees subdivision places many (and smaller) voxels into areas with plenty of geometry and less (and larger) voxels into areas with only a little geometry.” [Helin 2003]

There are many variations of the octree:

- **Octree-R:** ”There is a variant of the octree called Octree-R that results in arbitrary positioning of the splitting planes inside the interior nodes. The difference between the octree and Octree-R is similar to that between the BSP tree and the kd-tree. The smart heuristic algorithm for positioning the splitting planes inside the octree interior node is applied independently for all three axes. Then the speedup between the octree-R and the octree can be from 4 percent up to 47 percent, depending on the distribution of the objects in the scene.” [Havran 2000]
- **Loose octree:** ”Loose octree, introduced by Thatcher Ulrich (2000), is like an octree where all the voxels have been subdivided into a certain level. The looseness comes from the fact that the voxels of the same level overlap each other, e.g., in an octree the voxel size would be  $N*N*N$ , but the same voxel in a loose octree would be  $2N*2N*2N$ . The objects of the 3D-scene are surrounded by a bounding sphere and depending on the spheres radius the object is positioned into a voxel on level  $p$  (the larger the radius the larger is the voxels size on level  $p$ ). Spheres center  $c$  indicates the objects position on the grid.” [Helin 2003]

The octree is not inherently suited to dynamic updates since it has to be rebuilt if an item leaves its subvoxel, but the loose octree may be better suited [Helin 2003] and the Dynamic Irregular Octree [Pfeifer et al. 2003] is specially crafted for optimized dynamic updates. Sudarsky and Craig Gotsman also manage to dynamically updating a standard octree using temporal bounding volumes [Sudarsky and Gotsman 1999]. Octrees are among the most popular, simplest and fastest spatial partitioning structures and with some updates they also seem suited for dynamic updates.

### 3.5 BSP trees

Binary space partitioning (BSP) is a method for recursively subdividing space. The recursion usually stops when one or more requirements are met, often being that each part is convex. Like the name suggests, binary space partitioning divides space using one plane into two subvoxels, compared to octrees (quadtrees), which divide space using 3 (2) planes into 8 (4) subvoxels (see section 3.4). The subdivision of the scene results in a tree data structure known as BSP tree.

Note: some literature divides BSP trees into axis-aligned and polygon-aligned versions, where the polygon-aligned version uses the plane defined by one of its polygons as splitting plane. However, since the axis-aligned version is equivalent with the kd-tree (see section 3.6), we are talking here just about the polygon-aligned version (sometimes the axis-aligned BSP tree has a constraint concerning the position of the splitting plane). [Havran 2000]

Binary space partitioning was initially used because it provides effortless back-to-front sorting of polygons for using the painter’s al-

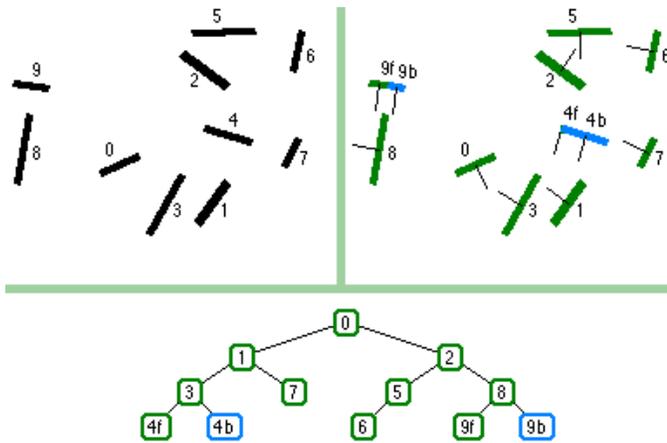


Figure 7: An example 2d-BSP-tree generated with the BSP-Java-Applet (<http://symbolcraft.com/graphics/bsp/>)

gorithm (when dividing until each subvoxel contains a convex set). The BSP tree still remains in wide use today although exact polygon sorting is less of an issue now because of the hardware-z-puffer. BSP trees are very expensive to set-up because of the long search for the best splitting plane at each step and are quite ill-suited to dynamic updates. Dynamism is often provided by using a second, concurrent (non-BSP-)hierarchy for dynamic objects.

Time complexity for creating a BSP tree ( $n$  is the number of geometric objects to be placed):

$$O = (n^2 * \lg(n)) [Ranta - Eskola2001] \quad (1)$$

Since the polygons in the scene have to be split along the splitting planes their number increases while building the BSP tree. It is important to keep this increase of the number of polygons low by choosing splitting planes that only split few polygons. In addition it is very desirable to choose each splitting plane in a way that results in an equal distribution of polygons into the two subvoxels, in order to keep the tree balanced and his depth low. Finding the best splitting-plane is usually a tradeoff between these two properties, and building the BSP tree is a very expensive process since it often involves trying all the possible choices at each step. The usefulness of the final BSP tree highly depends on these tradeoffs and the algorithm used in the generation of the tree.

### 3.6 kd-trees

A k-dimensional tree (kd-tree) is a special version of the BSP tree which only uses splitting planes that are perpendicular to the coordinate system axes. As such it is (similar to) an axis-aligned BSP tree (see section 3.5), whereas in the (polygon-aligned) BSP tree the splitting planes can be arbitrary. The distinction between different versions is based on the splitting rules: it can be necessary for the splitting plane to pass through one of the points that are to be stored (e.g. the median in the case of the balanced kd-tree), while there is no such restriction in the adaptive kd-tree [Samet 1990b]. Additionally it is possible either to cycle trough the axes used to select the splitting plane, or to select the dimension based on some criteria (like using the dimension with the maximum spread).

While a kd-tree is certainly better suited for dynamic updates than the BSP tree, like the Octree it does not seem like a inherently good

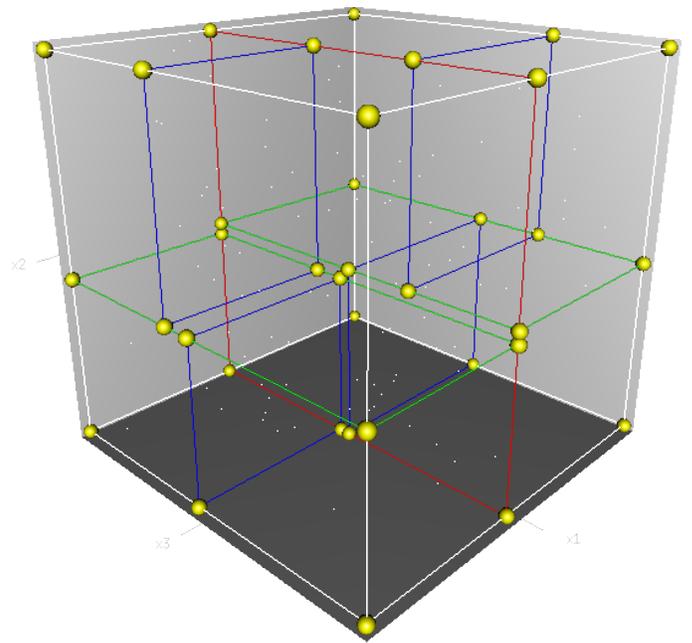


Figure 8: "A 3-dimensional kd-tree. The first split (red) cuts the root cell (white) into two subcells, each of which is then split (green) into two subcells. Finally, each of those four is split (blue) into two subcells. Since there is no more splitting, the final eight are called leaf cells. The yellow spheres represent the tree vertices." [Wikipedia 2006d]

fit either. Procopiuc et al. propose a dynamic scalable version of the kd-tree called Bkd-tree, which consists of a set of balanced kd-trees [Procopiuc et al. 2002].

Time complexity for creating a kd-tree ( $n$  is the number of geometric objects to be placed):

$$O = (n * \log(n)) [Wikipedia2006d] \quad (2)$$

### 3.7 Other spatial data structures

"A bounding volume for a set of objects is a closed volume that completely contains the union of the objects in the set. Bounding volumes are used to improve the efficiency of geometrical operations by using simple volumes to contain more complex objects. Normally, simpler volumes have simpler ways to test for overlap. [...] In many applications the bounding box is aligned with the axes of the coordinate system, and it is then known as an axis-aligned bounding box (AABB). To distinguish the general case from an AABB, an arbitrary bounding box is sometimes called an oriented bounding box (OBB). AABBs are much simpler to test for intersection than OBBs, but have the disadvantage that when the model is rotated they cannot be simply rotated with it, but need to be recomputed." [Wikipedia 2006b]

"A natural extension to bounding volumes is a bounding volume hierarchy (abbreviated to BVH [...]), which takes advantage of hierarchical coherence. Given the bounding volumes of objects, an n-ary rooted tree of the bounding volumes is created with the bounding volumes of the objects at the leaves. Each interior node [...] of [the] BVH corresponds to the bounding volume that completely encloses the bounding volumes of the subtree [...]" [Havran 2000] The interesting property of BVH is that although a bounding volume of a

node always completely includes its child bounding volumes, these child bounding volumes can mutually intersect [Havran 2000], similar to the loose octree (see section 3.4). Contrary to all the other spatial data structures discussed above, the BVH is not a spatial subdivision, and can also be constructed bottom-up, in contrast to top-down. BVH are likely to better accommodate to the underlying geometry compared to common spatial subdivisions like the kd-tree. However, a downside is that the individual bounding volumes often largely intersect, especially at the top of the hierarchy.

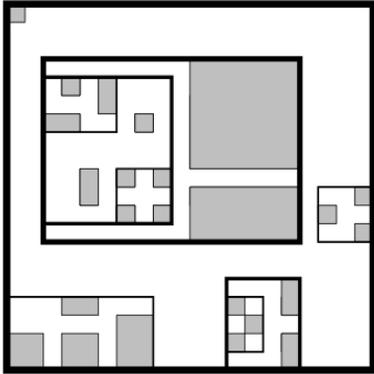


Figure 9: "A bounding volume hierarchy. Here the grey rectangles are the objects bounding rectangles. Other rectangles are the bounding rectangles constructing the hierarchy." [Helin 2003]

The AABB Tree is a special case of the bounding volume hierarchy, where AABBs are used as bounding volumes. Joshua Shagam and Joseph J. Pfeiffer, Jr. (who also invented the Dynamic Irregular Octree in the same year) thoroughly examine the possibility of using a modified AABB tree aptly called dynamic AABB tree as a fully dynamic data structure. "For the dynamic AABB tree, we don't limit the number of children that a node can have, and we allow objects to be stored in internal nodes. Furthermore, rather than build the entire tree based on static data, we only build subtrees which are actively being queried. Finally, the tree as a whole has a nesting heuristic, which dictates how nodes are split and how objects are stored in the children." [Shagam 2003] They develop and test four nesting heuristics called K-D, Ternary, Octree and Icoseptree, concluding "Based on the preceding information, it is fairly straightforward to determine that the best performance overall is with the icoseptree heuristic, and that the icoseptree heuristic is also likely to continue to scale exceptionally well for even greater orders of magnitude. [...] The modifications to the AABB tree presented here make it quite effective at efficiently determining visibility [...]" [Shagam 2003] Although they sadly don't test their structures against other dynamic structures (not even their own Dynamic Irregular Octree), their idea certainly seems worth investigating. It may be worth noting that not all (occlusion culling) algorithms may work on top of the dynamic AABB tree (without modification), since many are tailored for the standard hierarchical structures like the octree or kd-tree.

Meißner et al. explore the "Generation of Subdivision Hierarchies for Efficient Occlusion Culling of Large Polygonal Models" and present three novel algorithms for efficient scene subdivision, called D-BVS, p-HBVO and ORSD, and compare these with the subdivision of SGIs OpenGL Optimizer toolkit [Meißner et al. 1999]. While the results are quite promising they do not provide any information if their new algorithms support any dynamic scene updates. Including more existing methods for comparison would have made their results a lot more meaningful too.

We have now covered the common spatial subdivisions as well as

the bounding volume hierarchies - hybrid structures, which combine spatial subdivisions with bounding volume hierarchies, are also possible.

## 4 Occlusion Culling Algorithms

### 4.1 Overview & Features

As already explained (see section 1), occlusion culling is the process of removing parts of the scene are occluded by some other part, in order to avoid rendering them unnecessarily.

Occlusion culling is "global" as it involves interrelationship among polygons and is thus far more complex than backface and view-frustum culling. Since testing each individual polygon for occlusion is too slow, almost all the algorithms place a hierarchy on the scene, with the lowest level usually being the bounding boxes of individual objects, and perform the occlusion test top-down on that hierarchy. [Cohen-Or et al. 2000]

Occlusion culling algorithms can be classified based on numerous criteria and features:

- **Online vs. offline and point vs. region:**

One major distinction is whether the algorithm pre-computes and stores the visibility information or determines it dynamically at runtime. This is also connected to the distinction of point versus region visibility. Since it is impossible to pre-compute the information for all possible points, all those algorithms determine the information for specific regions. Online algorithms are usually point-based. It is worth noting that point-based algorithms are usually more effective in the sense that they are able to cull larger portions of the scene.

- **Image space vs. object space:**

"For point-based methods, we will use the classical distinction between object and image-precision. Object precision methods use the raw objects in their visibility computations. Image precision methods, on the other hand, operate on the discrete representation of the objects when broken into fragments during the rasterization process. The distinction between object and image precision is, however, not defined as clearly for occlusion culling as for hidden surface removal since most of these methods are conservative anyway, which means that the precision is never exact." [Cohen-Or et al. 2000]

"Image-space algorithms are usually highly robust, since no computational geometry is involved. Degenerate primitives, non-manifold meshes, T-junctions, holes or missing primitives are all handled without special treatment. Parametric surfaces and everything that can be rendered and bounded with a volume can be used, at least in theory. [...] Object-space algorithms usually have no such desirable features. What makes them important, is their theoretical possibility for utilizing temporal coherence, which is missing from every image-space algorithm." [Aila 2000]

- **Conservative vs. approximate:**

"Most techniques [...] are conservative, that is, they over-estimate the visible set. Only a few approximate the visible set, but are not guaranteed to find all the visible polygons. Two classes of approximation techniques can be distinguished: sampling and aggressive strategies. The former use either random or structured sampling (ray-casting or sample views) to estimate the visible set and hope that they will not miss visible objects. They trade conservativeness for speed

and simplicity of implementation. On the other hand, aggressive strategies are based on methods that can be conservative, but choose to lose that property in order to have a tighter approximation of the visible set. They choose to declare some objects as invisible, although there is a slight chance that they are not, often based on their expected contribution to the image.” [Cohen-Or et al. 2000]

- **Continuous vs. point sampled visibility:**

”Continuous visibility methods determine the visibility in all view directions that pass through the image, which is an infinitely large set of view directions. In contrast to that are point sampled visibility methods which determine the visibility only for a limited set of view directions, e.g. for the centers of all pixels in the image. Point sampling can also be used for object space occlusion culling.” [Hey and Purgathofer 2001]

- **Individual vs. fused occluders:**

”Given three primitives, A, B, and C, it might happen that neither A nor B occlude C, but together they do occlude C. Some occlusion-culling algorithms are able to perform such an occluder-fusion, while others are only able to exploit single primitive occlusion. Occluder fusion used to be mostly restricted to image-precision point-based methods.” [Cohen-Or et al. 2000]

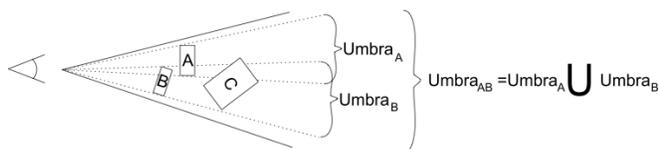


Figure 10: ”The principle of occluder fusion. Objects A and B can together occlude object C, even though neither of them can occlude C on their own.” [Aila 2000]

”Point sampled image space occlusion culling methods implicitly support occluder fusion because in their image space occlusion information they do not distinguish between different occluders. Therefore the occluders are automatically combined without having to do additional computations for the occluder fusion. Occluder fusion is very important for occluders like trees, because each single leaf of a tree usually occludes only very few objects, if any, behind it. But all the leaves of the tree together can represent an important occluder that occludes many objects behind it. Occlusion culling methods which do not support occluder fusion can usually only be used efficiently in restricted scenes which contain objects that are large enough to represent strong occluders.” [Hey and Purgathofer 2001]

- **All versus a subset of occluders:**

Either all objects or just a subset can be used as occluders. Using all objects instead of a subset has the advantage of maximized occlusion. Occluder selection usually happens heuristically, selecting objects that are assumed to occlude large parts of the scene (typically big objects are selected). Additionally, simplified representations of the occluders can be synthesized at this step. [Hey and Purgathofer 2001] Occluder selection often necessitates pre-processing, which may prevent dynamic scene-updates.

- **Virtual occluders:**

”Law and Tan made a superb observation that any hidden object or volume in the scene can act as an occluder. Occluding geometry can be replaced with considerably simpler virtual occluders. A classic example is a dense forest where visibility

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

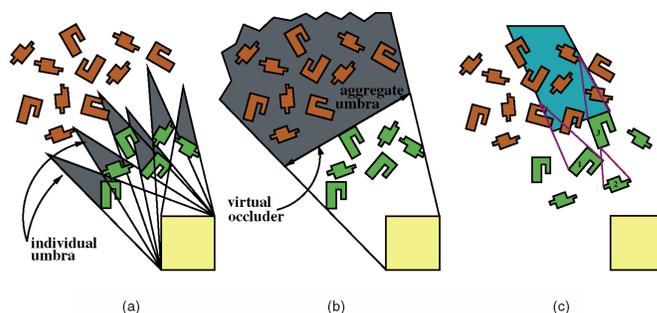


Figure 11: ” (a) The union of the umbrae of the individual objects is insignificant. (b) But, their aggregate umbra is large and can be represented by a single virtual occluder. (c) The individual umbrae (with respect to the yellow viewcell) of objects 1, 2, and 3 do not intersect, but yet their occlusion can be aggregated into a larger umbra.” [Cohen-Or et al. 2000]

- **Supported scenes:**

”Although it is desirable to support general scenes, many occlusion culling methods are nevertheless restricted to certain types of scenes. Visibility pre-computation methods are restricted to mainly static scenes. Occlusion culling methods which use portals for their visibility calculation, usually require architectural environments. Several methods are restricted to terrains, which are usually based on a height field, and several other methods are restricted to walkthroughs in urban environments or to 2.5D scenes, which are modeled on a ground plan. But of course also the (in)ability of a method to use all objects as occluders and to support occluder fusion decides whether the method is suitable for general scenes or not.” [Hey and Purgathofer 2001]

- **Use of coherence:**

”Scenes in computer graphics typically consist of objects whose properties vary smoothly. A view of such a scene contains regions of smooth changes (changes in color, depth, texture, etc.) at the surface of one object and discontinuities between objects. The degree to which the scene or its projection exhibit local similarities is called coherence. Coherence can be exploited by reusing calculations made for one part of the scene for nearby parts. Algorithms exploiting coherence are typically more efficient than algorithms computing the result from the scratch. [...] three types of visibility coherence:

- **Spatial coherence:** Visibility of points in space tends to be coherent in the sense that the visible part of the scene consists of compact sets (regions) of visible and invisible points.
- **Image-space, line-space, or ray-space coherence:** Sets of similar rays tend to have the same visibility classification, i.e. the rays intersect the same object.
- **Temporal coherence:** Visibility at two successive moments is likely to be similar despite small changes in the scene or a region/point of interest.

The degree to which an algorithm exploits various types of

coherence is one of the major design paradigms in research of new visibility algorithms. The importance of exploiting coherence is emphasized by the large amount of data that need to be processed by the current rendering algorithms.” [Bittner and Wonka 2003]

• **Other criteria:**

- “Some methods are restricted to 2D floorplans or to 2.5D (height fields), while others handle 3D scenes.” [Cohen-Or et al. 2000]
- “Convexity of the occluders can be required by some methods (typically for object-precision methods).” [Cohen-Or et al. 2000]
- “Several occlusion culling methods require that a certain kind of bounding volumes or spatial subdivision structure is used.” [Hey and Purgathofer 2001]
- “Many occlusion culling methods require that the scene is traversed in a front to back order to make efficient occlusion culling possible.” [Hey and Purgathofer 2001]
- Output sensitivity is a desired attribute for an occlusion culling algorithm - the running time should be proportional to the size of the visible set.
- Some algorithms depend on graphics hardware, some on a hardware z-buffer and some on hardware occlusion-query support.
- Another classification-factor is the degree of overestimation in conservative algorithms or the number of missed objects in an approximate algorithm.

Despite all these criteria, if we look at the aim of occlusion culling, that is culling away geometry to save time when rendering, we see that the most important feature of an occlusion culling algorithm is its performance, which could be defined as the ratio of the number of culled objects to the time needed. If the algorithm is too slow, we might even lose time instead of saving time by using it. Of course the algorithm also has to fit the scene and data structure in use as well as meet the other requirements.

While usage of the hardware z-buffer is sometimes considered as an occlusion culling technique, this is certainly false. The z-buffer is a tool for exact visibility, and while it prevents occluded geometry from ending up on screen, it does nothing to speed up rendering by omitting occluded geometry. Even the early-z-rejection found on newer hardware (designed to prevent texture and shader lookups for geometry that will not end up on screen) is no substitute for real occlusion culling, because it will not reject polygons until the rasterization level.

## 4.2 (Pre-computed) Potentially Visible Set

The Potentially Visible Set (PVS) is a superset of the visible set, typically calculated from a region (cell). While the term is also used to describe a dynamically obtained superset of the visible set (compare “conservative visibility set” [Cohen-Or et al. 2000]) we are referring here to the pre-computed PVS that is calculated from preprocessed visibility information and stored alongside the scene. Using the PVS is strictly speaking not part of the occlusion culling stage of the hidden surface removal, because we do not start out with the full set, but instead just with the Potentially Visible Set, which is then reduced by frustum culling. Normally occlusion culling takes place after this step. However, the outcome is similar: (most) occluded geometry is not considered for drawing on

screen. The PVS works on the cells that are defined by the spatial data structure and for every cell a list of all other cells that are potentially visible is produced. Generating this extensive information usually takes very long upfront (and also takes much space to store), but increases the rendering performance significantly. Referring to our classification, the PVS is a offline and region-based algorithm, other properties may depend on the exact algorithm used.

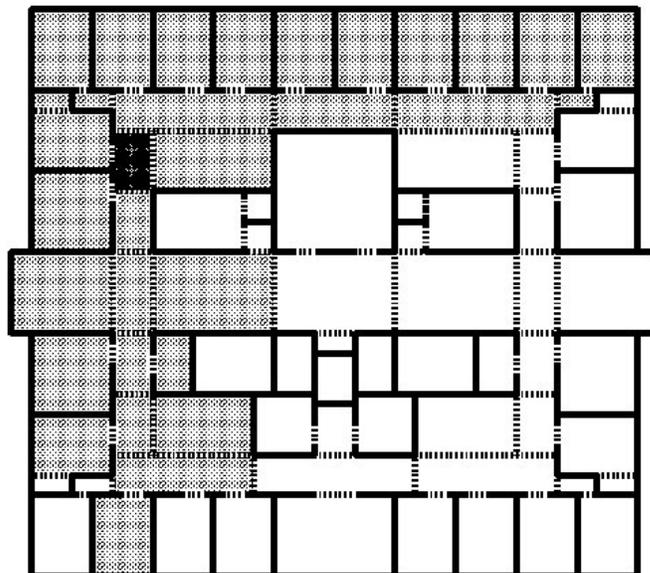


Figure 12: A demonstration of the idea of the PVS. Only the grey cells are potentially visible from the black cell.[Kortensjan 2001]

A similar technique to the PVS is the Global Occlusion Map, which results in a more compact representation of the occlusion information [Hua et al. 2002]. Kortun et al. research the possibility of using virtual occluders as “An Efficient Intermediate PVS representation” [Kortun et al. 2000]. S. Nirenstein and E. Blake proposed an interesting way to speed-up the expensive PVS-generation using hardware-acceleration [Nirenstein and Blake 2004].

Naturally using pre-computed information eliminates the possibility for dynamic updates to the scene, so it is covered here only for comparison and completeness. Since the pre-computed PVS is defined on complete cells it is often significantly bigger than the visible set from a specific point, even in indoor scenes where the technique works best. Despite all its shortcomings it has been in wide use in the past.

## 4.3 Portals

Like the PVS (see section 4.2) Portals are not strictly a occlusion culling technique, but yield similar results to it. Portals are not even part of the hidden surface determination, but instead a method of the visible surface determination (see section 1), which calculates the visible objects, instead of culling away the hidden objects. This distinction is quite artificial, most papers count portals to the occlusion culling algorithms and many do not make a distinction between visible and hidden surface determination (it seems the terminology in the field of visibility is hardly standardized). However, using portals for the rendering achieves what we are having in mind in this section, not considering occluded geometry for drawing. So now, what are portals exactly?

Indoor scenes are often composed of rooms that are connected by tight doors, resulting in heavy occlusion. The portal system divides the scene in cells (typically the rooms) that are connected by so called portals (typically the doors). "A cell is a polyhedral volume of space; a portal is a transparent 2D region upon a cell boundary that connects adjacent cells. Cells can only 'see' other cells through the portals. [...] Given such a spatial partitioning of the model, we can determine each frame what cells may be visible to the viewer." [Luebke and Georges 1995] The portal algorithm flags the current cell (the cell the camera is located in) for rendering, determines which adjacent portals are in the viewing frustum, and flags their connected cells for rendering too. The algorithm now clips the viewing frustum against the visible portals and recurses to the connected cells (since it is possible to see the portal that is connected to another room). While this system usually overestimates the visibility and does not account for occlusion due to objects in the cells or due to the shape of the cells, it yields a usually small number of visible cells to be rendered.



Figure 13: A picture designed to explain the concept of portals, cells and mirrors. [Luebke and Georges 1995]

A nice property of the portal system is that mirrors are easily integrated, they are treated as portals that transform the attached cell about the plane of the mirror [Luebke and Georges 1995]. Following this thought it is also possible to introduce portals that let you see into far-away cells, by using a different transformation.

Referring to our classification, portals are online, conservative, point-based and operate in object-space. Regarding occluder fusion: "Cell and portal methods are a special case since they consider the dual of occluders, openings, implicitly fusing together multiple walls." [Cohen-Or et al. 2000]

The portal system is quite nicely suited for dynamic scenes, with some restrictions. The contents of cells may change, as long as they are still connected to their portals. It is possible to introduce a flag to portals to define if they are opened or closed, this may be toggled at runtime. Arbitrary portals can be created, although it may be difficult to determine the destination cell, e.g. when breaking through a wall. [Tyberghein 1998]

The portal system has its drawbacks too, it is really only suited to tight indoor scenes, therefore it is no universal solution. Furthermore the portals must either be manually placed or a portal placement algorithm has to be developed, which often gives suboptimal

results and takes long to compute.

#### 4.4 Hierarchical Z-Buffer & Hierarchical Occlusion Maps

"The Hierarchical Z-buffer (HZB) is an extension of the popular HSR method, the Z-buffer. [...] It uses two hierarchies: an octree in object-precision and a Z-pyramid in image-precision. The Z-pyramid is a layered buffer with a different resolution at each level. At the finest level, it is just the content of the Z-buffer; each coarser level is created by halving the resolution in each dimension and each element holding the furthest Z-value in the corresponding 2x2 window of the finer level below. This is done all the way to the top, where there is just one value corresponding to the furthest Z-value in the buffer. During scan-conversion of the primitives, if the contents of the Z-buffer change, then the new Z-values are propagated up the pyramid to the coarser levels." [Cohen-Or et al. 2000]

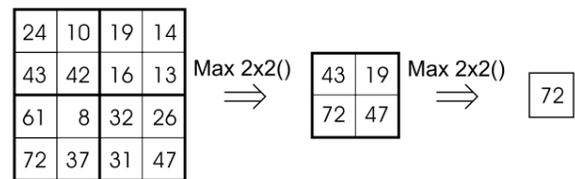


Figure 14: "Hierarchical Z-buffer principle." [Aila 2000]

The HZB requires modifications to graphics hardware, which unfortunately do not exist, and software based implementations are slow. However: "An optimized version of the hierarchical z-buffer has been proposed that allows to integrate a hierarchical z-buffer stage into the rendering pipeline of conventional graphics hardware. [...] Adaptive hierarchical visibility is a simplified one layer version of the hierarchical z-buffer where bucket sorted polygon bins are rendered and occlusion tested. It is simpler to implement in graphics hardware than the hierarchical z-buffer." [Hey and Purgathofer 2001]

As proposed, the HBZ requires an octree as spatial subdivision method, although it could be adapted to other spatial structures [Hey and Purgathofer 2001].

"The hierarchical occlusion map method is similar in principle to the HZB, though it was designed to work with current graphics hardware. In order to do this, it decouples the visibility test into an overlap test (do the occluders overlap the occludee in screen space?) and a depth test (are the occluders closer?). It also supports approximate visibility culling; objects that are visible through only a few pixels can be culled using an opacity threshold. The occlusion is arranged hierarchically in a structure called the Hierarchical Occlusion Map (HOM) and the bounding volume hierarchy of the scene is tested against it. However, unlike the HZB, the HOM stores only opacity information, while the distance of the occluders (Z-values) is stored separately. The algorithm then needs to independently test objects for overlap with occluded regions of the HOM and for depth." [Cohen-Or et al. 2000]

"An opacity map can be used instead of a hierarchical occlusion map. Whereas the the hierarchical occlusion map resembles a pyramid of mipmapped textures, the opacity map corresponds to a summed area table which allows to perform an overlap test in constant time. The generation of the summed area table has to be done in software." [Hey and Purgathofer 2001] One drawback of hierarchical occlusion maps is that they need a preprocessing step to identify potential occluders (which prevents dynamic scenes), but

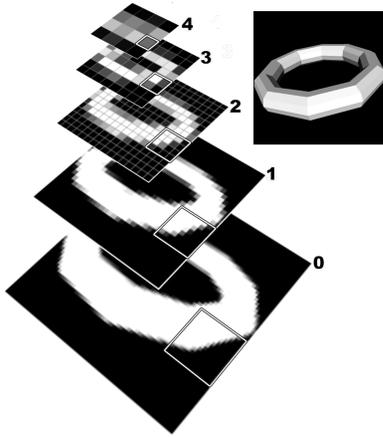


Figure 15: "The hierarchy of occlusion maps. This particular hierarchy is created by recursively averaging over 2 blocks of pixels. The outlined square marks the correspondence of one top-level pixel to pixels in the other levels. The image also shows the rendering of the torus to which the hierarchy corresponds." [Zhang et al. 1997]

this can be avoided with incremental occluder selection [Hey and Purgathofer 2001] as it is used for incremental occluder maps [Aila 2000].

## 4.5 Occlusion Queries

Occlusion Queries are an occlusion culling feature that is built right into the graphics hardware. Occlusion queries are likely to end up as the predominant technique, like the z-buffer which is now the de-facto standard exact visibility technique because of its broad hardware support. Hardware support for occlusion queries is common since the 2001 launch of the ATI Radeon 7500/8500 and the NVIDIA GeForce 3, on the software side it is supported by the OpenGL ARB\_occlusion\_query extension and DirectX 9. Fundamentally, occlusion queries are quite simple: after issuing a query for a specific object or its bounding box the GPU answers exactly how many pixels should have ended up visible on-screen. This allows not only for occlusion culling, but also enables us to discard an object if its visible pixels are below a specific threshold (contribution culling) or substitute the rendered object with a simpler version (LoD). The catch? The naive approach to using occlusion queries is not necessarily faster, but can even be slower than rendering without it, because the CPU has to wait until the GPU finishes drawing all triangles (those given before the test and all that are part of the test) until it gets its answer, resulting in CPU stalls and GPU starvation. There is also another problem with the basic approach, bounding boxes cannot write to the depth buffer, since it is possible that the bounding box of object A completely occludes the bounding box of object B, but the real object A does not occlude object B. Since the naive approach does not work well, the real question is how to use occlusion queries efficiently. [Sekulic 2004]

However, if we can find an efficient algorithm for using occlusion queries there are several advantages to their use (besides those already mentioned):

- Generality of occluders:  
"We can use the original scene geometry as occluders, since the queries use the current contents of the z-buffer."
- Occluder fusion:

"The occluders are merged in the z-buffer, so the queries automatically account for occluder fusion. Additionally this fusion comes for free since we use the intermediate result of the rendering itself."

- Generality of occludees:  
"We can use complex occludees. Anything that can be rasterized quickly is suitable."
- Exploiting the GPU power:  
"The queries take full advantage of the high fill rates and internal parallelism provided by modern GPUs."
- Simple use:  
"Hardware occlusion queries can be easily integrated into a rendering algorithm. They provide a powerful tool to minimize the implementation effort, especially when compared to CPU-based occlusion culling."

[Bittner et al. 2004]

### 4.5.1 Efficient Occlusion Queries

Dean Sekulic gives in his article "Efficient Occlusion Queries" an introduction to occlusion queries and proposes some improvements over the naive approach. To overcome the latency introduced by occlusion queries he suggests checking the query result not until the next frame. His rendering loop consists of rendering all the visible objects of the last frame, and issuing occlusion queries for the bounding boxes of objects that were invisible in the last frame. While the rendering of objects that should be visible lags by one frame, this should not be a problem. Checking if visible objects are no longer visible can be done just every few frames, which reduces the number of outstanding occlusion queries at any given time. Besides being a definite improvement over the naive approach his method also solves the problem of occluders vs. occludees. [Sekulic 2004] However, I suspect that his method still does not yield very good results since it makes no use of any hierarchical properties and therefore still has a very high number of unnecessary queries. On the bright side, this also means that there are no restrictions concerning the spatial data structure in use, so a simple non-hierarchical grid could be used, which is very easy to update in a dynamic scene.

### 4.5.2 Coherent Hierarchical Culling

The paper "Coherent Hierarchical Culling" seeks to minimize the number of issued queries and overcome the CPU stalls introduced by occlusion queries by filling the latency time with other tasks, such as rendering visible scene objects or issuing other, independent occlusion queries. "[They] reuse the results of occlusion queries from the last frame in order to initiate and schedule the queries in the next frame. This is done by processing nodes of a spatial hierarchy in a front-to-back order and interleaving occlusion queries with rendering of certain previously visible nodes. The proposed scheduling of the queries makes use of spatial and temporal coherence of visibility." [Bittner et al. 2004] Their method can be easily implemented, has promising test-results, several real-world implementations are already available and it can be used with any hierarchical data-structure.

### 4.5.3 Near Optimal Hierarchical Culling

"The main idea [of Near Optimal Hierarchical Culling] is to use a statistical model describing the occlusion probability for each

occlusion query in order to reduce the number of wasted queries which are the reason for the reduction in rendering speed. We also describe an abstract parameterized model for the graphics hardware performance. The parameters are easily measurable at startup and thus the model can be adapted to the graphics hardware in use. Combining this model with the estimated occlusion probability our method is able to achieve a near optimal scheduling of the occlusion queries”[Guthe et al. 2006] Like Coherent Hierarchical Culling, their method can be easily implemented in existing real-time rendering packages, and it can be used with any hierarchical data-structure. Their test results even suggest that their method is constantly superior to Coherent Hierarchical Culling. They also claim that: ”We have experimentally verified that it is superior to state-of-the-art techniques under various test conditions. Even in low depth complexity situations, where previous approaches could introduce a significant overhead compared to view frustum culling, the presented method performs at least as good as view frustum culling. This means that the introduced method removes the main obstacle for the general use of hardware occlusion queries.” [Guthe et al. 2006]

#### 4.5.4 Other occlusion query methods

Hillesland et al. were among the first to explore efficient use of occlusion queries, in ”Fast and Simple Occlusion Culling using Hardware-Based Depth Queries” [Hillesland et al. 2002].

Staneker et al. propose ”Improving Occlusion Query Efficiency with Occupancy Maps” [Staneker et al. 2003] and later build on this work in ”Occlusion-Driven Scene Sorting for Efficient Culling” [Staneker et al. 2006].

Guthe et al. were not the first to investigate statistical optimization of occlusion queries, Kovalcik et al. described ”Occlusion Culling with Statistically Optimized Occlusion Queries” earlier [Kovalcik and Sochor 2005].

#### 4.6 Other occlusion culling methods

There are several other methods that can be used for occlusion culling and may be worth investigating.

##### Image space methods:

- **Directional Discretized Occluders:**  
”The directional discretized occluders (DDOs) approach is similar to the HZB and HOM methods in that it also uses both object and image-space hierarchies. Bernardini et al. introduce a method to generate efficient occluders for a given viewpoint . These occluders are then used to recursively cull octree nodes during rendering, similarly to HZB and HOM.” [Cohen-Or et al. 2000] Since DDO entirely relies on pre-computed occluders this technique is of no use in our quest for a fully dynamic occlusion culling algorithm.
- **Hierarchical Polygon Tiling with Coverage Masks:**  
”Greene proposed hierarchical polygon tiling algorithm operating on a per polygon basis. It exploits coverage masks to accelerate operations on the z-pyramid. Additionally if a front-to-back order of polygons is established, the z-pyramid can be replaced by a coverage pyramid. The coverage pyramid contains only two state values indicating if the corresponding part of screen is occluded. The depth tests are eliminated completely.” [Bittner 2003] The strict front-to-back traversal is guaranteed by organizing the scene into an octree of BSP-trees.

Timo Aila lists advantages (”elegantly bundles visibility determination and antialiasing to significantly reduce memory requirements of supersampling”), disadvantages (”only works with fully static environments, since dynamic construction of an octree of BSP-trees is computationally very expensive”) and concludes: ”The first drawback is a totally unacceptable requirement and effectively rules out every interactive environment. [...] Our conclusion is that while being a very elegant approach, Hierarchical Polygon Tiling with Coverage Masks has mostly theoretical value.” [Aila 2000].

- **Lazy occlusion grid:**  
The lazy occlusion grid is a conservative image-based occlusion culling method based on a low-resolution grid upon a conventional z-buffer. ”This grid is updated in a lazy manner which reduces the number of expensive occlusion queries at pixel-level significantly compared to a busy update. It allows fast decisions if an object is occluded or potentially visible. The grid is used together with a bounding volume hierarchy that is traversed in a front to back order and which allows to cull large parts of the scene at once.” [Hey et al. 2001] The method works without the need for any preprocessing, and so could be well suited for dynamic scenes.
- **OpenGL-Assisted Occlusion Culling using a Virtual Occlusion Buffer** [Cohen-Or et al. 2000]
- **Approximate Volumetric Visibility** [Cohen-Or et al. 2000]
- **Occluder Shadow Footprints** [Cohen-Or et al. 2000]
- **Image Space BSP Trees** [Hadwiger and Varga 1997]
- **Occlusion Culling Using Silhouettes of Meshes** [Aila 2000] [Aila 2005]

##### Object space methods:

- **Aspect Graph:**  
”The aspect graph [...] partitions the view space into cells that group view points from which the projection of the scene is qualitatively equivalent. The aspect graph is a graph describing the view of the scene (aspect) for each cell of the partitioning. The major drawback of this approach is that for polygonal scenes with n polygons there can be  $\binom{n}{2}$  cells in the partitioning for an unrestricted view space.” [Bittner 2003] The idea is improved upon with the 3D Visibility Complex, Visibility Skeleton and Convex Aspect Graph Variants. However, all of them remain unpractical [Aila 2000]. What sets apart the ideas mentioned in this paragraph from all the other ones covered in this paper, is that they are global visibility algorithms [Bittner and Wonka 2003].
- **Culling using Shadow Frusta:**  
”Hudson and Manocha first select all convex objects in a scene. For each one of them the solid angle is measured. If the solid angle exceeds a fixed threshold, the object is accepted as an occluder. At navigation-time they first build shadow frusta from some of the best occluders, and hierarchically cull axis-aligned bounding boxes of the scene hierarchy using the frusta.” [Aila 2000] Since the algorithm works only with convex occluders it is impractical in the general case, additionally their implementation relies on a pre-computed occluder selection, preventing dynamic scene updates [Cohen-Or et al. 2000].
- **BSP Tree Culling & Occlusion Trees:**  
”The method described [Culling using Shadow Frusta] can be improved using BSP trees. Bittner et al. combine the shadow frusta of the occluders into an occlusion tree. [...] The tree

starts as a single lit (visible) leaf and occluders are inserted, in turn, into it. If an occluder reaches a lit leaf, then it augments the tree with its shadow frustum; if it reaches a shadowed (invisible) leaf, then it is just ignored since it means it already lies in an occluded region. Once the tree is built, the scene hierarchy can be compared with it. The cube representing the top of the scene hierarchy is inserted into the tree. If it is found to be fully visible or fully occluded, then we stop and act appropriately; otherwise, its children are compared with the occlusion tree recursively. This method has an advantage over [Culling using Shadow Frusta] since, instead of comparing the scene with each of the shadow frusta, it is compared with one tree [...], while taking in into account occluder fusion.” [Cohen-Or et al. 2000] However, since this method is based on the static BSP structure it is not suited for dynamic scenes.

- **Large Convex Occluders:** [Cohen-Or et al. 2000].

## 5 Conclusion

As we have seen the problem of handling dynamic scenes and doing dynamic occlusion culling is not an easy one, but recent development in structures & algorithms, faster CPUs and GPUs and hardware support for occlusion culling may finally add up to an end for largely static scenes. To enable dynamic scenes a dynamic data structure has to be chosen, as well as a compatible dynamic occlusion culling algorithm. In the field of hierarchical data structures, the dynamic versions of kd-trees and octrees, like the Bkd-tree, loose octree or dynamic irregular octree are certainly worth a look. Dynamic AABB Trees or simple non-hierarchical grids may also be of interest. Occlusion queries certainly look like the new universal method of choice for occlusion culling, due to the widespread hardware support today and the advances in algorithms that use them, like Near Optimal Hierarchical Culling.

## 6 Acknowledgments

The author would like to thank his mentor Oliver Mattausch for his support.

## References

AILA, T., AND MIETTINEN, V. 2004. dpvs: An occlusion culling system for massive dynamic environments. *IEEE Computer Graphics and Applications* 24, 2, 86–97.

AILA, T. 2000. *SurRender Umbra: A Visibility Determination Framework for Dynamic Environments*. Master’s thesis, Helsinki University of Technology.

AILA, T. 2005. *Efficient Algorithms for Occlusion Culling and Shadows*. PhD thesis, Helsinki University of Technology.

BATAGELO, H., AND WU, S., 2001. Dynamic scene occlusion culling using a regular grid.

BITTNER, J., AND WONKA, P., 2003. Visibility in computer graphics.

BITTNER, J., HAVRAN, V., AND SLAVÍK, P. 1998. Hierarchical visibility culling with occlusion trees. In *Proceedings of Computer Graphics International ’98 (CGI’98)*, IEEE, IEEE, 207–219.

BITTNER, J., WIMMER, M., PIRINGER, H., AND PURGATHOFER, W. 2004. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum* 23, 3 (Sept.), 615–624.

BITTNER, J. 2003. *Hierarchical Techniques for Visibility Computations*. PhD thesis, Department of Computer Science and Engineering. Czech Technical University in Prague.

COHEN-OR, D., CHRYSANTHOU, Y., AND SILVA, C., 2000. A survey of visibility for walkthrough applications.

DENIS HAUMONT, O. M., AND NIRENSTEIN, S. 2005. A low dimensional framework for exact polygon-to-polygon occlusion queries. In *Rendering Techniques 2005: Proceedings of the 16th symposium on Rendering*, Eurographics Association, Eurographics Association, 211–222.

GUTHE, M., BALÁZS, A., AND KLEIN, R. 2006. Near optimal hierarchical culling: Performance driven use of hardware occlusion queries. In *Eurographics Symposium on Rendering 2006*, The Eurographics Association, T. Akenine-Möller and W. Heidrich, Eds., The Eurographics Association.

HADWIGER, M., AND VARGA, A., 1997. Visibility culling.

HAVRAN, V. 2000. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

HELIN, V., 2003. Hierarchies for occlusion culling.

HEY, H., AND PURGATHOFER, W. 2001. Occlusion culling methods. In *Proceedings of EUROGRAPHICS 2001*, Eurographics Association, Eurographics Association.

HEY, H., TOBLER, R. F., AND PURGATHOFER, W. 2001. Real-time occlusion culling with a lazy occlusion grid. Tech. Rep. TR-186-2-01-02, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Jan. human contact: technical-report@cg.tuwien.ac.at.

HILLESLAND, K., SALOMON, B., LASTRA, A., AND MANOCHA, D., 2002. Fast and simple occlusion culling using hardware-based depth queries.

HO, P. C., AND WANG, W. 1999. Occlusion culling using minimum occluder set and opacity map. In *Information Visualization, 1999. Proceedings. 1999 IEEE International Conference on*, IEEE, IEEE, 292–300.

HUA, W., BAO, H., PENG, Q., AND FORREST, A. R. 2002. The global occlusion map: a new occlusion culling approach. In *VRST ’02: Proceedings of the ACM symposium on Virtual reality software and technology*, ACM Press, New York, NY, USA, ACM Press, 155–162.

KOLTUN, V., CHRYSANTHOU, Y., AND COHEN-OR, D., 2000. Virtual occluders: An efficient intermediate pvs representation.

KORTENJAN, M., 2001. Ifc Vortrag von michael kortenjan zum thema potentially visible sets. [Online; accessed 3-January-2007].

KOVALCIK, V., AND SOCHOR, J. 2005. Occlusion culling with statistically optimized occlusion queries. In *WSCG (Short Papers)*, 109–112.

- LAINE, S. 2005. A general algorithm for output-sensitive visibility preprocessing. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, ACM Press, New York, NY, USA, ACM Press, 31–40.
- LUEBKE, D., AND GEORGES, C. 1995. Portals and mirrors: simple, fast evaluation of potentially visible sets. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics*, ACM Press, New York, NY, USA, ACM Press, 105–ff.
- MEISSNER, M., BARTZ, D., HHTNER, T., MLLER, G., AND EINIGHAMMER, J. 1999. Generation of subdivision hierarchies for efficient occlusion culling of large polygonal models. Tech. Rep. WSI-99-13, Department of Computer Science, University of Tbingen.
- NIRENSTEIN, S., AND BLAKE, E. 2004. Hardware accelerated aggressive visibility preprocessing using adaptive sampling. In *Rendering Techniquies 2004: Proceedings of the 15th symposium on Rendering*, Eurographics Association, Eurographics Association, 207–216.
- PFEIFFER, J., SHAGAM, J., AND JR., 2003. Dynamic irregular octrees.
- PROCOPIUC, O., AGARWAL, P., ARGE, L., AND VITTER, J., 2002. Bkd-tree: A dynamic scalable kd-tree.
- RANTA-ESKOLA, S. 2001. *Binary Space Partioning Trees and Polygon Removal in Real Time 3D Rendering*. Master's thesis, Uppsala University, Computing Science Department.
- SAMET, H. 1990. *Applications of spatial data structures: Computer graphics, image processing, and GIS*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- SAMET, H. 1990. *The design and analysis of spatial data structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- SEKULIC, D. 2004. Efficient occlusion culling. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education.
- SHAGAM, J. 2003. *Dynamic spatial partitioning for real-time visibility determination*. Master's thesis, New Mexico State University, Department of computer science.
- STANEKER, D., BARTZ, D., AND MEISSNER, M. 2003. Improving occlusion query efficiency with occupancy maps. In *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, IEEE Computer Society, Washington, DC, USA, IEEE Computer Society, 15.
- STANEKER, D., BARTZ, D., AND STRASSER, W. 2006. Occlusion-driven scene sorting for efficient culling. In *Afrigraph '06: Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, ACM Press, New York, NY, USA, ACM Press, 99–106.
- STOLTE, N., 2007. Nilo stolte home page - octree. [Online; accessed 3-January-2007].
- SUDARSKY, O., AND GOTSMAN, C. 1999. Dynamic scene occlusion culling. *IEEE Transactions on Visualization and Computer Graphics* 5, 1 (1), 13–29.
- TYBERGHEIN, J. 1998. *Crystal Space Online Manual: 4.9.9 Dynamic Worlds*.
- WIKIPEDIA, 2006. Bounding volume — wikipedia, the free encyclopedia. [Online; accessed 3-January-2007].
- WIKIPEDIA, 2006. Hidden surface determination — wikipedia, the free encyclopedia. [Online; accessed 31-December-2006].
- WIKIPEDIA, 2006. Kd-tree — wikipedia, the free encyclopedia. [Online; accessed 3-January-2007].
- WIKIPEDIA, 2006. Scene graph — wikipedia, the free encyclopedia. [Online; accessed 3-January-2007].
- ZHANG, H., MANOCHA, D., HUDSON, T., AND HOFF III, K. E. 1997. Visibility culling using hierarchical occlusion maps. *Computer Graphics 31*, Annual Conference Series, 77–88.