8.4 Game-Engine-Friendly Occlusion Culling

Jiri Bittner, bittner@fel.cvut.cz,
Czech Technical University in Prague

Oliver Mattausch, matt@cg.tuwien.ac.at,
Vienna University of Technology

Michael Wimmer, wimmer@cg.tuwien.ac.at,
Vienna University of Technology

Introduction

Occlusion culling is an important technique to reduce the time for rendering complex scenes [1]. It saves CPU time, geometry processing time, and fragment processing time for objects that are occluded by other parts of scenes. The availability of so-called hardware occlusion queries has made occlusion culling easily available on commodity PCs [2,3,5]. Such a query returns the number of fragments that would be rasterized if an object (usually a bounding volume of a complex object) were rendered. Occlusion queries lead to significant speedups of rendering if many primitives are occluded, but they also come at a cost. Issuing an occlusion query takes some time, and it usually requires a change of rendering state. If we perform too many queries or there isn't a lot of occlusion in the scene, the overall rendering performance might even drop compared to pure view frustum culling. Another problem preventing the widespread usage of occlusion queries is the difficulty of integrating them into already optimized rendering loops of game engines.

This article presents a method that minimizes the overhead associated with occlusion queries. The method reduces the number of required state changes and should integrate easily with most game engines. The key ideas are batching of the queries and interfacing with the game engine using a dedicated render queue. We also present some additional optimizations that reduce the number of queries issued as well as the number of rendered primitives. The algorithm is based on the well-known coherent hierarchical culling algorithm, which we will briefly recap next.

Coherent Hierarchical Culling

The coherent hierarchical culling (CHC) algorithm proposed by Wimmer and Bittner [5] aims at good utilization of the GPU by clever interleaving of occlusion queries and rendering. Note that CHC works on a spatial hierarchy such as bounding volume hierarchies (BVHs), kD-trees, or octrees. Using a spatial hierarchy has been shown to be a key for achieving reasonable gain of occlusion culling on complex scenes. CHC works well in many cases, but it also has several problems that we shall address in this article. First, let's recap how the CHC algorithm works.

The CHC Algorithm

The CHC algorithm traverses a given hierarchy in a front-to-back order and issues queries only for previously visible leaf nodes of the hierarchy 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.

In more detail the algorithm works as follows: For previously invisible nodes it issues an occlusion query, which is then also stored in the *query queue*. For previously visible interior nodes it immediately recurses to its children. For previously visible leaves it issues an occlusion query and renders the associated geometry without waiting for the query result.

After each visited node the algorithm checks the front of the query queue. If the result of a query is available, the algorithm proceeds as follows: If the query result does not indicate a change in visibility, no additional work is required. Otherwise, if a previously invisible interior node becomes visible, its children are processed by putting them into the traversal queue (marked as previously invisible nodes). If a previously invisible leaf becomes visible, the associated geometry is rendered. If a previously visible leaf is found invisible, it is only marked as invisible (this change in visibility will be reflected in the next frame). The changes in visibility are propagated in the hierarchy by pushing visibility status up or down (see Figure 8.4.1). The pseudo code of the CHC algorithm is shown in Listing 8.4.1.

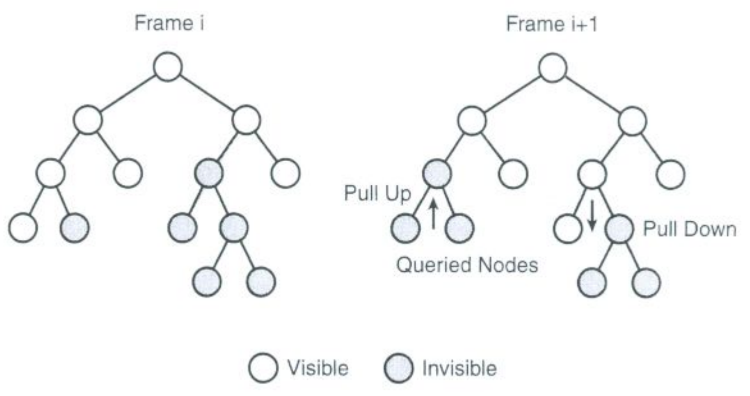

FIGURE 8.4.1 Visibility of hierarchy nodes determined by the CHC algorithm in two consecutive frames.

Listing 8.4.1 Pseudo code of the CHC algorithm

```
CHC begin
DistanceQueue.push(Root);
while !DistanceQueue.Empty() || !QueryQueue.Empty() do
    while !QueryQueue.Empty() &&
        (DistanceQueue.Empty() || FirstQueryFinished) do
            while !FirstQueryFinished then wait;
            N = QueryQueue.Dequeue();
            HandleReturnedQuery(N);

    if !DistanceQueue.Empty() then
        N = DistanceQueue.DeQueue();
        N.IsVisible = false; // set invisible by default
        if InsideViewFrustum(N) then
            if !WasVisible(N) then
               QueryNode(N);  // query previously invisible node
        else
            if N.IsLeaf then
               QueryNode(N); // query only prev. visible leaves
            TraverseNode(N);  // traverse previously visible node
End
```

```
HandleReturnedQuery(Q) begin
if Q.visiblePixels > threshold then
      if !WasVisible(N) then               // traverse previously
invisible node
            TraverseNode(N);               // which turned visible
      PullUpVisibility(N);                 // mark the node as visible
End

TraverseNode(N) begin
if IsLeaf(N) then
      Render(N);
else
      DistanceQueue.PushChildren(N);
End

PullUpVisibility(N) begin
while !N.IsVisible do                // mark node as visible and propagate
      N.IsVisible = true;            // this to its parents
      N = N.Parent;
End
```

Problems with CHC

The reduction of the number of queries (queries are not issued on previously visible interior nodes) and clever interleaving work very well for scenarios that have a lot of occlusion. However, for view points where much of the scene is visible, the method can become even slower than conventional view frustum culling, which is a result of numerous state changes and wasted queries. This problem is more pronounced on newer hardware, where rendering geometry becomes quite cheap compared to querying.

Another problem with 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 leads to a high number of engine API calls. Additionally, unless a dedicated depth-only pass is used, the method does not allow the engine to perform material sorting.

These two problems make the CHC algorithm less attractive for game developers, who call for an algorithm that is reliably faster than view frustum culling and is easy to integrate into the game engine. This article will provide several modifications to the CHC algorithm with the aim of solving its problems and delivering a game-engine-friendly occlusion culling method.

Reducing State Changes

In the CHC algorithm a state change is required for every occlusion query. This state change involves disabling writing to color and depth buffers, which is then reenabled after the query. Also, complex shaders should be disabled for the geometry rendered during the query.

On current GPU architectures state changes are still rather costly operations. It turns out that the state changes can 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 shoot for about 200 state changes per frame as an acceptable value on current hardware [6]. Our first step toward a game-engine-friendly algorithm is thus the reduction of state changes.

Batching Queries

A simple solution to avoid state changes for every occlusion query is to batch the queries instead of issuing them immediately. 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.

How do we batch the queries so that the batching does not harm the final visibility classification of hierarchy nodes? Our proposal is to use two additional queues for scheduling occlusion queries. These queues will be used to accumulate the queries for nodes of different visibility classifications, as we discuss in the next two sections.

Batching Previously Invisible Nodes

The previously invisible nodes to be queried are inserted into a queue that we call i-queue (i stands for previously invisible). 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 a query for each node in the i-queue.

As a result, for the batch of size *b* we perform approximately *b* times less state changes than the CHC algorithm. On the other hand, increasing the batch size delays the availability of the query results. This means that visibility changes could be detected later, and possible follow-up queries might introduce further latency, if there is not enough alternative work left (e.g., rendering visible nodes).

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

Batching Previously Visible Nodes

Recall that the CHC algorithm issues a query for a previously visible node and renders the geometry of the node without waiting for the result of the query. However, the result of the query is not critical for the current frame since it will only be used in the next frame. Therefore, we will not issue the queries immediately, but instead the corresponding nodes will be stored in a queue that we call v-queue. The nodes from the v-queue will then be used to fill up idle 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 practically 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 have been captured only 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. A visualization of state changes required by the described method is depicted in Figure 8.4.2.

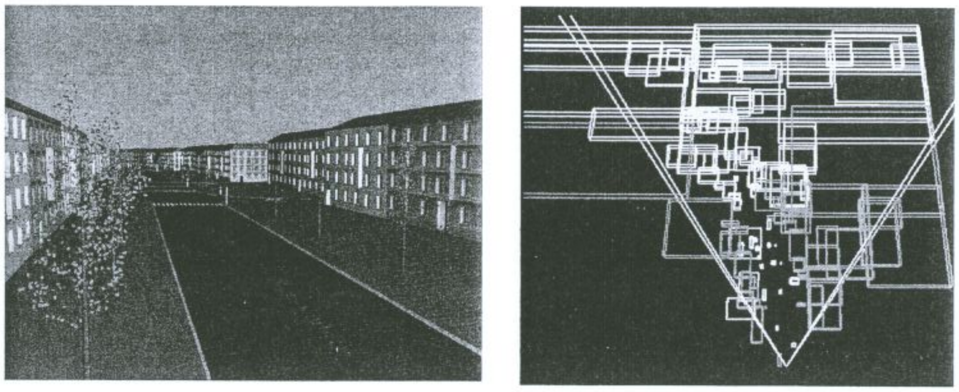
FIGURE 8.4.2 Visualization of state changes. (Left) View of a city scene. (Right) The culling algorithm introduces only two additional state changes (state change is depicted by changing the color of hierarchy nodes).

Game Engine Integration

Integrating occlusion culling into game engines has received very little attention in the literature. Hence, we will have a look at how the described method can be integrated into the engine so that we can efficiently reuse the existing highly optimized rendering loops.

Depth-Only Pass

One possibility of integrating occlusion culling into a game engine is by using a dedicated depth-only pass for determining visibility, followed by shading passes for completing the picture. In the depth-only pass the content of the z-buffer is initialized, using occlusion culling, without writing to the color buffer. For subsequent shading passes we already know the visibility classification of all nodes in the hierarchy. We collect visible nodes and render them in an order that can be optimized by the engine (e.g., sorting by materials). For the shading passes we skip all invisible geometry at no cost. Additionally, invisible fragments of the geometry contained in visible nodes are culled early in the pipeline.

Using a depth-only pass we eliminate the problem of enforced front-to-back ordering due to occlusion culling for the shading passes, as we only have to maintain the front-to-back order for the depth-only pass. Since we do not use any materials or shaders in the depth-only pass, the enforced front-to-back order of geometry does not introduce any additional state changes. However, there might be a problem with engine API overhead if many rendering calls are issued. Additionally, the code for the depth-only pass and the shading passes might be shared in some engines, which might complicate the integration of occlusion culling into one of the passes.

Fortunately, there is a very simple workaround of this problem, which we describe next.

Batching Through a Render Queue

We use the following idea to allow the game engine to perform its internal sorting optimizations: When our culling algorithm is going to render some geometry, we will not render it immediately. Instead we store the nodes to be rendered in a render queue. The render queue will accumulate all nodes scheduled for rendering. Note that many engines already contain a render queue that can be used for this purpose. If the engine API does not allow manipulating the render queue, we just add another queue to the method.

The contents of the render queue will be processed by the engine in a single API call just before a batch of occlusion queries is about to be issued. The engine can then apply its internal material and shader sorting and render the objects stored in the queue in the new order.

Note that the number of objects in the queue passed to the engine depends on the batch size of the i-queue, but it will also change during the frame. Typically, we will have large batches in the beginning of the frame and smaller batches later on. This follows from the fact that in the beginning of the frame we schedule rendering of many nearby visible nodes, while not processing many invisible nodes, which are being accumulated in the i-queue.

As an alternative, we can set a minimal number of primitives for issuing a render call. In this case the rendering API can be controlled independently from the query batch size. However, this approach can increase the number of rendered triangles, as some occlusion can be missed due to delayed rendering.

The overview of the different queues discussed so far is shown in Figure 8.4.3.

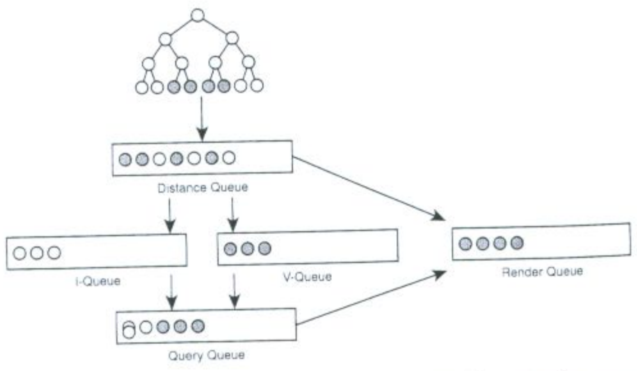

FIGURE 8.4.3 Different queues used by the algorithm. Darker gray nodes correspond to previously invisible nodes, and lighter gray nodes represent previously visible nodes. The overlapping previously invisible nodes in the query queue correspond to a multiquery that will be discussed later in the article.

Skipping Tests for Visible Nodes

The CHC algorithm introduced an important optimization that reduces the number of queries on previously visible leaves. With this optimization a visible leaf is assumed to stay visible for n_{av} frames, and it will only be tested in the frame $n_{av} + 1$. As a result, the average number of queries for previously visible leaves is reduced by a factor of $n_{av} + 1$.

This simple method, however, has a problem that the queries get temporally aligned. The query alignment becomes problematic in situations when many nodes 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. Many nodes become visible at once, and the queries of those nodes will be issued and then scheduled for the n_{av} + 1th frame. Thus, most of the queries will be aligned again. The average number of queries per frame will be reduced, but the alignment can cause observable frame rate drops.

The first solution that comes to mind is a randomization of n_{av} by a small random value. However, this does not solve the problem in a satisfying manner. If the randomization is small, the queries might still be very much aligned. On the other hand, if the randomization is large, some of the queries will be processed too late, and thus the change from visible to invisible state will be captured too late.

We propose a different solution. We will randomize only the first invocation of the occlusion query and then use regular sampling. 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 would use a regular sampling interval given by n_{av} (see Figure 8.4.4).

The optimal value of n_{av} depends on the scene itself, visibility coherence, and hardware parameters as well as the rendering engine parameters. Fortunately, our results indicate that these dependencies are quite weak, and a value of 5–10 is a safe and robust choice in practice.

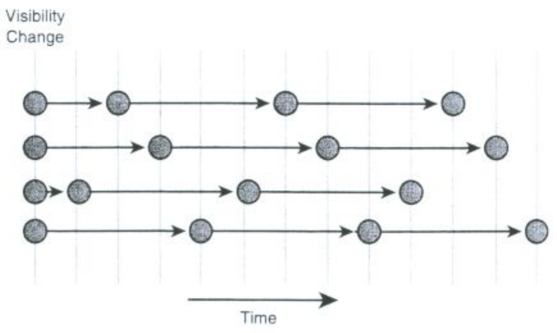

FIGURE 8.4.4 Scheduling of queries of visible nodes using randomization of the first invocation of the query.

Further Optimizations

The previous sections described the core ideas of a game-engine-friendly occlusion culling algorithm. Here we present additional optimizations that further reduce the number of issued queries and the number of rendered primitives. Note that these optimizations are not critical for the game engine integration, but they further boost the performance by an additional 5–20%.

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 of 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. These methods constitute an overhead of calculating and maintaining the bounding volumes, which can become costly, especially for dynamic scenes. Is there a solution that could provide tighter bounding volumes without the need for calculating more complex bounding shapes? The answer is yes, and it follows from the properties of current rendering architectures.

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. A simple solution to our problem is thus to replace a single large bounding volume by several smaller ones. In the case of an internal node of the hierarchy, the tighter bounding volume can be obtained by collecting bounding volumes of its children at a particular depth (see Figure 8.4.5). For leaf nodes the tighter bounding volumes have to be constructed explicitly. Alternatively, we can construct a slightly deeper hierarchy and then mark interior nodes of the hierarchy containing less than a specified number of triangles as virtual leaves, that is, nodes that are considered as leaves during traversal. In this case the method of gathering child nodes can be used to establish a collection of tight bounding volumes also for the virtual leaves.

Increasing the number of bounding volumes provides a tighter fit to the geometry, but using too many small bounding volumes might be counterproductive due to either increased fill rate or transform rate. Therefore, when collecting the child nodes for the tight bounding volume, we limit the search to a specified maximal depth d_{max} from the node (transform rate constraint). Also, 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 (fill rate constraint). The following values gave good results in our tests: $d_{max} = 3$, $s_{max} = 1.4$.

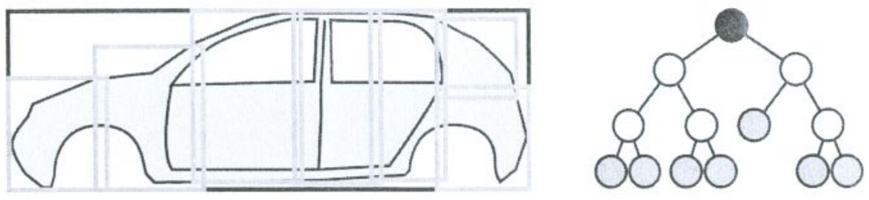

FIGURE 8.4.5 For a given node (in black) the tighter bounding volumes (in gray) are obtained by gathering several child nodes in the hierarchy.

Tight bounding volumes provide several benefits at almost no cost: (1) culling of leaves that would otherwise be classified as visible, which reduces the number of rendered primitives, (2) earlier culling of interior nodes of the hierarchy, which reduces the number of queries, (3) increased coherence of visibility classification of interior nodes, which avoids changes in visibility classification for interior nodes caused by repeated pull-up and pull-down of visibility.

MULTIQUERIES

Common occlusion culling techniques use one occlusion query per invisible primitive. However, if some invisible nodes remain invisible, a single occlusion query for all these nodes would be sufficient to verify their visibility status. Such a query would render all bounding boxes of the nodes, and return zero if all nodes remain occluded.

Assuming a certain coherence of visibility, we can group 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 nodes in the group minus 1. However, in the second case the multiquery for the batch was wasted, and we proceed by individual queries on the nodes.

To find suitable node groupings that minimize the effect of wasted batches, we use an adaptive mechanism based on a cost-benefit heuristics. Before we describe the actual heuristics, we first quantize the coherence of visibility in the scene, which will then be used as a major factor driving the cost model.

Estimating Visibility Coherence

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. There is a strong correlation of this value with the "history" of the node, that is, 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 the car engine. On the contrary, even in slow-moving scenarios, there are always some nodes on the visible border that 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 queries:

$$P_{keep}(i) \approx \frac{n_i^{keep}}{n_i^{all}}$$

where n_i^{keep} is the number of already tested nodes that have been in the same state for i frames and keep their state in the I + 1th frame, and n_i^{all} is the total number of already tested nodes that have been in the same state for i frames.

The values n_i^{keep} and n_i^{all} are tabulated and constantly updated during the walk-through. In the first few frames there are not enough measurements for an accurate computation of $p_{keep}(i)$, especially for higher values of i. We solve this problem by piecewise constant propagation of the already computed values to the higher values of i.

As an alternative to the measured function, we suggest using an analytic formula that fits reasonably well with the measurements we did on several test scenes using typical navigation sequences:

$$P_{keep}(i) \approx 0.99 - 0.7e^{-1}$$

Cost-Benefit Heuristics for Multiqueries

To compile multiqueries we use a greedy algorithm that maximizes a benefit-cost ratio.

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

$$C(M) = 1 + p_{fail}(M) \cdot |M|,$$

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:

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

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

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, that is, $p_{keep}(i_N)$. 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 $V(Mj) = B(Mj)/C(Mj)$. 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 a high probability of staying invisible and small multiqueries for nodes that are likely to turn visible. An example of compiled multiqueries is depicted in Figure 8.4.6.

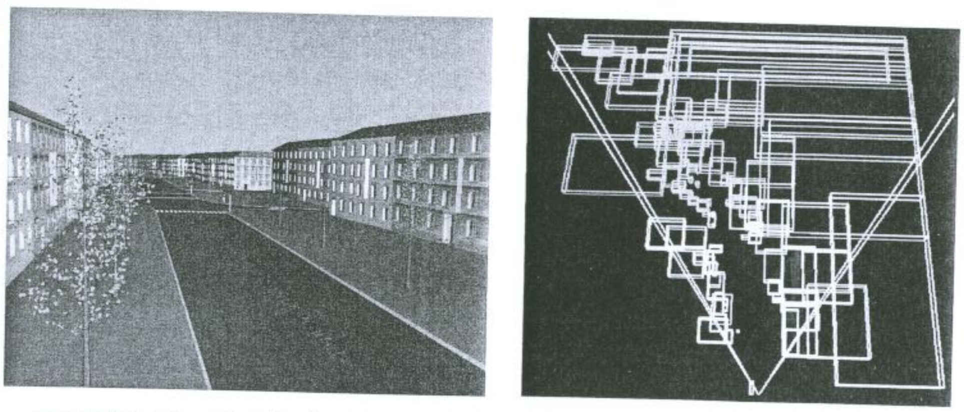

FIGURE 8.4.6 Visualization of multiqueries. (Left) A view of the city scene. (Right) (In)visibility of previously invisible nodes is successfully verified by only two multiqueries.

Putting It All Together

When we combine all modifications to the CHC algorithm described in this article, we end up with a method that we call CHC++ [3]. CHC++ keeps the simplicity of the previous technique, but has several properties that make it more interesting for use in a game engine.

Let us summarize once more the main ideas of CHC++. The algorithm uses two new queues for scheduling queries (v-queue and i-queue). These two queues are the key for reduction of state changes. The i-queue accumulates processed nodes that have been invisible in the previous frames. When there are a sufficient number of nodes in the queue, we apply a batch of occlusion queries for nodes in the i-queue. Visible nodes scheduled for testing in the current frame are placed in the v-queue. The queries for nodes stored in the v-queue are used to fill up the idle time if it should occur. At the end of the frame the remaining nodes in the v-queue form a single batch of queries.

Visible geometry that is about to be rendered is accumulated in the render queue. The render queue is then processed by the rendering engine just before a batch of queries from the i-queue is about to be issued. The algorithm for scheduling the queries on previously visible nodes uses a temporally jittered sampling pattern to reduce the number of queries and to distribute them evenly over frames (Listing 8.4.2).

Listing 8.4.2 The pseudo code of the CHC++ algorithm

```
CHC++ begin
Collect

DistanceQueue.push(Root);
while !DistanceQueue.Empty() || !QueryQueue.Empty() do
    while !QueryQueue.Empty() &&
      (DistanceQueue.Empty() || FirstQueryFinished) do
            while !FirstQueryFinished && !v-queue.Empty() then
                IssueQuery(v-queue.Dequeue()); // fill-up wait time
            N = QueryQueue.Dequeue();
            HandleReturnedQuery(N);
        else
```

in this article,
e simplicity of
interesting for

lgorithm uses
se two queues
ites processed
ire a sufficient
s for nodes in
e are placed in
fill up the idle
in the v-queue

in the render
before a batch
scheduling the
ing pattern to
rames (Listing

```
then
wait time
```

```
        if !DistanceQueue.Empty() then
                N = DistanceQueue.DeQueue();
                N.IsVisible = false;                  // invisible by default
                if InsideViewFrustum(N) then
                    if !WasVisible(N) then
                        QueryPreviouslyInvisibleNode(N);
                    else
                        if N.IsLeaf
                            if QueryReasonable(N) then
                                v-queue.Push(N);
                            else
                                PullUpVisibility(N);
                        TraverseNode(N);
        if DistanceQueue.Empty() then              // no nodes to traverse
            IssueMultiQueries();                   // issue multiqueries if any

while !v-queue.empty() do                          // issue batch of remaining
v-queries
        IssueQuery(v-queue.Dequeue());

.. .. // possible to do some other work

while !QueryQueue.Empty()                          // handle remaining v-queries
        N = QueryQueue.Dequeue();
        HandleReturnedQuery(N);

End CHC++


TraverseNode(N) begin
if IsLeaf(N) then
        Render(N);
else
```

```
        DistanceQueue.PushChildren(N);
        N.IsVisible = false;
End TraverseNode

PullUpVisibility(N) begin
while !N.IsVisible do
        N.IsVisible = true; N = N.Parent;
End PullUpVisibility

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);
End HandleReturnedQuery

QueryPreviouslyInvisibleNode(N) begin
i-queue.Push(N) ;
if i-queue.Size() >= b then
        IssueMultiQueries();
End QueryPreviouslyInvisibleNode



IssueMultiQueries() begin
while !i-queue.Empty() do
        MQ = i-queue.GetNextMultiQuery();
        IssueQuery(MQ); i-queue.PopNodes(MQ);
End IssueMultiQueries
```

CONCLUS

Tl
de
cu
op
ri
al
cu

th
of
in
th
ua
lo
ne
ra

ACKNOW

Th
th
Gr
01

REFERENC

[1]
app
[2]
har
[3]
Co
[4]
[5]
Ad
[6]
Co

Conclusion

This article addressed issues of integrating occlusion culling into a game engine. We described several extensions to the previously published coherent hierarchical culling method in order to improve its efficiency and make its integration into optimized rendering loops of game engines easier. The core of the proposed algorithm remains simple and should be easy to implement in various frameworks. We also proposed several additional optimizations that provide a further increase of culling efficiency with reasonable implementation effort.

The described method provides more than an order of magnitude reduction of the number of state changes as well as the number of engine API calls. The number of queries is also significantly reduced. These savings should provide significant increases of frame rate; the actual speedup is largely dependent on the type of scene, the engine architecture, and the hardware used. The method copes well with the situation when the view point moves from a highly occluded region into a region with low occlusion when much of the scene becomes visible. In the scenes we tested, the new algorithm is typically between 1.5 and 3 times faster than CHC, while frame rates never dropped below standard view frustum culling.

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 grant no. 48p11, and the EU under the FP6 project no. IST-014891-2 (Crossmod).

References

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

[2] 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).

[3] MATTAUSCH O., BITTNER J., WIMMER M.: CHC++: Coherent Hierarchical Culling Revisited. Computer Graphics Forum, Proceedings of EUROGRAPHICS 2008.

[4] SEKULIC, D.: Efficient Occlusion Culling. *GPU Gems*, pp. 487–503. Addison-Wesley (2004).

[5] WIMMER M., BITTNER J.: Hardware occlusion queries made useful. *GPU Gems 2*, pp. 91–108. Addison-Wesley (2005).

[6] WLOKA, M.: Batch, Batch, Batch: What Does It Really Mean? Presentation at Game Developers Conference 2003.