Near Optimal Hierarchical Culling:
Performance Driven Use of Hardware Occlusion Queries

Michael Guthe[†], Ákos Balázs[‡], and Reinhard Klein[§]

Institute for Computer Science II, Universität Bonn, Germany

Abstract

The most efficient general occlusion culling techniques are based on hardware accelerated occlusion queries. Although in many cases these techniques can considerably improve performance, they may still reduce efficiency compared to simple view frustum culling, especially in the case of low depth complexity. This prevented the broad use of occlusion culling in most commercial applications. In this paper we present a new conservative method to solve this problem, where the main idea 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. The implementation of the algorithm is straightforward and it can be easily integrated in existing real-time rendering packages based on common hierarchical data structures.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Hidden line/surface removal I.3.3 [Computer Graphics]: Display algorithms

1. Introduction

In many graphics applications, such as first-person computer games and architectural walkthroughs, the user navigates through a complex virtual environment. Often the user can only see a relatively small fraction of the scene. Therefore, the goal of culling techniques is to quickly determine a so-called potentially visible set (PVS), which is the visible fraction of all objects or a superset of it. While the relatively simple view frustum culling only removes geometry that is projected outside the viewport, occlusion culling techniques also try to identify geometry that is occluded and therefore does not contribute to the final image.

If scenes with high depth complexity are rendered, removing invisible geometry can significantly increase the rendering performance. On the other hand, if the depth complexity is moderate as in Figure 1 or even lower, most occlusion
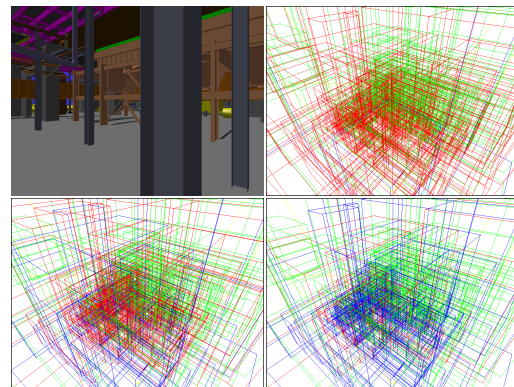

[†] e-mail:guthe@cs.uni-bonn.de
[‡] e-mail:edhellon@cs.uni-bonn.de
[§] e-mail:rk@cs.uni-bonn.de

Figure 1: *Left to right / top down: a) Moderately occluded view of the power plant model; b) bounding boxes of successful (green) and wasted (red) occlusion queries using [BWPP04]; c) our method on a GeForce 5900; and d) on a Radeon 9800. Leaves rendered without queries are blue.*

culling techniques need more time to determine visibility than what can be saved by not rendering occluded geometry.

This problem prevented the broad use of occlusion culling techniques in most consumer applications so far since it often leads to a significant performance loss in such situations. The only exception are precomputed visibility methods but these are restricted to static closed environments and thus their usability is limited. To come up with a solution that takes advantage of occlusion culling in case of high depth complexity and avoids culling in situations with low depth complexity, an adaptable algorithm is needed.

The most common approach for efficient occlusion culling with hardware based occlusion queries is to organize the scene in a spatial hierarchy. For rendering, this hierarchy is traversed in a front to back manner. During the traversal a decision has to be made for each node whether to test for occlusion or not. Current GPUs also allow to perform occlusion queries parallel to the traversal. In this case a second decision is required whether to wait for the result of the query, or to directly continue traversing. To optimize the rendering performance, the algorithm has to adapt to the current depth complexity and graphics hardware for both decisions.

2. Related Work

With the demand for rendering scenes of ever increasing complexity, there have been a number of visibility culling methods developed in the last decade. A comprehensive survey of visibility culling methods was presented by Cohen-Or et al. [COCSD03]. Another recent survey of Bittner and Wonka [BW03] discusses visibility culling in a broader context of other visibility problems. According to the domain of visibility computation, the different methods can be categorized into from-point and from-region visibility algorithms. From-region algorithms (e.g. cells and portals [TS91]) compute a PVS in an offline preprocessing step, while from-point algorithms are applied online for each particular viewpoint (e.g. [GKM93, HMC*97, ZMHH97, KS01]). While a variant of hierarchical z-buffers [GKM93] is used inside recent graphics cards to reduce the internal memory transfer, hardware occlusion queries [NA01] are supported via the graphics API to allow applications to determine how many fragments of a rendering operation pass the z-buffer test. To decide if an object is occluded, the application can render its bounding box with an occlusion query and without frame buffer writing. If the number of fragments returned by the query is zero, the bounding box is not visible and thus the object is occluded.

2.1. Hardware occlusion queries

The main advantage of hardware occlusion queries is their generality and the fact that they do not require any precomputations. However, due to the required read-back of information from the graphics card and the long graphics pipeline, the queries introduce a high latency if the application waits for the result. This latency can be hidden by exploiting the possibility of issuing several occlusion queries in parallel and using their results later. During traversal two decisions have to be made for each node: first, whether to issue a query and second, if a query is issued, to wait for the result before continuing traversal or to traverse the subtree immediately. If a query is issued for each node of the hierarchy and the traversal algorithm waits for the results before traversing the subtree, the method degenerates into a breadth-first traversal, as almost certainly all queries of a level can be issued before the first one is finished and thus many occlusion queries are performed before anything is rendered. In order to avoid such problems Bittner et al. [BWPP04] proposed to use temporal coherence to guide these two decisions. A query is issued either if the node was not visible in the last frame or is a leaf node in order to determine its visibility for the next frame. If a query is issued for a previously invisible node, traversal of the subtree is delayed until the result is available. Previously visible leaf nodes are immediately rendered. In this scheme, occlusion queries are only performed along a front of termination nodes in the hierarchy consisting of previously invisible inner nodes and visible leaf nodes.

Although a significant speedup was achieved compared to the naïve approach, two major problems were not solved. The first problem is that each occlusion query needs some time, so if too many of them are wasted, the performance is reduced compared to view frustum culling alone. The main source of wasted queries is the fact that all visible leaf nodes need to be queried to determine the visibility state of all nodes in the hierarchy. In addition, many queries that are performed for inner nodes which were frustum culled in the previous frame are wasted as well. Assuming that these nodes are visible is however even worse, since then all of their leaves would be rendered with an additional query for each. The second problem is that the performance gain of occlusions is delayed by one frame, since previously visible leaf nodes are directly rendered and the query results are only used in the next frame. The first problem was already partially addressed by Staneker et al. [SBS04] who proposed a method to save queries for objects that are certainly visible. A variety of software tests, like occupancy maps or software occlusion tests with reduced resolution, are used to determine if the object covers screen areas that are still empty. However, when using current graphics hardware these relatively costly software tests are unfortunately almost always slower than the hardware occlusion query itself.

2.2. Graphics hardware parametrization

To achieve a given constant frame rate in scenes with highly varying complexity, several methods to estimate the rendering time have been proposed starting with the work of Funkhouser and Séquin [FS93]. Considering the fact that an upper bound of the rendering time is required for a constant frame rate, Wimmer and Wonka [WW03] modified this method and also adapted it to the characteristics of current

graphics hardware. Although their method is very efficient in achieving a constant frame rate, it cannot be used to guide the decision of whether rendering with occlusion culling is better than rendering without occlusion culling, as in this case an approximate time is required rather than an upper bound. In addition, neither of the above mentioned two algorithms is able to give time estimations for hardware occlusion queries which are different from those for the rendering of the bounding box.

3. Analytical Models

To achieve a near optimal scheduling of occlusion queries, both the outcome of each query and the times required for rendering and for performing the query itself must be estimated. This estimation is based on two analytical models which we derive in the following using the definitions in Table 1 and 2 respectively.

3.1. Occlusion probability

For a set of objects $\{O_i\}_{0 \leq i < n}$ ordered with increasing distance from the viewer, we define a probability function $p_{cov}(O_i)$ that describes the chance of the object O_i to be completely covered. The probability function is based on the fraction of screen pixels $c_{scr}(O_i)$ covered by all objects closer to the viewer than O_i, which is the fraction of pixels in front of the current object O_i. Note that this function does not exploit temporal coherence yet, which will be addressed later. Since the probability estimation should not become the bottleneck of the algorithm, we derive $p_{cov}(O_i)$ from this single coverage value and assume O_i to be randomly placed on the screen. A theoretically better solution would be to use a grid, but in practice the overhead for this proved to be too high already at very low grid resolutions. Now let $c_{scr}(O_i)$ be known, $c(O_i)$ the screen fraction covered by O_i, then the average fraction of visible pixels of O_i is $(1 - c_{scr}(O_i))c(O_i)$. This gives an expected value for $c_{scr}(O_{i+1})$ of

$$c_{scr}(O_{i+1}) = c_{scr}(O_i) + (1 - c_{scr}(O_i))\, c(O_i).$$

Since calculating the exact value of $c(O_i)$ would require an occlusion query by itself, we use an approximation based on the fraction of pixels $c_{bb}(O_i)$ covered by the bounding box of O_i, which can be calculated efficiently. Let $R_{cov}(O_i)$ be the average ratio of $c_{bb}(O_i)$ to $c(O_i)$, then we can approximate

$$c(O_i) \approx R_{cov}(O_i) c_{bb}(O_i).$$

Then let $A_{bb}(O_i)$ be the surface area of the bounding box, $d(O_i)$ the distance between the bounding box of O_i and the viewer, w and h the width and height of the screen in pixels, and θ the vertical field of view. Then the screen fraction $c_{bb}(O_i)$ covered by the bounding box is approximated by:

$$c_{bb} \approx \left(\frac{1}{d(O_i)\sqrt{\frac{w}{h}} 2 \tan \frac{\theta}{2}} \right)^2 \frac{A_{bb}(O_i)}{6}.$$

$p_{occl}(O_i)$	probability of O_i being occluded
$p_{cov}(O_i)$	probability of O_i being covered
$c_{scr}(O_i)$	fraction of pixels covered by all objects closer to the viewer than O_i
$c(O_i)$	fraction of pixels covered by O_i
$c_{bb}(O_i)$	fraction of pixels covered by the bounding box of O_i

Table 1: *Symbols used for the probability estimation.*

To estimate the ratio $R_{cov}(O_i)$, we assume that O_i is sphere-like. Given the surface area, the radius of the sphere is $r^2 = \frac{1}{4\pi} A(O_i)$. After projection, the covered area $A_{proj}(O_i)$ of the object is $\pi r^2 = \frac{1}{4} A(O_i)$. If we assume that the bounding box is viewed from the front, it covers the area of $\frac{1}{6} A_{bb}$ after projection, which means that

$$R_{cov}(O_i) \approx \frac{3}{2} \frac{A(O_i)}{A_{bb}(O_i)}.$$

Based on $c_{scr}(O_i)$, we derive a model for the probability $p_{cov}(O_i)$ that all pixels of O_i are already covered. For this model we assume that both the pixels covered by O_i and those covered by O_0 to O_{i-1} form rectangles with the screen aspect ratio, as shown in Figure 2.
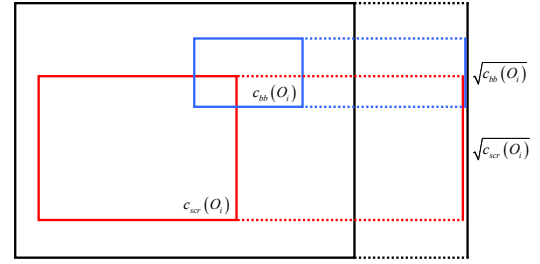

Figure 2: *Model used to estimate the coverage probability.*

Again, let O_i have a random position on screen leading to

$$p_{cov}(O_i) = \begin{cases} \left(\sqrt{c_{scr}(O_i)} - \sqrt{c_{bb}(O_i)} \right)^2 & : c_{bb}(O_i) < c_{scr}(O_i) \\ 0 & : c_{bb}(O_i) \geq c_{scr}(O_i) \end{cases}.$$

Figure 3 shows how well the estimated probability fits to a measured distribution for randomly placed objects. The measurement was performed by drawing 10,000 random ellipsoids distributed in the view frustum. This was repeated 100 times to obtain the average visibility probability. Note that the noise is due to the low number of samples for some combinations of $c_{scr}(O_i)$ and $c_{bb}(O_i)$, as especially large bounding boxes with high screen coverage are rare in this setting.

In addition to the coverage probability $p_{cov}(O_i)$, temporal coherence is exploited by using the occlusion status of O_i in the last frame to estimate the occlusion probability $p_{occl}(O_i)$
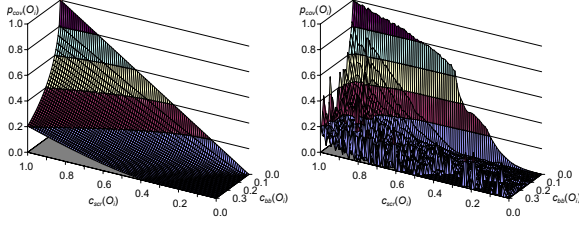
Figure 3: *Comparison of estimated and measured $p_{cov}(O_i)$ against estimated $c_{scr}(O_i)$ and $c_{bb}(O_i)$.*

in the current frame. First of all, if O_i was occluded, we assume that it will be occluded again. Second, if O_i was visible, we need to consider the probability that it will be occluded for two consecutive frames $(p_{cov}(O_i)^2)$ to exploit coherence. In addition, visible objects tend to remain visible and thus $p_{occl}(O_i)$ will be lower than $p_{cov}(O_i)$. Finally, if O_i was frustum culled, we cannot exploit coherency so we use $p_{cov}(O_i)$ directly. Therefore, the occlusion probability is:

$$p_{occl}(O_i) = \begin{cases} \frac{1}{2}p_{cov}(O_i)^2 & : & \text{prev. visible} \\ p_{cov}(O_i) & : & \text{prev. outside view frustum} \\ 1 & : & \text{prev. occluded} \end{cases}$$

3.2. Render and query time

In addition to $p_o(O_i)$, the times required for rendering $t_r(O_i)$ and for the query $t_o(O_i)$ need to be known.

3.2.1. Parameterizing the hardware

The rendering pipeline of today's hardware is basically divided into three parallel stages, the setup stage, the vertex stage and the pixel/fragment stage. When several parallel rendering calls are issued, the slowest of these stages determines the performance, i.e. $t_r(O_i) = \max(t_r^s(O_i), t_r^v(O_i), t_r^f(O_i))$. Note that this formula is a combination of the ones used by Funkhouser and Séquin [FS93] and by Wimmer and Wonka [WW03], since Funkhouser and Séquin considered the pipelining but of course not the architectural changes of graphics hardware in the last decade. Wimmer and Wonka neglected the pipelining as they required a reliable upper bound for the rendering time and the introduced overestimation increased the robustness of their method.

To estimate the rendering and occlusion query times, the time required for each of the three graphics pipeline stages needs to be estimated. Since $t_r^v(O_i)$ depends linearly on the number of triangles $N_\triangle(O_i)$ and $t_r^f(O_i)$ linearly on the number of processed fragments $f(O_i)$, the time required for processing O_i during these stages can be calculated from the material dependent times per triangle $T_r^\triangle(m_j)$ and per fragment $T_r^f(m_j)$ with $t_r^v(O_i) = N_\triangle(O_i)T_r^\triangle(m_j)$ and $t_r^f(O_i) = f_o(O_i)T_r^f(m_j)$. For occlusion queries, the estimation of $t_o^v(O_i)$ is not required, since the bounding vol-

umes have a constant low number of triangles which can be accounted for within T_o^s. Instead of the number of fragments $f(O_i)$ covered by the object, the number of fragments $f_{bb}(O_i)$ covered by the bounding box of O_i is required to estimate $t_o^f(O_i)$. As it is also possible to issue a query along with the rendering itself to use the occlusion status of O_i in the next frame, the overhead T_o^o compared to rendering without a query must also be measured. Table 2 summarizes the constant characteristic times required for the rendering and occlusion time estimations.

$T_r^s(m_j)$	setup time per rendering call
$T_r^\triangle(m_j)$	time per rendered triangle
$T_r^f(m_j)$	time per shaded fragment
T_o^s	setup time per occlusion query
T_o^f	time per fragment during occlusion query
T_o^l	maximum occlusion query latency
T_o^o	overhead time for a query during rendering

Table 2: *Hardware dependent parameters required for rendering and occlusion time estimation.*

While $N_\triangle(O_i)$ is constant, $f(O_i)$ and $f_{bb}(O_i)$ change with every frame. As $f(O_i)$ cannot be calculated exactly without the rasterization of O_i, we derive it from $c_o(O_i)$. To account for the possible overdraw during rasterization, we presume two fragments per pixel and thus $f(O_i) = 2wh \cdot c_o(O_i)$ and $f_{bb}(O_i) = wh \cdot c_{bb}(O_i)$.

3.2.2. Parameter measurement

To measure these characteristic times two triangle meshes are used: one with a high number of triangles (O_+) to determine $T_r^\triangle(m_j)$, and one with a low number (O_-) to measure $T_r^s(m_j)$ and $T_r^f(m_j)$. For $T_r^\triangle(m_j)$ and $T_r^s(m_j)$, O_+ and O_- are rendered pixel-sized and for the per fragment performance, O_- is rendered filling the whole screen. The setup and per fragment times for the occlusion query are measured analogously. These measurements are required for each material, but materials can be clustered into groups of similar shader complexity and thus the total number of such groups is usually very low in practice.

4. Rendering Algorithm

Since all occlusion culling techniques are based on exploiting spatial coherence, first a hierarchy must be generated for a given scene. The type of hierarchy depends on the requirements of the application, e.g. whether the scene is mainly static or fully dynamic. In our tests we used the p-HBVO algorithm [MBH*01] since our models were static. As required for any hardware occlusion culling technique, this scene hierarchy is traversed in a front to back order, while issuing occlusion queries. Note that sorting and traversal do not slow down rendering, since they are performed

on the otherwise almost idle CPU. In contrast to Bittner et al. [BWPP04] we consider the characteristics of the graphics hardware and perform a cost/benefit balancing to amortize the cost of wasted queries over time. In addition, we do not only issue queries for termination nodes, but consider nodes on all levels of the hierarchy to find the optimal balance between query time and expected speedup. This also means that unlike the CHC method multiple queries can be issued for a subtree in case previous queries did not succeed but subsequent queries are still reasonable according to the heuristic. Additionally it also allows using the query result for a performance gain already in the current frame. Figure 4 shows the pseudo-code for the traversal algorithm.

```
DistanceQueue.Insert(Root);
while( ¬DistanceQueue.Empty() ∨ ¬QueryQueue.Empty() )
    while( ¬QueryQueue.Empty() ∧ FirstQueryFinished() )
        Node = QueryQueue.Pop();
        Node.SetVisible(GetQueryResult(Node));
        if( Node.IsVisible() )
            if( Node.IsLeaf() )
                SetParentsVisible(Node);
            if( Node.WaitForResult() )
                Process(Node);
        else if( ¬Node.WaitForResult() )
            QueryQueue.Remove(Node.Children());
            DistanceQueue.Remove(Node.Children());
    if( ¬DistanceQueue.Empty() )
        Node = DistanceQueue.Pop();
        Node.SetVisible(false);
        if( InsideViewFrustum(Node) )
            if( QueryReasonable(Node) )
                IssueQuery(Node);
                QueryQueue.Insert(Node);
                if( ¬Node.WaitForResult() )
                    Process(Node);
            else
                Process(Node);

Process(NodeType Node)
if( Node.IsLeaf() )
    Render(Node);
else
    DistanceQueue.Insert(Node.Children());
```

Figure 4: *Pseudo-code of the traversal method. Differences to the CHC algorithm are emphasized.*

The test whether an occlusion query is reasonable now depends on two factors: a) the performance tradeoff between query cost and expected benefit (Section 4.1); b) the cost and benefit of the current node compared to that of its children (Section 4.2). If a query for an inner node is issued, the query latency must be considered in the decision whether the child nodes are added to the traversal queue immediately or only when the current node is found to be visible (Section 4.3).

4.1. Performance tradeoff

Issuing an occlusion query for a node H_i is clearly not reasonable if rendering the node is faster, i.e. $t_r(H_i) < t_o(H_i)$, so queries are never issued for such nodes. While nodes that were previously occluded are always queried, an additional cost/benefit balancing is performed for nodes that were previously visible by issuing an occlusion query only after the node has been rendered without querying for n frames, such that the cost $\mathcal{C}(H_i)$ for the occlusion query is compensated by the benefit $\mathcal{B}(H_i)$ of a possible occlusion. This leads to the condition, that $\mathcal{C}(H_i) \leq n\mathcal{B}(H_i)$. Since the benefit is accumulated over all levels of the hierarchy while the cost is per level, the benefit needs to be evenly distributed among all levels by dividing it with the depth of the hierarchy. Given the total number of hierarchy nodes N_h, we obtain:

$$\mathcal{C}(H_i) = t_o(H_i)$$
$$\mathcal{B}(H_i) = p_o(H_i)(t_r(H_i) - t_o(H_i))/\log_2(N_h + 1).$$

If H_i was removed by the view frustum culling in the last frame, the estimated processing time $t_e(H_i)$ including an occlusion query is $t_e(H_i) = t_o(H_i) + (1 - p_o(H_i))t_r(H_i)$. If $t_e(H_i) < t_r(H_i)$, a query is issued, otherwise H_i is treated as if it was tested and found visible in the current frame.

4.2. Granularity

Let $H_{j_0}, ..., H_{j_n} \in S(H_i)$ be the children of the currently processed node, H_i. Since during traversal the view frustum culling and distance calculation are also performed for H_{j_k}, $t_r(H_{j_k})$ and $t_o(H_{j_k})$ can be estimated as well. We use this possibility to evaluate if $\sum_k \mathcal{C}(H_{j_k}) < \mathcal{C}(H_i)$ holds, which is the case for example if there exists at least one k for which H_{j_k} is view frustum culled or $\sum_k c_{bb}(H_{j_k}) \ll c_{bb}(H_i)$ holds. In this case no query is issued for H_i since it is cheaper to query the child nodes. Analogously a query is only issued, if the benefit for the current node is higher than for its children and thus $\mathcal{B}(H_i) > \sum_k \mathcal{B}(H_{j_k})$. Now let $N_l(H_i)$ denote the number of leaves in the subtree for which the root node is H_i. In order to issue a query for H_i it is also required that $T_o^o N_l(H_i) > t_o(H_i)$ holds, as otherwise querying the leaf nodes during rendering is cheaper. The only exception to the last two rules is, if there is at least one k for which $t_r(H_{j_k}) < t_o(H_{j_k})$ holds meaning that the current node is the last one for which the complete subtree will be queried. Together with the performance tradeoff, these conditions define the test if a query is reasonable, which is shown in Figure 5 as pseudo-code.

4.3. Latency

Let $S(H_i)$ denote the set of nodes which are the child nodes of H_i. Due to the latency introduced by the query, a choice has to be made to either insert $S(H_i)$ into the traversal queue immediately (and remove them again later if H_i was occluded), or to only insert them later when H_i is found to be visible. If $S(H_i)$ are inserted only after the occlusion query

```
if( (Node.RenderTime() < Node.QueryTime())
 ∨ (∑_Child Child.Cost() < Node.Cost()) )
    return false;
if( Node.WasOccluded() )
    return true;
if( Node.Cost() > Node.FramesSinceLastQry() · Node.Benefit() )
    return false;
forall Child ∈ Node.Children()
    if( Child.RenderTime() < Child.QueryTime() )
        return true;
if( (Node.Benefit() > ∑_Child Child.Benefit())
 ∧ (T_o^o · Node.NumLeaves() > Node.QueryTime()) )
    return true;
return false;
```

Figure 5: *Pseudo-code of reasonability test.*

failed, the one frame delay of the CHC algorithm is eliminated, but the front to back traversal is not maintained anymore and nodes occluded by $S(H_i)$ can erroneously found to be visible. If $S(H_i)$ are directly inserted however, the time spent to process them is wasted if the occlusion query for H_i succeeds and the effect of the delay is only reduced. We address this problem slightly differently than the CHC algorithm by directly inserting $S(H_i)$ if H_i was previously visible or view frustum culled, and only delaying the insertion of $S(H_i)$ if H_i was previously occluded whereas CHC only inserts H_i directly if it was previously visible.

In addition, a synchronization between CPU and GPU is performed in order to minimize both out-of-order and unnecessary processing. This can be accomplished by using the graphics API synchronization (e.g. glFlush()) that waits until the last issued command starts executing if supported by the driver. If this is not the case – which is identified during the measurements – another possibility to synchronize is to calculate the maximum number of possible parallel queries N_q and assume that the first query is finished when the number of active queries reaches N_q. Given the maximum query latency T_o^l which is also measured along with the other hardware dependent characteristic times, we obtain:

$$N_q = \left\lceil \frac{T_o^l}{\min(T_o^s, T_o^o + \min_i T_r^s(m_i))} \right\rceil + 1.$$

If the synchronization is supported by the API, both are used to minimize the negative effects. Otherwise only the maximum number of parallel queries is used. The synchronization is performed when the algorithm checks if the first query is already finished. The test based on N_q is free, while the API provided one is only used when it is reasonably fast and thus the cost of the CPU/GPU synchronization is negligible.

5. Results

We have integrated our method into a simple OpenGL-based scene graph and performed benchmark tests on four different scene types shown in Figure 6 – the vertex transform limited Power Plant model, the fragment shading limited Vienna, the low depth complexity Dragon model, and the moderate depth complexity C-Class model – with different graphics cards and quality settings.
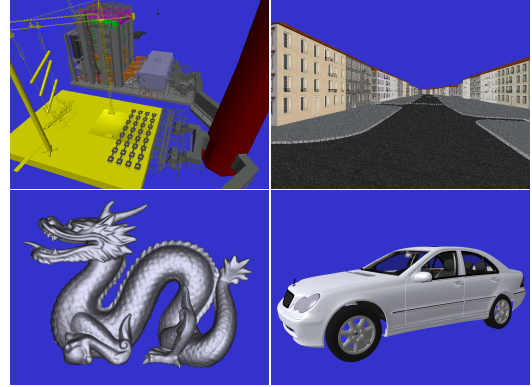

Figure 6: *Models used for measurements.*

Table 3 shows the number of triangles and hierarchy nodes for all models. Since the Dragon and some objects in the Power Plant model consist of several hundred thousand triangles, first all objects are subdivided recursively until each object contains at most 1,000 triangles also using the p-HBVO algorithm. This extra subdivision improves the performance of all methods.

	Power Plant	Vienna	Dragon	C-Class
#triangles	12,748,510	892,920	871,414	1,861,466
#nodes	38,867	20,021	2,535	5,775

Table 3: *Model statistics.*

5.1. Overall performance comparison

Table 4 shows the average and minimum frame rates achieved when using view frustum culling only (VFC), coherent hierarchical culling (CHC) [BWPP04], the proposed method (NOHC) and the theoretically optimal algorithm that only queries occluded nodes for which issuing a query is faster than simply rendering them. In addition, a node is not queried if querying the children is faster. Comparing to this theoretical algorithm gives the overhead required for the wasted queries using the CHC algorithm and thus shows how much it is reduced with our approach. The results in Table 4 were obtained using the latest (as of April 2006) drivers both for ATI and nVidia cards. In addition to using different graphics cards, we also tested the method using two driver performance/quality settings, where the maximum performance setting means no anti-aliasing, no anisotropic filtering and only bilinear texture filtering (no mip-mapping) and the high quality setting refers to maximum anti-aliasing,

method	Radeon 9800XT performance	Radeon 9800XT quality	GeForce 5900Ultra performance	GeForce 5900Ultra quality	GeForce 7800GTX performance	GeForce 7800GTX quality
Power Plant						
VFC	16.52/143.45	19.96/2580.62	12.51/152.86	18.88/224.16	6.28/103.09	7.86/128.21
CHC	7.07/ 41.53	10.64/ 135.27	6.29/ 46.91	11.21/ 86.27	2.64/ 22.73	3.98/ 36.90
NOHC	6.87/ 34.22	10.37/ 92.01	4.84/ 40.01	9.45/ 69.48	1.88/ 22.03	3.11/ 35.59
optimum	6.31/ 23.97	8.71/ 81.21	3.77/ 27.41	7.01/ 47.24	1.59/ 20.04	2.62/ 25.51
Vienna						
VFC	26.65/86.62	26.68/86.72	16.31/60.26	21.80/63.62	8.90/39.37	11.91/39.53
CHC	8.67/66.86	10.12/78.90	5.74/62.95	11.66/64.92	2.40/29.67	5.51/35.34
NOHC	7.67/53.75	9.67/59.15	3.37/45.83	9.21/51.12	1.84/26.32	4.07/26.60
optimum	7.35/41.44	8.95/48.91	2.98/29.96	7.85/40.23	1.60/17.30	3.92/18.66
Dragon						
VFC	10.05/10.13	10.26/10.71	12.13/12.35	13.33/14.07	7.12/7.25	7.15/7.31
CHC	8.79/11.38	9.12/11.68	11.42/14.83	12.43/14.95	6.24/7.53	6.81/7.62
NOHC	6.94/ 8.24	7.20/ 9.12	8.17/10.04	9.59/11.63	3.59/4.36	3.86/4.60
optimum	6.15/ 7.81	7.01/ 8.92	7.59/ 9.38	8.97/11.04	3.43/4.30	3.72/4.53
C-Class						
VFC	26.32/27.31	26.93/28.88	31.81/33.27	32.97/34.15	18.62/19.55	18.83/19.65
CHC	15.17/17.66	15.66/18.04	19.63/22.98	24.43/28.02	10.74/11.64	13.37/14.29
NOHC	12.96/14.08	13.47/15.56	15.27/17.11	16.35/19.05	6.75/ 7.42	6.95/ 7.95
optimum	10.47/11.92	11.94/13.65	12.94/14.38	15.41/17.04	5.84/ 6.57	6.37/ 6.96

Table 4: *Comparison of average/maximum frame time in milliseconds for different culling techniques, graphics cards, and driver performance/quality settings.*

maximum anisotropic filtering and trilinear texture filtering between mip-map levels. From this comparison it is already visible that the proposed method reduces the maximum frame time compared to state-of-the-art culling techniques, independently of the used graphics hardware, quality setting and type of model. Note that in cases when the CHC algorithm is close to the optimum (e.g. the Power Plant model on the GeForce 7800GTX card) the improvement is minor. However, when the overhead introduced by the CHC method is high (e.g. on the GeForce 5900Ultra card), the improvement is significant. In addition to the reduced maximum frame time, the average frame time is also improved. Here the improvement however depends on the temporal coherence of occlusions and is thus more distinct for the Vienna, Dragon, and C-Class models – since they are more or less closed surfaces – than for the Power Plant model.

5.2. Detailed analysis

Since the shortcomings of the CHC method are most apparent on the GeForce 5900 with high performance settings, we analyze this configuration more extensively. Figure 7 shows a frame time comparison for a part of the Vienna walkthrough with high depth complexity. In this case view frustum culling would trivially perform much worse than occlusion culling, therefore comparison to view frustum culling is omitted. The overhead due to failed occlusion queries is significantly reduced compared to the CHC algorithm, almost doubling the performance on average. The spikes around the 8th sec and 16th sec are due to sudden viewpoint changes, which obviously do not influence the optimal algorithm.
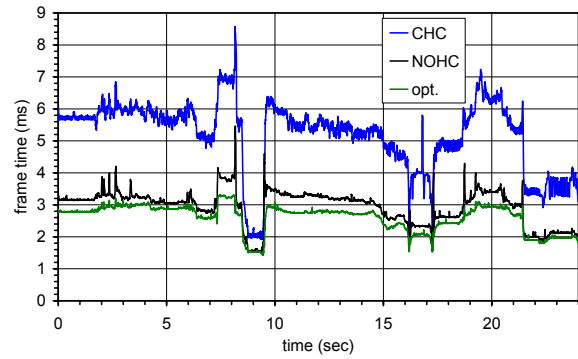

Figure 7: *Frame time comparison for the Vienna model with high depth complexity on a GeForce 5900Ultra at maximum performance driver settings.*

In low depth complexity situations however, occlusion culling might degrade performance so comparison to view frustum culling is important. Figure 8 shows a comparison during a period of low depth complexity in the middle of the Vienna walkthrough. In contrast to the CHC algorithm the performance of the proposed method is always superior to view frustum culling alone. This shows how well our method adapts to these worst case situations while also improving both the average and best case performance.

On the other hand, the graph also shows the limitations of the approach due to inaccurate estimation of the occlusion probability. When the probability is estimated too high (e.g. from the 34.5th sec to the 35th sec in the Vienna walkthrough), still some unnecessary queries are issued and the improve-
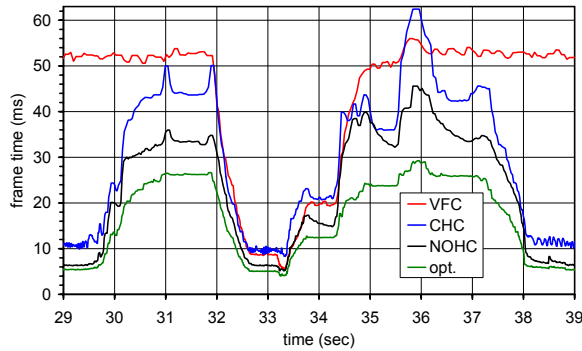
Figure 8: *Frame time comparison for the Vienna model with low depth complexity on a GeForce 5900Ultra at maximum performance driver settings.*

ment over the CHC algorithm is reduced. When the probability is estimated too low, the rendering time only gradually approaches the optimum (e.g. shortly after the 38th sec in the Vienna walkthrough). The second effect is however less noticeable, if the temporal and spatial coherence of occlusions is high. Therefore, it does not degrade the performance for the Dragon and C-Class model. In general, even if the assumptions made for the analytical models do not hold in a particular situation, the method still performs at least on par with (and usually better than) view frustum culling and practically always has a much smaller overhead than the CHC algorithm.

6. Conclusion and Future Work

We have presented a method to optimize the scheduling of hardware occlusion queries with respect to the performance characteristics of the currently used graphics hardware. It is very flexible and can be easily integrated into existing real-time rendering packages using arbitrary, application dependent scene hierarchies and bounding volumes. 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. The major potential in improving the method lies in developing more accurate, yet not much more complex, analytical models for the occlusion probability and the rendering/query time estimation, e.g. that recalibrate themselves during rendering. We have also observed during measurements that moderately increasing the number of triangles of the bounding volumes does not affect the query time. Therefore, tighter bounding volumes (e.g. k-dops) could also be used from which all hardware accelerated occlusion culling methods would benefit.

References

[BW03] BITTNER J., WONKA P.: Visibility in computer graphics. *Environment and Planning B: Planning and Design 30*, 5 (September 2003), 729–756. 2

[BWPP04] BITTNER J., WIMMER M., PIRINGER H., PURGATHOFER W.: Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum (Eurographics 2004) 23*, 3 (September 2004), 615–624. 1, 2, 5, 6

[COCSD03] COHEN-OR D., CHRYSANTHOU Y., SILVA C. T., DURAND F.: A survey of visibility for walkthrough applications. *IEEE Trans. Vis. Comput. Graph. 9*, 3 (2003), 412–431. 2

[FS93] FUNKHOUSER T. A., SÉQUIN C. H.: Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics (Proceedings of SIGGRAPH 93) 27*, 2 (1993), 247–254. 2, 4

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical Zbuffer visibility. *Computer Graphics (Proceedings of SIGGRAPH 93) 27*, 2 (1993), 231–238. 2

[HMC*97] HUDSON T., MANOCHA D., COHEN J., LIN M., HOFF K., ZHANG H.: Accelerated occlusion culling using shadow frusta. In *Proceedings of the Thirteenth ACM Symposium on Computational Geometry* (1997), pp. 1–10. 2

[KS01] KLOSOWSKI J. T., SILVA. C. T.: Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics 7*, 4 (2001), 365–379. 2

[MBH*01] MEISSNER M., BARTZ D., HÜTTNER T., MÜLLER G., EINIGHAMMER J.: Generation of Decomposition Hierarchies for Efficient Occlusion Culling of Large Polygonal Models. In *Vision, Modeling, and Visualization* (2001), pp. 225–232. 4

[NA01] NVIDIA, ATI: ARB occlusion query. http://oss.sgi.com/projects/ogl-sample/registry/ARB/occlusion_query.txt, 2001. 2

[SBS04] STANEKER D., BARTZ D., STRASSER W.: Occlusion Culling in OpenSG PLUS. *Computers & Graphics 28*, 1 (2004), 87–92. 2

[TS91] TELLER S. J., SÉQUIN C. H.: Visibility preprocessing for interactive walkthroughs. *Computer Graphics (Proceedings of SIGGRAPH 91) 25*, 2 (1991), 61–69. 2

[WW03] WIMMER M., WONKA P.: Rendering time estimation for real-time rendering. In *Rendering Techniques (Proceedings of Eurographics Workshop on Rendering)* (2003), pp. 118–129. 2, 4

[ZMHH97] ZHANG H., MANOCHA D., HUDSON T., HOFF III K. E.: Visibility culling using hierarchical occlusion maps. In *ACM SIGGRAPH 97* (1997), pp. 77–88. 2