ABSTRACT

This paper presents an effective algorithm for occlusion culling using hardware occlusion queries. Number of queries is optimized according to the results of the queries from several preceeding frames. Parts of the scene which are found to be unoccluded in the 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 the virtual environments still increases and requires usage of “clever” algorithms for displaying the scene. Simple brute force approach to rendering of complex scenes, does 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 objects 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 the recent years hardware based occlusion queries 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 with already rendered objects. Based on the results of a query the application can decide, whether to render full object or not.

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

One of the first was [Hil02a]. The scene is divided into 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 case the box is visible, objects in the cell's list are rendered (if they were not rendered before because they intersected another already processed cell).

Another approach was described in [Hey01a]. As opposed to the previous method, this algorithm works in screen-space. The screen is divided into regular low-resolution grid in which each cell has its own state. It can be occluded, unoccluded or unknown. When rendering starts, each cell's state is set to unoccluded. Objects in the scene are processed in front-to-back order. Each object is projected on the screen and the state of intersecting cells is tested. If all cells are occluded, the object is not rendered. If there is some unoccluded cell, the object is rendered. If there are some cells with the unknown state, their real state is tested using occlusion queries and the decision is made when the results are available. After rendering an object, the state of intersecting cells is changed to unknown. Because the occlusion queries
are not issued immediately after the object is
rendered, but only when it is really necessary, the
method is called “Lazy occlusion culling”.

Importance based algorithms with quality tradeoff use
occlusion queries to speed up rendering by omitting
objects, which are normally visible, but does not
contribute much to the final image [Cor02a]. The
scene is again divided by a grid. The algorithm
maintains priority queue of the cells and processes it
starting with the cell with the highest priority. Every
time a cell is visited and found visible it is rendered
and its unvisited neighbours are added into the queue.
This way all visible cells could be rendered, but it is
also possible to stop rendering when there is not
enough time to render the full frame. Some objects
will not be rendered, but it will be those with the least
priority and it will not affect final image too much.

Our method uses scheme similar to the 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. In case a node is
visible, its content is rendered, otherwise it is
skipped. To reduce number of queries not all nodes
are tested each frame. Instead, some heuristics is used
to detect nodes, which are probably visible and
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 Z-
buffer, there is some 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 number of pixels that
would have been visible, if the box was 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 usage 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 are not rendered yet when
issuing a query. However, occlusion queries might be
processed asynchronously. It is possible to start a
query, then render some object and use the result of
query later when it is available. Also 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 on the
Figure 1:

Sequential occlusion queries (slow):

```
Query 1 → Result 1 → Query 2 → Result 2
```

Interleaved occlusion queries (fast):

```
Query 1 → Result 1
Query 2 → Result 2
```

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
utilized axis-aligned BSP tree, but octree, kD-tree or
similar structure could also be used. Each object in
the scene is placed in exactly one node, that
encumbers 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 node will be
rendered without using an occlusion query or
occlusion-queried.

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

The second case is slightly more complicated. For
some node, the algorithm may decide that occlusion
query is not necessary (the decision process will be
described later). Objects which are stored in such
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 most complex. If there is not enough
information about results of the recent occlusion
queries, it is difficult to predict, whether objects in a
node should be rendered or not. At this situation 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 waste of time we
could use for processing another nodes. 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 new inserted nodes precede the currently processed node, the algorithm has to sometimes return back and pass again the queue. It can stop processing the queue at any time and return to the beginning of the queue. Usually when 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 back exactly at the time when result of the first query is available, but that would require additional checking of the status of the query, which 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:

```plaintext
queue.insert (root);
while (!queue.empty & !query_queue.empty) {
    queue.SetPointerToStart;
    while (!queue.empty) {
        act_node = queue.GetNodeAtPointer;
        action = CalcNodeAction (act_node);
        if (QUERY == action) {
            act_node.IssueQuery;
            query_queue.Add (act_node);
        } else if (RENDER == action) {
            RenderNode (act_node);
        }
        query.MovePointerToNextNode;
        query.Delete (act_node);
    }
    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};
            queue.SetPointer
                {AfterInsertedNodes};
        }
        query_queue.DeleteFirstNode;
    }
}
```

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 simple version of the CalcNodeAction function. This version does not utilize any results of the preceeding occlusion queries.

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

**Optimizations**

The CalcNodeAction function can make an estimation (based on the results of recent occlusion queries) and change 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 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:

```plaintext
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 = BASE_TIME * (2 - 0.5^occ_num);
}
```

The BASE_TIME constant depends on the speed of viewer's movement in a given application. For higher speeds we select lower number. In our tests, this constant was equal to 2/3 of second.

### 4. RESULTS

All tests were performed on 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 of triangles. It is smaller version of the UNC's power plant model. Unfortunately, it was not possible to render the full model with its 12 million of triangles in interactive frame rate on this configuration.

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

Third scene (Figure 4) consists of 5,000 randomly placed teapots that are made of 32 million of 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 the hierarchical structure in top-down manner to the leaves. Before rendering a node, occlusion query is utilized to get the visibility of the node's bounding box. If the box is invisible, the node and it's children are not going to be rendered.
- Statistical occlusion culling. This is the algorithm described in the previous chapter.

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

<table>
<thead>
<tr>
<th>Scene</th>
<th>No OC</th>
<th>Simple OC</th>
<th>Stat. OC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power plant</td>
<td>43</td>
<td>37</td>
<td>22</td>
</tr>
<tr>
<td>Library</td>
<td>14</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Teapots</td>
<td>33</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

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

Occlusion culling algorithm with statistically controlled occlusion queries gives the best results in most cases. However, sometimes is may be slower than “Simple OC” because statistics of a recent occlusion may give false prediction and unnecessarily many objects are rendered. But the statistics are used only for short time interval, so “Stat. OC” is usually only a bit slower in these problematical cases.

5. CONCLUSION AND FUTURE WORK
We have described new occlusion culling algorithm, which is able to render scenes up to 4 times faster than algorithms using view-frustum culling only. It can operate on any type of the scene, including a scene with moving objects.

There are many possibilities 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

