EUROGRAPHICS 2008 / G. Drettakis and R. Scopigno
(Guest Editors)

Volume 27 (2008), Number 3

CHC++: Coherent Hierarchical Culling Revisited

Oliver Mattausch[1], Jiří Bittner[2], Michael Wimmer[1]

[1]Vienna University of Technology, Austria
[2]Czech Technical University in Prague, Czech Republic

Abstract

We present a new algorithm for efficient occlusion culling using hardware occlusion queries. The algorithm significantly improves on previous techniques by making better use of temporal and spatial coherence of visibility. This is achieved by using adaptive visibility prediction and query batching. As a result of the new optimizations the number of issued occlusion queries and the number of rendering state changes are significantly reduced. We also propose a simple method for determining tighter bounding volumes for occlusion queries and a method which further reduces the pipeline stalls. The proposed method provides up to an order of magnitude speedup over the previous state of the art. The new technique is simple to implement, does not rely on hardware calibration and integrates well with modern game engines.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling

1. Introduction

Occlusion culling is an important technique to reduce the time for rendering complex scenes. The availability of so-called hardware occlusion queries has made runtime determination of visibility attractive. Hardware occlusion queries are a mechanism by which graphics hardware can quickly report the visibility status of simple proxy geometry. However it was only by exploiting temporal coherence, e.g. in the Coherent Hierarchical Culling (CHC) algorithm [BWPP04], that using hardware occlusion queries became feasible, as this avoids CPU stalls and GPU starvation.

The CHC algorithm works well in densely occluded scenes, but the overhead of hardware occlusion queries makes it fall behind even simple view-frustum culling (VFC) in some situations. This was recognized by Guthe et al. [GBK06], who provide an algorithm, called Near Optimal Hierarchical Culling (NOHC), which reduces the number of queries based on a clever statistical model of occlusion and a hardware calibration step. However, it turns out that even the optimum defined by Guthe et al. can still be improved by exploiting further sources of simplification.

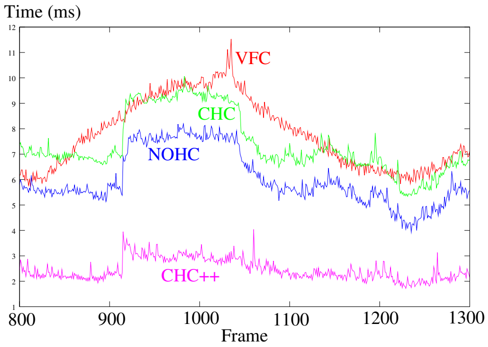

Figure 1: *Frame time comparison for a walkthrough of the Powerplant model for View Frustum Culling (VFC), Coherent Hierarchical Culling (CHC), Near Optimal Hierarchical Culling (NOHC), and our new algorithm (CHC++).*

In this paper, we propose CHC++, a method that significantly improves on previous online occlusion culling methods (see Figure 1). The core of the algorithm remains simple, requires no calibration, and allows easy integration into a game engine. The major contributions of the method are:

- **Reduction of state changes.** Despite its importance, the reduction of state changes was not explicitly addressed by previous occlusion culling methods. Our method provides a powerful mechanism to minimize the number of state changes by using batching of queries. As a result the total number of state changes is reduced by more than an order of magnitude (see Figure 2).

- **Reduction of number of queries.** Reducing the number of queries was a major goal of previous research on hardware based occlusion culling. For example, the NOHC algorithm proposed by Guthe et al. [GBK06] is very successful at reducing the number of queries for views with low occlusion. We propose two new methods for further reduction of the number of queries. The first method resolves visibility of many nodes in the hierarchy by a single query, the second method exploits tighter bounding volumes for the queries without the need for any auxiliary data structures like oriented bounding boxes or k-dops. As a result we achieve a significantly lower number of queries than the "optimal" algorithm defined by Guthe et al. [GBK06] (see Figure 2).
- **Reduction of CPU stalls.** The CHC algorithm does a good job at reducing CPU stalls, however in certain scenarios stalls still occur and cause a performance penalty. We propose a simple modification which provides further reduction of the wait time, which at the same time integrates well with our method for reducing state changes.
- **Reduction of rendered geometry.** Tighter bounding volumes will reduce the overestimation of visibility caused by bounding volumes and therefore reduce the amount of geometry classified as visible.
- **Integration with game engines.** Most game engines incorporate a highly optimized rendering loop in which sorting by materials and shaders is performed in order to minimize rendering state changes. Our method allows the rendering engine to perform such a sort on a batch of primitives stored in a render queue. Additionally the proposed technique significantly reduces the number of engine calls.

2. Related Work

Even in CPU-limited applications, which often occur with today's rapidly evolving graphics hardware, visibility culling can significantly reduce the time spent in the graphics driver and the rendering API, allowing better usage of the graphics hardware. For a general overview of visibility culling please refer to the thorough surveys of Cohen-Or et al. [COCSD02] and Bittner and Wonka [BW03].

Visibility algorithms can be roughly categorized into those that work as a preprocessing step and those that work at runtime. While preprocessing algorithms have no runtime overhead, they are often difficult to implement and work for static scenes only. Online occlusion culling on the other hand does not rely on a lengthy preprocessing step, is potentially more accurate as it computes visibility from a point, and allows for fully dynamic scenes. As most online culling algorithms work in image space, they allow automatic occluder fusion using rasterization. Before dedicated hardware support existed, online occlusion culling was mostly considered too costly for practical use, with some notable exceptions such as the Hierarchical Occlusion Maps from Zhang et al. [ZMHH97] or the dPVS system from Aila et al. [AM04].

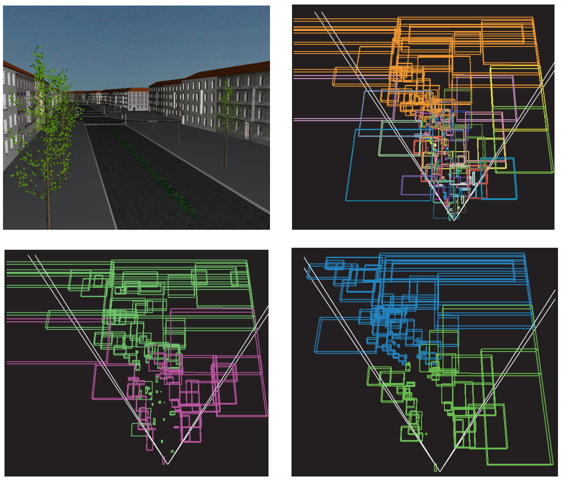

Figure 2: *Top left: A sample view point in a city scene. Top right: State changes required by the CHC algorithm (number of state changes = number of different colors of hierarchy nodes). Bottom left: State changes required by the CHC++ algorithm. Bottom right: Multiqueries: all invisible nodes are covered by only two occlusion queries (shown in different colors).*

With the introduction of hardware accelerated occlusion queries, online occlusion culling gained a lot of popularity. Hardware occlusion queries are relatively lightweight instructions that return the number of visible pixels of proxy geometry without the need of reading back the frame buffer. They opened the field for a variety of algorithms [KS01, HSLM02, GSYM03, SBS04, KS05]. However, the queries still come with a cost, so a naive implementation can be very slow due to idle time of the GPU and CPU that is caused by waiting for the query to return.

Coherent Hierarchical Culling [BWPP04], and later Near Optimal Hierarchical Culling [GBK06] avoid idle time by making use of temporal coherence. They will be discussed in more detail in the following section.

3. Overview

In this section we briefly review the CHC and the NOHC algorithms and discuss some of their issues. Then we describe the major components of the new CHC++ algorithm.

3.1. CHC and its problems

The Coherent Hierarchical Culling algorithm [BWPP04] makes use of temporal and spatial coherence to reduce the overhead and latency of hardware occlusion queries. It traverses the hierarchy in a front-to-back order and issues queries only for previously visible leaves and nodes of the previously invisible boundary. Previously visible leaves are assumed to stay visible in the current frame, and hence they are rendered immediately. The result of the query for these nodes only updates their classification for the next frame. The invisible nodes are assumed to stay invisible, but the algorithm retrieves the query result in the current frame in order to discover visibility changes. Refer to Figure 6 for the pseudocode of the original original CHC algorithm (unmarked parts).

The reduction of the number of queries (queries are not issued on previously visible interior nodes) and clever interleaving reduced the overhead of occlusion queries to an acceptable quantity. The algorithm works very well for scenarios that have a lot of occlusion. However, on newer hardware where rendering geometry becomes cheap compared to querying, or view points where much of the scene is visible, the method can become even slower than conventional view-frustum culling. This is a result of wasted queries and unnecessary state changes. This problem makes the CHC algorithm less attractive for game developers, who call for an algorithm which is reliably faster than view-frustum culling. Another problem of CHC lies in the complicated integration of the method into the rendering loop of highly optimized game engines. CHC interleaves rendering and querying of individual nodes of the spatial hierarchy which does not allow the engine to perform material sorting and leads to a higher number of engine API calls.

3.2. NOHC and its problems

The Near Optimal Hierarchical Culling algorithm proposed by Guthe et al. [GBK06] tackles the problem of wasted queries. The method uses a calibrated model of graphics hardware to estimate costs of the queries and costs of rendering. It estimates occlusion of nodes by using a simple screen coverage model and further corrections assuming temporal coherence. The occlusion estimation and hardware model are used in a cost/benefit heuristics which decides whether to apply an occlusion query on the currently processed node. This heuristic uses a sophisticated reasonability test for queries with a couple of rules.

The algorithm saves a significant number of queries, especially queries which would be applied on visible nodes. This can lead to a significant improvement over the CHC algorithm if the assumed visibility optimization proposed for CHC is not used.

The results for NOHC indicate that with a proper hardware calibration the method always performs better than view-frustum culling. In their paper, Guthe et al. [GBK06] also defined an optimal culling algorithm based on occlusion queries. The optimal algorithm is derived under the assumption that the status of every culled node has to be verified by an occlusion query. The NOHC method then closely approaches the optimal algorithm.

In our paper we show that the definition of optimality used by Guthe et al. [GBK06] still leaves significant room for improvement. In fact, the CHC++ algorithm is always clearly below the optimum defined by Guthe et al.

NOHC requires a hardware calibration step in which the hardware parameters are measured in a preprocess using artificial rendering scenarios. Measuring accurate parameters of the model requires very careful implementation. However, it turns out that even if precisely implemented, these measurements need not reflect the complex processes of state changes, pipelining, and interleaving rendering and occlusion queries during actual walkthroughs. Our new method does not rely on hardware calibration and aims to minimize its dependence on external parameters. In fact it leaves the user with loosely setting a few parameters whose influence is well predictable.

3.3. Building blocks of CHC++

The new CHC++ algorithm method addresses all previously mentioned problems, and extends CHC by including the following new components:

Queues for batching of queries. Before a node is queried, it is appended to a queue. Separate queues are used for accumulating previously visible and previously invisible nodes. We use the queues to issue batches of queries instead of individual queries. This reduces state changes by one to two orders of magnitude. The batching of queries will be described in Section 4.

Multiqueries. We compile multiqueries (Section 5.1), which are able to cover more nodes by a single occlusion query. This reduces the number of queries for previously invisible nodes up to an order of magnitude.

Randomized sampling pattern for visible nodes. We apply a temporally jittered sampling pattern (Section 5.2) for scheduling queries for previously visible nodes. This reduces the number of queries for visible nodes and while spreading them evenly over the frames of the walkthrough.

Tight bounding volumes. We use tight bounding volumes (Section 6) without the need for their explicit construction. This provides a reduction of the number of rendered triangles as well as a reduction of the number of queries.

Note that for all tests presented in the paper we used an axis-aligned bounding volume hierarchy (BVH) constructed according to the surface area heuristics [MB90]. The presented methods are however compatible with other types of spatial hierarchies [MBM*01], except for the tight bounding volumes optimization, which explicitly exploits the properties of BVH.

4. Reducing state changes

Changes of rendering state constitute a significant cost in the rendering pipeline. Previous occlusion culling methods focused mainly on scheduling the queries in a way that hides latencies and keeps the GPU occupied, as well as reducing the overall number of queries. However, even if the number of queries is reduced, every remaining query potentially leads to a state change in which at least writing to color and depth buffers is disabled and then re-enabled after the query. If complex shaders are used then this state change also involves switching the shader on and off.

It turns out that these changes of rendering state cause an even larger overhead than the query itself. The overhead may be on the hardware side (e.g., flushing caches), on the driver side or even on the application side. Thus it is highly desirable to reduce the number of state changes to an acceptable amount: game developers refer to about 200 state changes per frame as an acceptable value on current hardware.

Query batching. Our solution to this problem is based on batching occlusion queries instead of issuing queries immediately when they are requested by the algorithm. The rendering state is changed only once per batch and thus the reduction of state changes directly corresponds to the size of the query batches we issue. The batching algorithm handles visible and invisible nodes differently as described in the following sections.

4.1. Batching previously invisible queries

The invisible nodes to be queried are appended to a queue which we call *i-queue*. When the number of nodes in the i-queue reaches a user-defined batch size *b*, we change the rendering state for querying and issue an occlusion query for each node in the i-queue (in Section 5.1 we will see how several nodes can be combined in one occlusion query in order to reduce the number of queries).

The batch size *b* is tightly connected with the reduction of render state changes, giving approximately *b* times less state changes than the CHC algorithm. On the other hand, batching effectively delays the availability of query results for invisible nodes, which means that visibility changes could be detected later and follow-up queries spawned by them would introduce further latency if there is not enough alternative work (e.g., rendering visible nodes) left.

An optimal value for *b* depends on the scene geometry, material shaders, and the capabilities of the rendering engine with respect to material sorting. For our scenes and rendering engine we observed that precise tuning of this parameter is not necessary and that values between 20 and 80 give a largely sufficient reduction of render state changes while not introducing additional latency into the method.

4.2. Batching previously visible queries

Recall that the CHC algorithm issues a query for previously visible node and renders the geometry of the node without waiting for the result of the query. Similarly to CHC, our proposed method renders the geometry of previously visible nodes during the hierarchy traversal. However the queries are not issued immediately. Instead the corresponding nodes are stored in a queue which we call *v-queue*.

An important observation is that the queries for these nodes are not critical for the current frame since their result will only be used in the next frame. We exploit this observation by using nodes from the v-queue to fill up waiting time: whenever the traversal queue is empty and no outstanding query result is available, we process nodes from the v-queue.

As a result we perform adaptive batching of queries for previously visible nodes driven by the latency of the outstanding queries. At the end of the frame, when all queries for previously invisible nodes have been processed, the method just applies a single large batch for all unprocessed nodes from the v-queue.

Note that before processing a node from the v-queue, we also check whether a render state change is required. It turns out that in the vast majority of cases there is no need to change the render state at all as it was already changed by a previously issued query batch for invisible nodes. Therefore, we have basically eliminated state changes for previously visible nodes.

As a beneficial side effect, the v-queue reduces the effect of violations of the front-to-back ordering made by the original CHC algorithm. In particular if a previously hidden node occludes a previously visible node in the current frame, this effect would only be captured in the next frame, as the previously visible node would often be queried before the previously invisible node is rendered. This issue becomes apparent in situations where many visibility changes happen at the same time. Delaying the queries using the v-queue will make it more likely for such visibility changes to be detected.

4.3. Game engine integration

For easy integration of the CHC++ method into existing game engines we propose to use an additional queue in the algorithm which we call *render queue*. This queue accumulates all nodes scheduled for rendering and is processed when a batch of queries is about to be issued. When processing the render queue the engine can apply its internal material shader sorting and then render the objects stored in the queue in the new order. Another beneficial effect of the render queue is the reduction of engine API calls. These calls can be very costly and thus their reduction provides significant speedup as we experienced for example with the popular OGRE game engine.

The overview of the different queues used by the CHC++

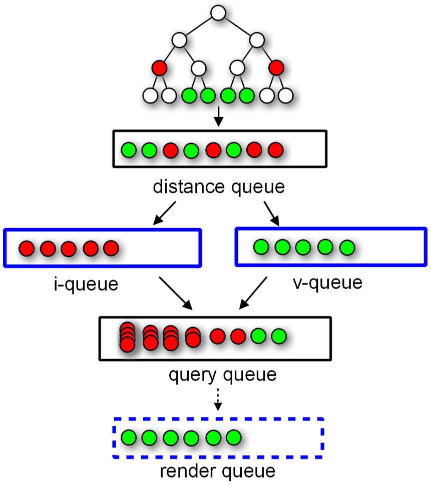

Figure 3: *Different queues used by the CHC++ algorithm. The queues which were not used by the CHC algorithm are highlighted in blue.*

algorithms is shown in Figure 3. Note that the overlaid nodes in the query queue correspond to multiqueries which will be discussed in Section 5.1.

5. Reducing the number of queries

Recent online occlusion culling methods focused on reducing the number of occlusion queries in order to reduce their overhead. In particular the method of Guthe et al. [GBK06] proposed a sophisticated approach for eliminating queries based on a cost/benefit heuristics and a calibrated model of the graphics hardware. In this section we propose two new methods which are able to reduce the number of queries even *below* the hypothetical algorithm previously defined as optimal by Guthe et al.

5.1. Multiqueries for invisible nodes

All previous techniques use one occlusion query per previously invisible primitive to be tested (node in a hierarchy, bounding volume, cell in a grid). The occlusion queries for these nodes were considered irreducible.

However, the following observation allows us to reduce the number of queries even for previously invisible nodes: If some previously invisible part of a scene remains invisible in the current frame, a single occlusion query for the whole part is sufficient to verify its visibility status. Such a query would render all bounding boxes of primitives in this scene part, and return zero if all primitives remain occluded. For example, in the extreme case of a static scene and a static view point, a single occlusion query could be used for all invisible parts of the scene.

Assuming a certain coherence of visibility, our new technique aims to identify such scene parts by forming groups of previously invisible nodes that are equally likely to remain invisible. A single occlusion query is issued for each such group, which we call a *multiquery*. If the multiquery returns zero, all nodes in the group remain invisible and their status has been updated by the single query. Otherwise the coherence was broken for this group and we issue individual queries for all nodes by reinserting them in the i-queue. Note that in the first case the number of queries is reduced by the number of primitives in the group. However, in the second case the multiquery for the batch was wasted.

We use an adaptive mechanism based on a cost/benefit heuristics to find suitable node groupings. The crucial part of the evaluation is the estimation of coherence in the visibility classification of the nodes, which is described in the next section. The actual heuristics will be described in Section 5.1.2.

5.1.1. Estimating coherence of visibility

In the vast majority of cases there is a strong coherence in visibility for most nodes in the hierarchy. Our aim is to quantify this coherence. In particular, knowing the visibility classification of a given node, we aim to estimate the probability that this node will keep its visibility classification in the next frame. Our experiments indicate that there is a strong correlation of this value with the "history" of the node, i.e., with the number of frames the node already kept the same visibility classification (we call this value *visibility persistence*). Nodes that have been invisible for a very long time are likely to stay invisible. Such nodes could be the engine block of a car, for example, that will never be visible unless the camera moves inside of the car engine. On the contrary, even in slow moving scenarios, there are always some nodes on the visible border which frequently change their classification. Hence there is a quite high chance for nodes that recently became invisible to become visible soon.

We define the desired probability as a function of the visibility persistence i, and approximate it based on the history of previous nodes:

$$p_{keep}(i) \approx \frac{n_i^{keep}}{n_i^{all}} \tag{1}$$

where n_i^{keep} is the number of already tested nodes which have been in the same state for i frames and keep their state in the $i+1$-th frame, and n_i^{all} is the total number of already tested nodes which have been in the same state for i frames. Figure 4 shows a plot of the probability p_{keep} against the visibility persistence i. The counters n_i^{keep} and n_i^{all} are accumulated over all previous frames of the walkthrough.

In the first few frames there are not enough measurements for accurate computation of $p_{keep}(i)$, especially for larger values of i. We solve this problem by piecewise constant propagation of the already computed values to the higher values of $p_{keep}(i)$.

As a simpler alternative to evaluating $p_{keep}(i)$ by measurements, we propose an analytic formula which corresponds reasonably well to the functions we measured for our scenes and walkthroughs:

$$p_{keep}(i) \approx 0.99 - 0.7e^{-i} \tag{2}$$

Using this function does not provide as accurate estimations of p_{keep} as the measured function, but can be used to avoid implementing the evaluation of the measured function. Figure 4 illustrates the analytic function and the measured functions for two different scenes.

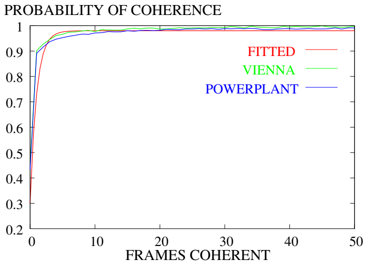

Figure 4: *$p_{keep}(i)$ in dependence on visibility persistence i. Note that the analytic function from Eq. 2 closely matches the functions measured for Powerplant and Vienna scenes.*

5.1.2. Cost/benefit model for multiqueries

Determining the optimal size of a multiquery for previously invisible nodes in a given batch (i.e., the i-queue) is a global optimization problem that requires evaluation of all possible partitions of the batch into multiqueries. Instead, we use a greedy model which maximizes a benefit/cost ratio for each multiquery.

The cost is the expected number of queries issued per one multiquery, which is expressed as:

$$C(M) = 1 + p_{fail}(M) * |M|, \tag{3}$$

where $p_{fail}(M)$ is the probability that the multiquery fails (returns visible, in which case all nodes have to be tested individually) and $|M|$ is the number of nodes in the multiquery. Note that the constant 1 represents the cost of the multiquery itself, whereas $p_{fail}(M) * |M|$ expresses the expected number of additionally issued queries for individual nodes. The probability p_{fail} is calculated from the visibility persistence values i_N of nodes in the multiquery as:

$$p_{fail}(M) = 1 - \prod_{\forall N \in M} p_{keep}(i_N), \tag{4}$$

The benefit of the multiquery is simply the number of nodes in the multiquery, i.e. $B(M) = |M|$.

Given the nodes in the i-queue, the greedy optimization algorithm maximizes the benefit at the given cost. We first sort the nodes in descending order based on their probability of staying invisible, i.e. $p_{keep}(i_N)$. Then, starting with the first node in the queue, we add the nodes to the multiquery and at each step we evaluate the value V of the multiquery as a benefit/cost ratio:

$$V(M_j) = \frac{B(M_j)}{C(M_j)} \tag{5}$$

It turns out that V reaches a maximum for a particular M_j and thus j corresponds to the optimal size of the multiquery for the nodes in the front of the i-queue. Once we find this maximum, we issue the multiquery for the corresponding nodes and repeat the process until the i-queue is used up. As a result we compile larger multiqueries for nodes with high probability of staying invisible and small multiqueries for nodes which are likely to turn visible.

5.2. Skipping tests of visible nodes

The original CHC algorithm introduced an important optimization in order to reduce the number of queries on previously visible nodes. A visible node is assumed to stay visible for n_{av} frames and it will only be tested in the frame $n_{av} + 1$. This optimization effectively reduces the average number of queries for previously visible leaves by a factor of $n_{av} + 1$.

This simple method however has a problem that the queries can be temporally aligned. This query alignment becomes problematic in situations when nodes tend to become visible in the same frame. For example consider the case when the view point moves from the ground level above the roof level in a typical city scene, causing many nodes to become visible in the same frame. Afterwards the queries of those nodes will be scheduled for the $n_{av} + 1$-th frame, and thus most of the queries will be aligned again. The average number of queries per frame will still be reduced, but the alignment can cause observable frame rate drops.

We observed that a randomization of $n_{av} + 1$ by a small random value $-r_{max} < r < r_{max}$ does not solve the problem in a satisfying manner. The problem is that if the randomization is small, the queries might still be very much aligned. On the other hand, if the randomization is big, some of the queries will be processed too late and thus the change from visible to invisible state will be captured too late.

We found that the most satisfying solution is achieved by

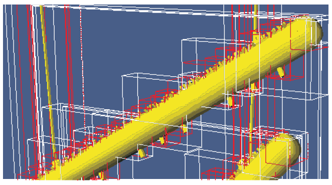

Figure 5: *Tight bounding volumes for nodes of the BVH which more closely represent the objects shown as spheres. The tight bounding volume consists of bounding boxes of the children (red) instead of the parent box (white).*

randomizing the *first invocation* of the occlusion query. After a node has turned visible, we use a random value $0 < r < n_{av}$ for determining the next frame when a query will be issued. Subsequently, if the node was already visible in the previous test, we use a regular sampling interval given by n_{av}.

We experimented with various values of n_{av}. The optimal value depends on the scene itself, inspection coherence, hardware parameters as well as the rendering engine parameters. Fortunately, our tests show that the dependence is not very strong and a value of $5-10$ has been a safe and robust choice for all tests.

6. Tighter bounding volumes

Apart from the overhead introduced by occlusion queries, the success of a culling algorithm depends strongly on how tightly the bounding volumes in the spatial hierarchy approximate the contained geometry. If the fit is not tight enough, many nodes will be classified as visible even though the contained geometry is not. There are several techniques for obtaining tight bounding volumes, mostly by replacing axis-aligned bounding boxes by more complex shapes. While these methods could directly be applied to most occlusion culling algorithms, they also constitute an overhead of calculating and maintaining these volumes. This can become costly especially for dynamic scenes.

We propose a simple method for determining tighter bounds for *inner nodes* in the context of hardware occlusion queries applied to an arbitrary bounding volume hierarchy. For a particular node we determine its *tight bounding volume* as a collection of bounding volumes of its children at a particular depth (see Figure 5).

It turns out that when using up-to-date APIs for rendering the bounding volume geometry (e.g., OpenGL vertex buffer objects), a slightly more complex geometry for the occlusion query practically does not increase its overhead. However, there might be a penalty for rasterizing the tight bounding volumes when some of the smaller bounding primitives overlap in screen space, thus increasing the fill rate compared to projecting the original bounding volume of the node. To avoid such a case, we use a simple test to ensure the usefulness of the tighter bounds. When collecting the child nodes for the tight bounding volume, we test if the sum of surface areas of the bounding volumes of the children is not larger than s_{max} times the surface area of the parent node (note that this does not depend on a particular view point). If this is the case, we terminate traversal and do not further refine the bounding representation. We terminate the search for bounding volumes if the depth from the node is greater than a specified maximal depth d_{max}. The following values gave good results in our tests: $d_{max} = 3$, $s_{max} = 1.4$.

Note that it is advantageous to determine the tight bounding volumes also for *leaves* of the hierarchy. This can be easily achieved by building a slightly deeper hierarchy and then marking interior nodes of the hierarchy containing less than a specified number of triangles as *virtual leaves*, i.e., interior nodes that are considered as leaves during traversal.

As a result, tight bounding volumes provide several benefits at almost no cost: (1) earlier culling of interior nodes of the hierarchy, (2) culling of leaves which would otherwise be classified as visible, (3) increase of coherence of visibility classification of interior nodes. The first property leads to a reduction of the number of queries. The second property provides a reduction of the number of rendered triangles. Finally, the third benefit avoids changes in visibility classification for interior nodes caused by repeated pull-up and pull-down of visibility.

7. Putting it all together

The CHC++ algorithm aims to keep the simplicity of the CHC algorithm, with several important add-ons. In this section we summarize the complete CHC++ algorithm and emphasize its main differences from the CHC algorithm. The pseudocode of the CHC++ algorithm is shown in Figure 6.

As in CHC, we use a priority queue for traversing the hierarchy. This queue provides a front-to-back order of the processed nodes. Unlike CHC, the new algorithm uses two new queues for storing nodes which should be queried (v-queue and i-queue). These two queues are the key for reduction of rendering state changes and compiling multiqueries.

The previously visible nodes are rendered immediately as for CHC. If they are scheduled for testing in the current frame, they are placed in the v-queue. The algorithm for scheduling the queries uses the discussed temporally jittered sampling pattern to reduce the number of queries and to distribute them evenly over frames. The queries for nodes stored in the v-queue are used to fill up the wait time if it should occur. At the end of the frame the remaining nodes in the v-queue form a single batch of queries.

The i-queue accumulates processed nodes which have been invisible in the previous frames. When there is a sufficient number of nodes in the queue, we apply a batch of occlusion queries for nodes in the i-queue while compiling them into the multiqueries.

When integrating the method into a game engine the vis-

```
   CHC++ begin
       DistanceQueue.push(Root);
       while !DistanceQueue.Empty() || !QueryQueue.Empty() do
           while !QueryQueue.Empty() do
               if FirstQueryFinished then
                   N = QueryQueue.Dequeue();
                   HandleReturnedQuery(N);
++             else
++                 // next prev. vis. node query;
++                 IssueQuery(v-queue.pop());
           if !DistanceQueue.Empty() then
               N = DistanceQueue.DeQueue();
               if InsideViewFrustum(N) then
                   if !WasVisible(N) then
                       QueryPreviouslyInvisibleNode(N);
                   else
                       if N.IsLeaf && QueryReasonable(N)
                       then
++                         v-queue.push(N);
                       TraverseNode(N);
++         if DistanceQueue.Empty() then
++             // issue remaining query batch;
++             IssueMultiQueries();
++     while !v-queue.empty() do
++         // remaining prev. visible node queries;
++         IssueQuery(v-queue.pop());
   end
```

```
   TraverseNode(N) begin
       if IsLeaf(N) then
           Render(N);
       else
           DistanceQueue.PushChildren(N);
           N.IsVisible = false;
   end
   PullUpVisibility(N) begin
       while !N.IsVisible do
           N.IsVisible = true; N = N.Parent;
   end
   HandleReturnedQuery(Q) begin
       if $Q.visiblePixels > threshold$ then
++         if $Q.size() > 1$ then
++             QueryInvididualNodes(Q); // failed multiquery
           else
               if !WasVisible(N) then
                   TraverseNode(N);
               PullUpVisibility(N);
       else
           N.IsVisible = false;
   end
++ QueryPreviouslyInvisibleNode(N) begin
++     i-queue.push(N) ;
++     if $i\text{-}queue.size() \geq maxPrevInvisNodesBatchSize$ then
++         IssueMultiQueries(); // issue the query batch
++
   end
++ IssueMultiQueries() begin
++     while !i-queue.Empty() do
++         MQ = i-queue.GetNextMultiQuery();
++         IssueQuery(MQ); i-queue.PopNodes(MQ);
++
++
   end
```

Figure 6: *Pseudo-code of the CHC++ main traversal loop and selected important functions. The differences to the original CHC are marked in blue.*

ible nodes are first accumulated in a render queue. The render queue is then processed by the engine before a batch of queries from i-queue is about to be issued.

8. Results

For all our results we used an Intel Quad Core 2.66 MHz CPU and an NVidia 8800 GTX graphics card. We tested our method on three different scenes: Vienna, a typical city scene with detailed street objects and trees (2,583,674 triangles and 10,535 BVH nodes); Pompeii, a generated city scene with detailed buildings (5,646,041 triangles and 22,468 BVH nodes), and the Powerplant model (12,748,510 triangles and 17,793 BVH nodes). In all plots we consistently use the following abbreviations: VFC for View-Frustum Culling, CHC for Coherent Hierarchical Culling, NOHC for Near Optimal Hierarchical Culling, and CHC++ for our new method. For all our measurements of CHC++ we used the following parameters: assumed visible frames $n_{av} = 10$, batch size $b = 50$, maximal depth for tighter bounds $d_{max} = 3$, and the maximal surface area increase for tighter bounds $s_{max} = 1.4$. Note that all walkthroughs shown here are included in the accompanying videos.

Figure 1, shown in the beginning of the paper, presents a frame time comparison for a walkthrough in the Powerplant. It can be seen that the CHC algorithm performs worse than view-frustum culling for some parts of the walkthrough. While NOHC is at least not worse than view-frustum culling, our algorithm performs up to two times better than NOHC.

Figure 7 shows the frame times in a walkthrough in Pompeii and studies the behavior with respect to NOHC and two artificial reference algorithms (NOHC-OPT and OPT). The NOHC-OPT method refers to the function defined as optimum by Guthe et al. [GBK06], which issues queries for all invisible nodes but only if they are feasible according to their cost model. The OPT method refers to an hypothetical algorithm that will only render the visible nodes of the hierarchy without issuing any query. OPT therefore does not depend on the cost or implementation of occlusion queries at all and is the fastest solution that can be achieved with a given hierarchy. We implemented the OPT method by recording the vis-

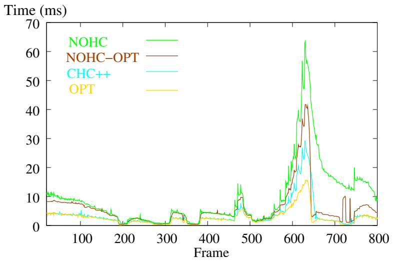

Figure 7: *Frame time comparison of NOHC, the optimal algorithm as defined by Guthe et al. (OPT-NOHC), CHC++, and an algorithm that renders only visible nodes without querying (OPT) in the Pompeii scene.*

ibility results using the exact stop-and-wait algorithm which does not make use of temporal coherence.

As claimed by the authors, NOHC is very close to NOHC-OPT algorithm, except for difficult view points with a lot of visible geometry. More notably, CHC++ is clearly significantly faster than NOHC-OPT practically everywhere. Furthermore, CHC++ is approaching the OPT curve for the moderately complex parts of the scene, which is remarkable since OPT cannot be beaten by any algorithm using occlusion queries on the given hierarchy.

There is still some noticeable overhead of CHC++ compared to OPT in the high frame time parts of the walkthrough, which correspond to views from over the houses where a lot of the scene becomes visible and we have to issue many queries to capture the change in visibility. The rest of the time difference is caused by an accumulation of minor things, like the overhead for maintaining all the queues.

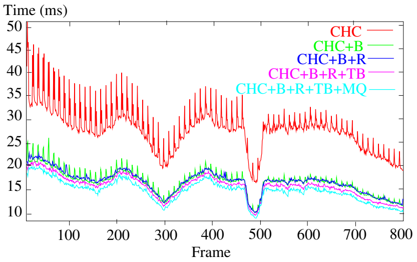

Figure 8: *The benefit of different optimizations in a walkthrough of the Powerplant. We start with CHC and add one optimization at a time. The bottom curve with all optimizations corresponds to CHC++. The abbreviations have the following meaning: B = batching of previously visible and invisible nodes, R = randomization, TB = tighter bounds, MQ = multiqueries.*

Figure 8 shows the benefit of each optimization in another walkthrough in the Powerplant. It is clearly visible that the batching brings the majority of the benefit. Query batching already removes a lot of the query overhead, otherwise the benefit of some of the other optimizations would be much more prominent. The randomization is most important in situations when many nodes become visible at once, which is well visible in the beginning of the walkthrough. The benefit of multiqueries depends on the absolute number of previously invisible nodes, which in turn depends on the properties of the hierarchy (a deeper hierarchy would mean more benefit from multiqueries). Note that the relative benefits of the different optimizations can change for different hardware architectures and rendering engines.

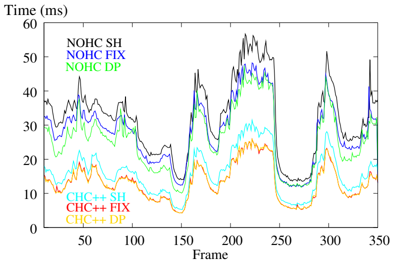

Figure 9: *Dependence of the frame time on shader complexity in a walkthrough in the Powerplant. DP refers to depth only pass, FIX to fixed pipeline shading only, SH to a shader of moderately high complexity.*

In Figure 9 we study how CHC++ and NOHC behave on a walkthrough in the Powerplant with respect to *shader complexity*. We made three different tests: In the first test we used a depth only pass (DP), in the second test we used the standard fixed pipeline material shading of the original Powerplant model (FIX). In the third test we applied a moderately complex shader to all renderable geometry (i.e., the shader has 40 texture lookups).

Note that the used walkthrough is challenging for methods that exploit coherence because it has many swift changes in visibility. As can be seen, the dependence on the shader complexity is very low for CHC++. NOHC shows a much stronger dependence, performing visibly better for the depth pass than for the shaded geometry. Still the depth pass is much slower than for CHC++. Obviously the state changes lower the performance for the depth-only pass as well, even if it only involves a switch of the depth write flag.

Figure 10 analyzes the behavior of all methods in the Vienna scene, particularly with respect to the number of *queries and state changes*. This figure shows that CHC++ fulfills the claim that it significantly reduces both queries and state changes, and that this also translates into a significant performance advantage over the other algorithms.

9. Conclusions

We proposed several modifications to the CHC algorithm [BWPP04]. These modifications provide a significant

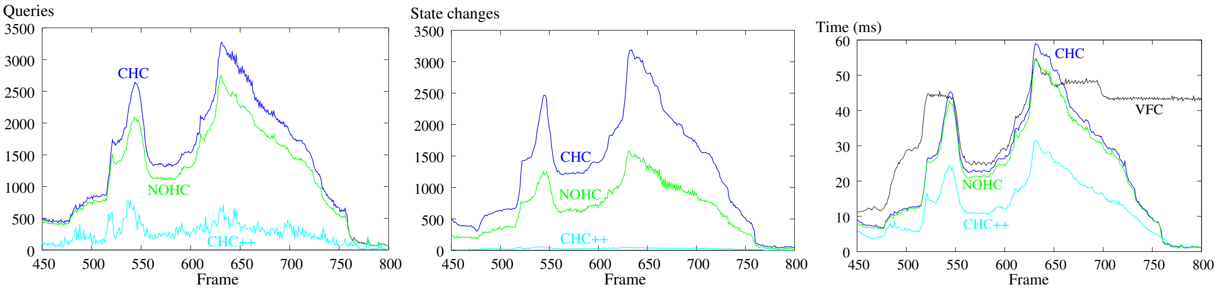

Figure 10: *Comparison of issued queries (left), state changes (middle), and the resulting frame rates for a walkthrough in Vienna. Note that VFC does not impose any additional state changes.*

reduction of state changes, number of queries, rendered triangles, and a further reduction of pipeline stalls. These benefits are achieved by batching of occlusion queries, multi-queries which cover more nodes with a single query, a randomly jittered temporal sampling pattern for queries, and tighter bounding volumes.

The results show that compared to previous methods, the new method provides up to two orders of magnitude reduction in the number of state changes and up to one order of magnitude reduction in the number of queries. These savings translate into a twofold speedup compared to CHC and about 1.5x speedup compared to NOHC [GBK06]. The proposed method is for most cases within a few percent of the "ideal" method which would know visibility classification in advance and render visible geometry only without using any occlusion queries. We believe that the new algorithm will become useful for game programmers as it is stable, easy to implement, and it integrates well with game engines.

In the future we want to study the possibility of automatic parameter adaptation during the walkthrough by exploiting the dependence of the total frame time on the number of issued queries and rendered triangles.

Acknowledgments

This work has been supported by the Ministry of Education, Youth and Sports of the Czech Republic under the research program LC-06008 (Center for Computer Graphics), the Aktion Kontakt grant no. 48p11, and the EU under the FP6 project no. IST-014891-2 (Crossmod)

References

[AM04] AILA T., MIETTINEN V.: dpvs: An occlusion culling system for massive dynamic environments. *IEEE Comput. Graph. Appl. 24*, 2 (2004), 86–97.

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

[BWPP04] BITTNER J., WIMMER M., PIRINGER H., PURGATHOFER W.: Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum 23*, 3 (Sept. 2004), 615–624. Proceedings EUROGRAPHICS 2004.

[COCSD02] COHEN-OR D., CHRYSANTHOU Y., SILVA C., DURAND F.: A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics.* (2002).

[GBK06] GUTHE M., BALÁZS A., KLEIN R.: Near optimal hierarchical culling: Performance driven use of hardware occlusion queries. In *Eurographics Symposium on Rendering 2006* (June 2006), Akenine-Möller T., Heidrich W., (Eds.), The Eurographics Association.

[GSYM03] GOVINDARAJU N. K., SUD A., YOON S.-E., MANOCHA D.: Interactive visibility culling in complex environments using occlusion-switches. In *SI3D* (2003), pp. 103–112.

[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 - Chapel Hill, Sept. 12 2002.

[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 (Oct. 2001), 365–379.

[KS05] KOVALCIK V., SOCHOR J.: Occlusion culling with statistically optimized occlusion queries. In *WSCG (Short Papers)* (2005), pp. 109–112.

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *Visual Computer 6*, 6 (1990), 153–65. criteria for building octree (actually BSP) efficiency structures.

[MBM*01] MEISSNER M., BARTZ D., MUELLER G., HUETTNER T., EINIGHAMMER J.: Generation of decomposition hierarchies for efficient occlusion culling of large polygonal models. In *Proceedings of the Vision Modeling and Visualization Conference 2001* (Nov. 21–23 2001), pp. 225–232.

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

[ZMHH97] ZHANG H., MANOCHA D., HUDSON T., HOFF III K. E.: Visibility culling using hierarchical occlusion maps. In *SIGGRAPH 97 Conference Proceedings* (Aug. 1997), Whitted T., (Ed.), Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 77–88. ISBN 0-89791-896-7.