IPT & EGVE Workshop (2005)
R. Blach, E. Kjems (Editors)

# Dynamic Bounding Volume Hierarchies for Occlusion Culling

Vít Kovalčík and Petr Tobola

Faculty of Informatics, Masaryk University Brno, Czech Republic

**Abstract**

*We present an algorithm for rendering complex scenes using occlusion queries to resolve visibility. To organize objects in the scene, the algorithm uses a ternary tree which is dynamically modified according to the current view and positions of the objects in the scene. Aside from using heuristic techniques to estimate unnecessary queries, the algorithm uses several new features to estimate the set of visible objects more precisely while still retaining the conservativeness. The algorithm is suitable for both static and dynamic scenes with huge number of moving objects.*

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Hidden surface removal

## 1. Introduction

The algorithm described in this paper provides an occlusion culling using an occlusion queries. The occlusion query is a hardware accelerated function present in the newer graphic cards.

The common use of the function is as follows: Before rendering an object, its bounding box is calculated and this bounding box is queried for visibility. Only if the bounding box is visible, the object will be rendered.

The function itself is simple. Once the function is activated, the programmer can send several triangles to the graphic card. The graphic card processes the triangles and returns the number of pixels, that passed the z-buffer test (i.e. which would be visible, if this was normal rendering).

However, while the principle is very simple, it is not trivial to use it correctly to gain significant performance boost. The main difficulty is a delay between issuing a query and receiving the result. A possible solution to this problem is to start several queries simultaneously (See figure 1). This approach can reduce the waiting time because the algorithm does not need to wait for the old query results before starting a new occlusion query.

Another effect, which has to be taken into account, is that issuing a query costs some CPU and GPU time, so as few as possible queries should be used. On the other hand, sometimes it is advantageous to use more queries to detect more invisible objects, in order to save the time rendering them.

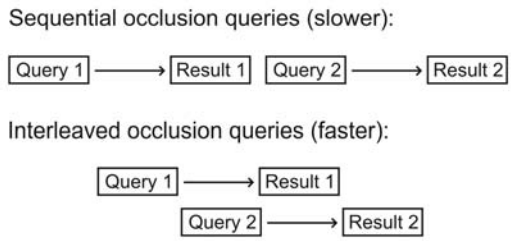


**Figure 1:** *Illustration of two styles of using the occlusion query.*

## 2. Related work

### 2.1. Related work on occlusion culling

Many occlusion culling algorithms were introduced in the previous years. One of the best known is called Hierarchical occlusion maps [ZMHH97]. It's principle is similar to using occlusion queries, although it is purely software based approach: When rendering a frame, several suitable objects are selected as occluders (usually those, which are big and/or near the viewpoint). The occluders are rendered in the normal way. Then the hierarchical occlusion map is created and remaining objects are tested against it and rendered if no occlusion can be found.

After the occlusion query function appeared in the graphic

cards, several occlusion culling algorithms exploiting this function have been developed.

One of the first contributions is described in [HSLM02]. This algorithm requires the scene to be divided into a grid. Each cell of the grid contains a list of objects that intersect the cell. When rendering a frame the grid is traversed in front-to-back order. The conservative visibility of each cell is evaluated by issuing the occlusion query for the cell's bounding box. The cell objects are rendered only if the bounding box is visible.

A different approach can be found in [HTP01]. Contrary to the previous method, this algorithm works in screen-space. The screen is divided into a low-resolution grid and each cell is assigned a variable storing its state. A state can have one of the following values: *occluded*, *unoccluded* or *unknown*. The scene objects are processed in approximate front-to-back order. Each object is projected onto the screen and the state of intersected cells is determined. If an *unknown* cell was intersected, then the occlusion queries are issued to get the actual state. If all of the cells are *occluded*, then the object is rejected. Otherwise the algorithm renders the object and the state of intersecting cells is changed to *unknown*. The algorithm is called "Lazy occlusion grid" because the occlusion queries are not issued immediately after rendering the object, but only when the state of the *unknown* cell is to be tested.

There is another group of algorithms called *approximative*. These algorithms are able to render the scene quickly at the cost of omitting smaller objects, which could be seen by the user, but their contribution to the final image is relatively small. An example of such algorithm can be found in [CKS02]. The scene is divided using a grid. The algorithm maintains a priority queue of cells that are going to be rendered. At the beginning of the rendering of the frame, only the user's nearest cell is in the queue. In each step, the first cell in the queue is retrieved and tested for visibility. If the cell is visible it is rendered and its neighbours are inserted into the queue according to their priority. Then the algorithm continues in the next step. The whole scene can be rendered perfectly in this way. However, it is possible to specify *budget* (time, number of polygons etc.) and stop the algorithm immediately after the budget is reached. The algorithm is not conservative because some low-priority objects might not be rendered. Nevertheless, we can assume that these objects make only insignificant contribution to the final scene image.

### 2.2. Related work on scene organization

In this paper, we focus on large dynamic scenes as well as on the static ones. We will describe some techniques and improvements suitable to display scenes containing huge number of moving objects.

In the recent years, the majority of research effort has been devoted to design efficient data structures for static scenes. This approach can result in highly optimized data structures allowing us to compute visibility quickly and very precisely. Although such data structures can be very efficient, they are usually substantially inflexible at the same time.

If a scene is modified, we have to update the data structure. The time required for the update can be considerably higher than the time saved by the used data structure and hence such data structures are not suitable for processing of dynamic scenes.

An example of efficient data structures for static visibility culling can be *Bounding Volume Hierarchies* (BVH) of scene objects. The BVH consists of a tree structure of bounding volumes of the scene objects. These bounding volumes can be used as occlusion query objects. Because the object for the occlusion query should be as simple as possible, the bounding boxes are usually used.

Unfortunately, the necessity of updating the BVH after every object position change leads to performance reduction. Hence we designed a two-level tree data structure, which allows us to select suitable occlusion query objects as well as quick structure modifications and updates.

## 3. Algorithm

Our algorithm provides conservative occlusion culling using heuristics and dynamic optimizations for the actual scene state. The basis of our algorithm is described in [KS05] (which is also similar to [BWPP04]). A tree structure is used to organize objects in the scene. The tree nodes are hierarchically traversed during the rendering and the occlusion queries are used for testing visibility of nodes. To reduce the number of occlusion queries, the algorithms use heuristics to estimate unnecessary queries. For example, a node visible in a few recent frames will probably be visible again in this frame. Hence we can skip the occlusion query and render it immediately. Also, the algorithm is able to issue several queries simultaneously and do some other work on CPU, while the GPU is processing the queries. (For the full details please see [KS05]).

The following sections describe improvements over the previous algorithm.

### 3.1. Ternary tree

The older algorithm uses axis aligned BSP tree to organize objects in the scene. The root node is split recursively and each object is assigned to the smallest node that is fully surrounding it. It is very simple to add objects into this structure and also move objects from node to node after the object has changed its position in the scene. However, it has also some drawbacks. The most problematic situation is when the root node is split and some objects are intersecting the splitting plane. Such objects are assigned to the root node,

which means they will be rendered every time the root node is found visible, which is virtually in every frame. The same problem arises also for nodes on the lower level.

To remedy this inconvenience, we are using ternary tree. Every node can have zero or exactly three children. Two of the children have the same role as described above. The additional child contain objects, which was previously in the parent node (i.e. objects intersecting the splitting plane.) We don't split this node any further. Now we form a bounding box for group of objects intersecting the cutting plane, issue a query for the visibility of this bounding box and perhaps do not render such objects if the bounding box is not visible. This was not possible using the binary tree.

Unfortunately, using a ternary tree has also a disadvantage - managing the tree, generation of a bounding boxes and issuing another query costs some CPU and GPU time. On the other hand, this is outweighed by the fact that we are rendering less objects then before, which results in speed up in most cases.

In our tests, the time required for managing the tree varied from 0.2 % of the total frame time to nearly 30 % (in a case, when huge number of objects crosses nodes' boundaries very frequently).

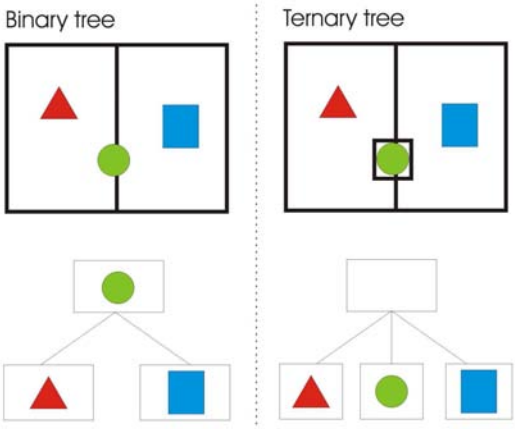


**Figure 2:** *Newly introduced node in a ternary tree holds objects, which intersect the splitting plane dividing the remaining two nodes.*

### 3.2. Shrinked bounding boxes

Another problem of the previous algorithm was that the bounding box of the whole node was tested for visibility, even when there was only one object substantially smaller then the node itself. This means that probability of the visibility of the node's bounding box was much higher than the probability of visibility of the object, so the object was rendered unnecessarily in many cases.

To avoid this unnecessary rendering, we are using a concept of TightBoxes. Under certain conditions we shrink bounding box so that it fits the objects in the node perfectly. Because the calculation of the TightBox may be expensive, we calculate the TightBoxes only for nodes containing objects, which haven't moved for a few frames. The TightBoxes are then reused in the next frame (if there was no change in the node and its children).

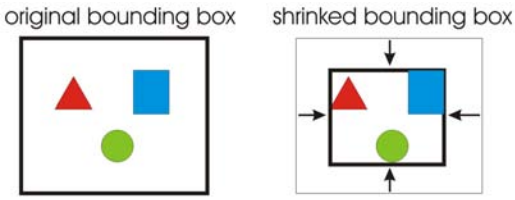


**Figure 3:** *Bounding boxes of the nodes are shrunk to better fit the objects inside.*

### 3.3. Additional query

Behind this feature is a simple idea: Instead of issuing one query for a node we can issue two separate queries. The first query returns the number of visible pixels and the second query returns the number of pixels covered on the screen including the invisible ones. This feature alone does not improve performance of the rendering (actually, it increases the fill-rate, therefore the rendering is going to be slightly slower), but then we have information that can be used to modify the tree and optimize the rendering. The modification is described in the next section.

### 3.4. Dynamic modifications of the tree

Because both the viewer and the objects move through the scene, we slightly update the scene tree every frame to adapt it to the current situation.

The additional query (described in the previous section) can report a ratio of visible pixels to total pixels for any node, which can be taken as a hint when altering the tree. The tree is modified in the following way: All of the tree's sibling leaves are traversed and then split or merged according to the following simple criteria.

- If both of the following conditions are met, the node is split,

$$\frac{Pixels_{visible}}{Pixels_{total}} \geq R_{split}$$

and

$$Objects_{node} \geq O_{split}$$

- If the following conditions hold for all sibling leaves of

one parent, the leaves are discarded and the objects from them are moved to the parent node,

$$\frac{Pixels_{visible}}{Pixels_{total}} \leq R_{merge}$$

and

$$Objects_{node} \leq O_{merge}$$

The constants $R_{split}$, $O_{split}$, $R_{merge}$ and $O_{merge}$ were set empirically to the following values: $R_{split} = 0.25$, $O_{split} = 12$, $R_{merge} = 0.75$ and $O_{merge} = 5$.

In addition to the described conditions, which can tell us *when* to split a node, we can also change the way *how* the node is split. In the common case, the node (which is axis-aligned bounding box) is split in halves by its longest side. In our dynamic splitting algorithm, we prefer a box side with maximal inner product of $(n.v)$ to be parallel with the splitting plane; $n$ means normalized normal vector and $v$ means normalized camera orientation vector.

### 3.5. Detection of colliding queries

Our algorithm takes advantage of the fact, that several occlusion queries can be active simultaneously. In other words, we can solve visibility for several nodes in nearly the same time as for one node. However, sometimes it may happen that we need to check nodes, which occupies the same screen space, so the objects in the nearer node can occlude the distant node. Unfortunately, the objects in the nearer node were not rendered yet and the occlusion query for the distant node may return that it is visible, while the objects in the nearer node will hide it. The better solution could be to send the occlusion query of the distant node after the occlusion query of the nearer node is processed and the nearer node is rendered if the occlusion query returns any visible pixels.

The test for detection whether two queries occupying the same screen space is simple. Both queries are projected onto the screen and their bounding rectangles are tested for intersection.

This strategy leads to the reduction of the number of rendered objects. On the other hand, it may sometimes lead to the unnecessary waiting until the first query is finished. However, in complex scenes the waiting is compensated by the fact that we know more precisely which objects are visible and thus we don't spend time on rendering the invisible ones.

## 4. Results

All the tests were performed on computer with AMD Athlon MP 2600+ processor, 2 GB of RAM and NVIDIA GeForce 6800 GT with 256 MB of memory.

Two scenes were used for testing:

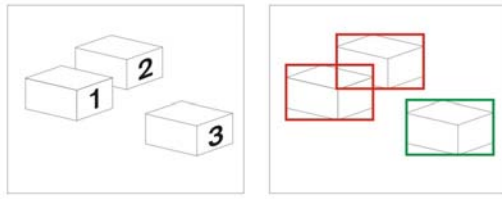


**Figure 4:** *Projected bounding boxes forming rectangles on the screen. The rectangles for bounding boxes number 1 and 2 overlap, so the farther node must wait till the nearer one is processed. Bounding box number 3 can be processed independently in any time.*

- The well-known power plant model with 2500 bunny models, which are moving through the power plant. The movement is simple and no collision detections are made as this would require enormous CPU power.
  The power plant was preprocessed to gain better control over the occlusion of objects: The whole model was cut into pieces by the uniform grid creating 11050 objects with total of over 18 million triangles. (The original power plant has less triangles, but we have split some of them into more parts.) Every bunny has 69451 triangles, which means the bunnies add other 173 million triangles.
  We have tested three variants of this scene: The full power plant with moving bunnies, the power plant with static bunnies and the power plant alone.

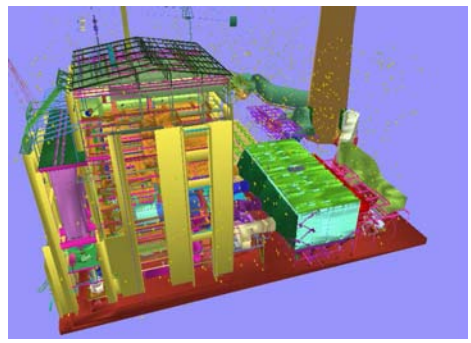

**Figure 5:** *The power plant model. Small spots are the bunny models moving freely around the power plant.*

- Scene with 2500 teapots (16 million triangles). Again, no collision detections are made. Although the number of objects is high, there are a lot of holes, so it is difficult to detect an occlusion.
  Two variants have been tested: Teapots with some basic movement and a static scene.

The results are summarized in table 1. The algorithm works well for huge scenes with a lot of complex objects. It works well even when half of the objects in the scene is moving, however it can also handle static scenes.

| Scene | Time (sec.) | | | # of rendered objects | | | # of occlusion queries | | |
|---|---|---|---|---|---|---|---|---|---|
| | New | Old | Ratio | New | Old | Ratio | New | Old | Ratio |
| Power plant + moving bunnies | 394 | 557 | 0.71 | 408 | 812 | 0.50 | 103 | 27 | 3.82 |
| Power plant + static bunnies | 152 | 422 | 0.36 | 276 | 747 | 0.37 | 92 | 26 | 3.54 |
| Power plant alone | 135 | 249 | 0.54 | 272 | 656 | 0.41 | 91 | 25 | 3.64 |
| Teapots (moving) | 27 | 29 | 0.93 | 2289 | 2474 | 0.93 | 59 | 38 | 1.55 |
| Teapots (static) | 22 | 24 | 0.92 | 1884 | 2135 | 0.88 | 96 | 47 | 2.04 |

**Table 1:** *Comparison of the introduced (new) and the basic (old) algorithm. "Time" column contains information about how long it takes to fly through the particular scene using both algorithms. Remaining columns show the average number of rendered objects and occlusion queries per frame.*

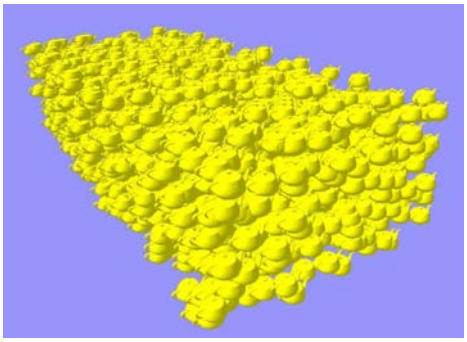

**Figure 6:** *The teapots scene.*

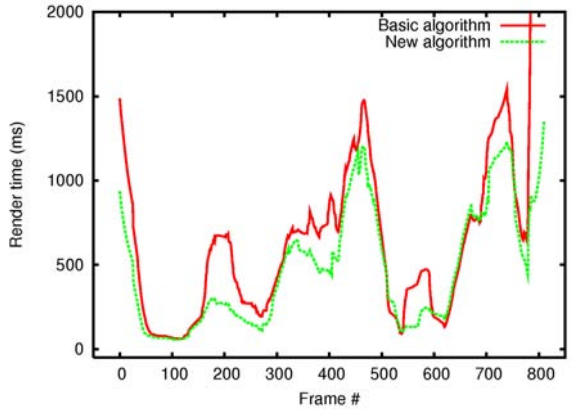


**Figure 7:** *Comparison of rendering speeds of the basic and the new algorithm.*

An interesting point to note is that even when the new algorithm uses more occlusion queries, it is faster than the old one. The reason is simple: There are also more objects, which are found to be invisible and the algorithm saves time by not rendering them.

The Teapots scene caused some problems, because there was only a small occlusion that was hard to detect. Therefore the featured algorithm draws every frame nearly 2300 objects, on average, out of 2500 present in the scene. (However, it is still better than the older algorithm with an average of 2470 rendered objects.)

We have included two graphs showing detailed progress of the first test - fly through a scene with power plant and 2500 moving bunnies. You may see the time of rendering of each frame (figure 7) and also the number of rendered objects (figure 8).

In the graphs and in the table you may notice that although the older algorithm renders twice as many objects as the new algorithm, it is not twice as slow (but only about 1.4 times slower). The reason is that the newer algorithm has to recalculate auxiliary data every frame, because of massive movement in the scene. This is not necessary in the static scenes, where the difference between the algorithms is much greater.

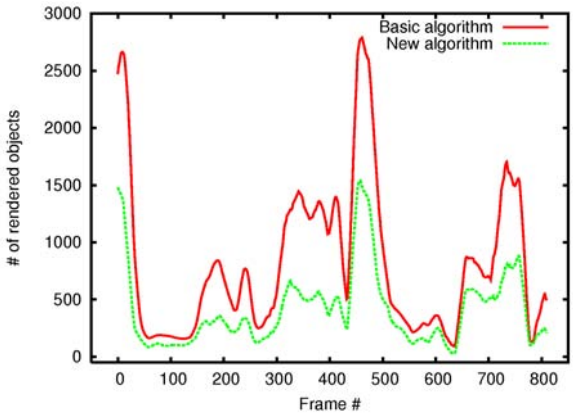


**Figure 8:** *Comparison of number of rendered objects in basic and in the new algorithm.*

## 5. Conclusion

We have presented an occlusion culling algorithm based on our previous one, but with many improvements. The new version of the algorithm is still conservative, but can reduce

number of rendered objects considerably. It is especially useful on huge scenes and also natively supports scenes with vast amount of moving objects.

In the current version, the algorithm uses several constants, which were set empirically. As a future work, we would like to calculate these parameters automatically, probably depending on the number of objects in the scene and distances between them.

**References**

[BWPP04] BITTNER J., WIMMER M., PIRINGER H., PURGATHOFER W.: *Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful*. Tech. rep., 2004.

[CKS02] CORRÊA W. T., KLOSOWSKI J. T., SILVA C. T.: Fast and simple occlusion culling. *Game Programming Gems 3, Charles River Media* (2002).

[HSLM02] HILLESLAND K., SALOMON B., LASTRA A., MANOCHA D.: *Fast and simple occlusion culling using hardware-based depth queries*. Tech. Rep. TR02-039, Department of Computer Science, University of North Carolina, 2002.

[HTP01] HEY H., TOBLER R. F., PURGATHOFER W.: *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, January 2001.

[KS05] KOVALČÍK V., SOCHOR J.: Occlusion culling with statistically optimized occlusion queries. In *WSCG 2005 short papers proceedings* (2005), Skala V., (Ed.), pp. 109–112.

[ZMHH97] ZHANG H., MANOCHA D., HUDSON T., HOFF III K. E.: Visibility culling using hierarchical occlusion maps. *Computer Graphics 31*, Annual Conference Series (1997).