Occlusion Culling with Statistically Optimized Occlusion Queries

Vít Kovalčík
Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno,
Czech Republic
xkovalc@fi.muni.cz

Jiří Sochor
Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno,
Czech Republic
sochor@fi.muni.cz

ABSTRACT

This paper presents an effective algorithm for occlusion culling using hardware occlusion queries. The number of queries is optimized according to the results of the queries from several preceding frames. Parts of the scene which are found to be unoccluded in recent frames, are tested less often thus reducing the number of queries performed per frame. The algorithm is applicable to any kind of scene, including scenes with moving objects. The algorithm utilizes a tree structure containing objects in the scene.

Keywords

Visibility, real-time rendering, occlusion culling, occlusion query

1. INTRODUCTION

The number of details in virtual environments is still increasing and requires the use of "clever" algorithms for displaying a scene. Simple brute force approaches to rendering complex scenes, do not achieve interactive frame rates. Therefore algorithms performing occlusion culling have to be used. Such algorithms are able to detect objects, which are occluded by another object(s) from a user's point of view, and quickly discard these hidden objects from further processing.

There are many methods for performing occlusion culling (for more details see survey [Coh03a]). In recent years hardware based occlusion queries have started to be used. The query allows the programmer to indirectly access the Z-buffer and test if an object is visible or if it is shielded by already rendered objects. The tested area is usually just a bounding box of a fully detailed object. Based on the results of a query the application can decide whether or not to render a full object.

WSCG 2005 SHORT papers proceedings,
ISBN 80-903100-9-5
WSCG'2005, January 31-February 4, 2005
Plzen, Czech Republic.

Despite the simplicity of the occlusion query function, it is not trivial to use it correctly to gain a significant performance boost. Several algorithms for using occlusion queries have been developed.

One of the first was [Hil02a]. The scene is divided into a grid and each cell in the grid contains list of objects that are intersecting it. When rendering a frame, the grid is processed by layers in front to back order. For each cell the visibility of its bounding box is queried and in cases where the box is visible, objects in the cell's list are rendered, unless they had not been previously rendered because they intersected another already processed cell.

Another approach was described in [Hey01a]. In contrast with the previous method, this algorithm works in screen-space. The screen is divided into a regular low-resolution grid in which each cell remembers whether the relevant part of the screen is occluded. When rendering, objects in the scene are processed in front-to-back order and each object is tested against the occlusion grid. Because the occlusion state of the cells in the grid is updated only when it is necessary, this method is called "Lazy occlusion culling" (for more details see [Hey01a]).

Our method uses a scheme similar to one recently described in [Bit04a]. The whole scene is organized in a tree structure. During rendering the nodes are traversed and their visibility is tested using occlusion queries on their bounding boxes. The contents of a

node are rendered only where such a node is found visible.

In order to reduce the number of queries, we are using heuristics to detect nodes, which are probably visible and the algorithm renders such nodes without issuing a query.

2. OCCLUSION QUERY

Occlusion query is a hardware function present in modern graphic cards. The principle is simple: After a part of the scene is rendered onto screen and to the Z-buffer, there is a complex object to be rendered. Instead of rendering it, the displaying algorithm may choose to test whether the object is actually visible. This test is performed by retrieving the bounding box of the object and applying the occlusion query on it. The occlusion query returns the number of pixels that would have been visible, if the box had been rendered. This is done by comparing the box with stored Z-buffer values. If the number of possibly visible pixels is equal to zero, the bounding box is hidden by previously rendered object(s) and it is not necessary to render the complex object.

Unfortunately, the use of the occlusion query function is not that simple. Due to the buffering of data sent to a graphic card it often happens that previous parts of the scene have not been rendered at the time when a query is issued. However, occlusion queries might be processed asynchronously. It is possible to start a query, then render some object and use the result of the query later when it is available. Furthermore, it is not necessary to wait with the next query until the previous one is finished – the queries may run simultaneously. The processes are illustrated in Figure 1:

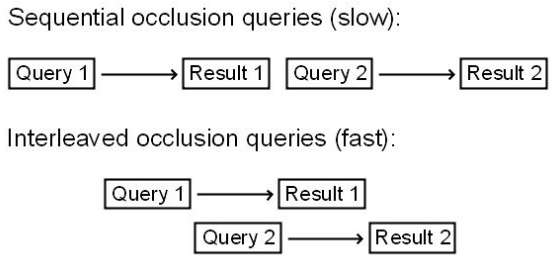

Figure 1: Illustration of simple and advanced use of the occlusion queries

3. ALGORITHM

Overview

Our algorithm requires the scene to be organized in a hierarchical tree structure. In our experiments we utilize an axis-aligned BSP tree, but octree, kD-tree or other similar structures could also be used. Each object in the scene is placed in exactly one node, that encom-passes the object fully and as tightly as possible.

When rendering a frame, the algorithm sets up a queue which holds nodes to be processed. Initially it contains the root node only. The queue is processed in natural order and for each node the algorithm decides whether the objects in a node will be discarded, queried for visibility or rendered without using an occlusion query.

The first case is straightforward. If the node is discarded, for example because of frustum culling, it is removed from the queue and the algorithm moves to the next node in the queue.

The second case is slightly more complicated. For some nodes, the algorithm may decide that an occlusion query is not necessary (the decision process will be described later). Objects which are stored in such a node are immediately rendered and the node is removed from the queue. Its descendants are placed in the queue at the position of the deleted node. The newly inserted nodes are sorted in front-to-back order. The algorithm then continues with the first descendant.

The third case is the most complex. If there is not enough information about the results of recent occlusion queries, it is difficult to predict, whether objects in a node should be rendered or not. At this point, the query is issued. The result of the query will be available after some time. It is possible to wait for the query to finish, but it would be a waste of time that could be used for processing another node. Hence the algorithm starts to process the next node in the queue instead of waiting for the result.

When a query finishes, depending on the result the node may be either skipped or the objects in the node are rendered and the node in the queue is replaced by its descendants. Because newly inserted nodes precede the currently processed node, the algorithm has to sometimes return and pass through the queue again. It can stop processing the queue at any time and return to the beginning of the queue, usually after the number of queries exceeds some threshold (about 20) and there is high probability that the first queries are already finished. It would be possible to stop processing the queue and return to the exact time when the result of the first query is available, but that would require additional checking of the status of the query, which in itself is time consuming.

The actual implementation uses two queues – one is the main query described above and the other is the queue with nodes with the occlusion query issued and not finished yet. Here is the overview in pseudocode:

```
queue.insert (root);
while (!queue.empty) {
  while (!queue.empty &&
  !query_queue.FirstNode.AnswerReady) {

    act_node = queue.FirstNode;
    action = CalcNodeAction (act_node);
```

```
    if (QUERY == action) {
      act_node.IssueQuery;
      query_queue.Add (act_node);
    } else if (RENDER == action) {
      RenderNode (act_node);
    }
    query.DeleteFirstNode ();
  }

  queue.SetPointerToStart;
  while (!query_queue.empty) {
    visible_pixels =
      query_queue.FirstNode.GetResult;
    SaveStatistics
      (query_queue.FirstNode,
      visible_pixels);
    if (visible_pixels > 0) {
      RenderNode
        (query_queue.FirstNode);
      queue.AddChildrenBeforePointer
        (query_queue.FirstNode);
    }
    query_queue.DeleteFirstNode;
  }
}
```

The *CalcNodeAction* function is crucial for the algorithm. It takes a node as a parameter and returns the value, which informs the rest of the algorithm, what actions should be taken for the given node. The actions are:

- *RENDER*. Objects in the node will be rendered without issuing a query.
- *SKIP*. The node is invisible, the objects in the node will not be rendered.
- *QUERY*. Occlusion query will be issued to determine if the node is visible or not.

Here is a pseudocode for a simple version of the *CalcNodeAction* function. This version does not utilize any results of the preceding occlusion queries.

```
if (FrustumCulled (node))
  return SKIP;
else if (ViewerIsInside (node))
  return RENDER;
else
  return QUERY;
```

Optimizations

The *CalcNodeFunction* can make an estimation (based on the results of recent occlusion queries) and change a return value from *QUERY* to *RENDER*. This estimation has to be done carefully, otherwise we could end up with rendering many objects, which are actually occluded. On the other hand, we do not want to use many occlusion queries as it may severely reduce performance.

The algorithm stores the results of several recent occlusion queries for every node and uses them to determine whether to initiate an occlusion query or not. The more times the node was found visible, the less often the query will be issued to check if it is still visible.

Pseudocode for the optimized *CalcNodeAction* function follows:

```
if (FrustumCulled (node))
  return SKIP;
else if (ViewerIsInside (node))
  return RENDER;
else {
  if (StatisticsTooOld (node))
    return QUERY;

  if (OCCLUDED ==
    LastQueryResult (node))
    return QUERY;

  occ_num = GetNumOfUnoccludedResults
    (node);
  not_query_time = last_query_time +
    BASE_TIME * (2 - 0.5^occ_num);

  if (actTime > not_query_time)
    return QUERY;

  return RENDER;
}
```

The BASE_TIME constant depends on the speed of viewer's movement and also on the type of scene. For higher speeds we select a lower number. In our tests, this constant was equal to 2/3 of second, which is a good compromise between forgetting the statistics too early and predicting the occlusion incorrectly because of too old information.

4. RESULTS

All tests were performed on a computer with Intel Pentium 4/2.0 GHz processor, 512 MB of RAM and ATI Radeon 9700 with 128 MB of memory.

Three scenes were tested. The first one (Figure 2) was the power plant model containing nearly 2 million triangles. It is a smaller version of the UNC's power plant model. Unfortunately, it was not possible to render the full model with its 12 million triangles in interactive frame rate with this configuration.

The second scene (Figure 3) was a computer-generated library with shelves containing nearly 40,000 books with a total of over 10 million triangles. The shelves does not have back sides, so the books were the main occluders.

The third scene (Figure 4) consists of 5,000 randomly placed teapots that are made of 32 million triangles.

Three different rendering algorithms were used:

- No occlusion culling. This algorithm uses only view-frustum culling, it does not use any kind of occlusion culling.
- Simple occlusion culling. The algorithm starts with the root of the scene hierarchy and traverses

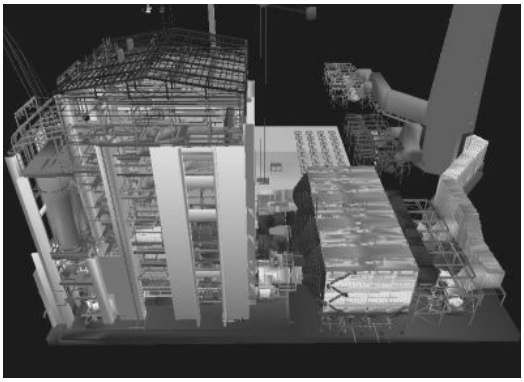

Figure 2: The power plant model (first test scene)

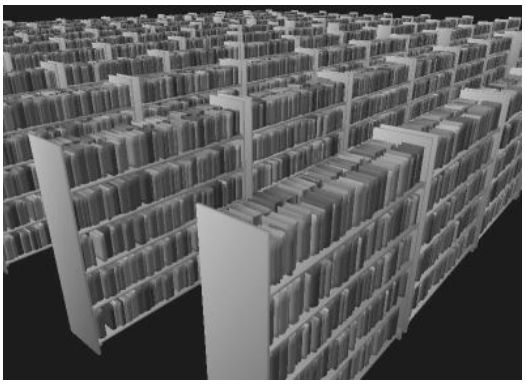

Figure 3: The library (second test scene)

the hierarchical structure in a top-down manner to the leaves. Before rendering a node, the occlusion query is utilized to get the visibility of the node's bounding box. If the box is invisible, the node and it's descendants are not going to be rendered.

– Statistical occlusion culling. This is the algorithm described in the previous section.

For each scene we run these algorithms to render a fly-through containing several hundred frames and we measured the total rendering time. The result are shown in Table 1:

Scene	No OC	Simple OC	Stat. OC
Power plant	43	37	**22**
Library	14	7	**4**
Teapots	33	11	**11**

Table 1. Time (in seconds) to fly through several scenes using three different rendering algorithms.

The occlusion culling algorithm with statistically controlled occlusion queries gives the best results in most cases. However, sometimes it may be slower than "Simple OC" because the statistics of a recent occlusion may give a false prediction and unnecessarily render many objects. But the statistics are used only for a brief interval, so "Stat. OC" is usually only a little slower in these problematical cases.

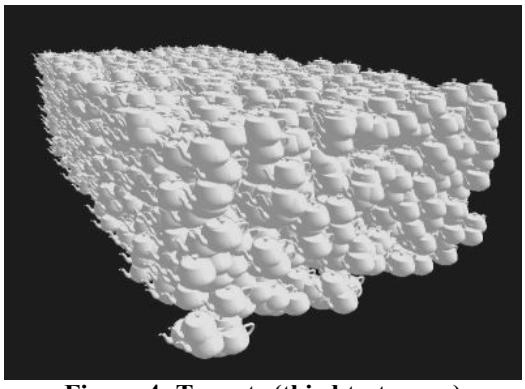

Figure 4: Teapots (third test scene)

5.CONCLUSION AND FUTURE WORK

We have described a new occlusion culling algorithm, which is able to render scenes up to four times faster than algorithms using view-frustum culling only. It can operate on any type of scene, including a scene with moving objects.

There are many directions for future work. The algorithm can be improved by better ordering of the queries, or by improving prediction function based on the recent statistics.

6. ACKNOWLEDGMENTS

This work was supported by grant VZ MSM 143300003.

7. REFERENCES

[Bit04a]Bittner, J., Wimmer, M., Piringer, H., Purgathofer, W. Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. Computer Graphics Forum (Proc. Eurographics 2004) 23(3):615-624, 2004.

[Coh03a] Cohen-Or, D. Chrysanthou, Y., Silva, C.T., Durand, F. A survey of visibility for walkthrough applications. IEEE Transactions on Visualisation and Computer Graphics 9, 3. 2003.

[Cor02a] Corrêa, W.T., Klosowski, J.T., Silva, C. T. Fast and Simple Occlusion Culling. Game Programming Gems 3, Charles River Media, 2002.

[Hey01a] Hey, H., Tobler, R.F., Purgathofer, W. Real-Time Occlusion Culling with a Lazy Occlusion Grid. Technical Report TR-186-2-01-02, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2001.

[Hil02a] Hillesland, K., Salomon, B., Lastra, A., Manocha, D.. Fast and simple occlusion culling using hardware-based depth queries. Technical Report TR02-039, Department of Computer Science, University of North Carolina, 2002.