EUROGRAPHICS 2004 / M.-P. Cani and M. Slater
(Guest Editors)

Volume 23 (2004), Number 3

Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful

Jiří Bittner[1], Michael Wimmer[1], Harald Piringer[1,2] and Werner Purgathofer[1]

[1]Institute of Computer Graphics and Algorithms, Vienna University of Technology
[2]VRVis Vienna

Abstract

We present a simple but powerful algorithm for optimizing the usage of hardware occlusion queries in arbitrary complex scenes. Our method minimizes the number of issued queries and reduces the delays due to the latency of query results. We reuse the results of occlusion queries from the last frame in order to initiate and schedule the queries in the next frame. This is done by processing nodes of a spatial hierarchy in a front-to-back order and interleaving occlusion queries with rendering of certain previously visible nodes. The proposed scheduling of the queries makes use of spatial and temporal coherence of visibility. Despite its simplicity, the algorithm achieves good culling efficiency for scenes of various types. 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]: Three-Dimensional Graphics and Realism

1. Introduction

Visibility culling is one of the major acceleration techniques for the real-time rendering of complex scenes. The ultimate goal of visibility culling techniques is to prevent invisible objects from being sent to the rendering pipeline. A standard visibility-culling technique is *view-frustum culling*, which eliminates objects outside of the current view frustum. View-frustum culling is a fast and simple technique, but it does not eliminate objects in the view frustum that are occluded by other objects. This can lead to significant *overdraw*, i.e., the same image area gets covered more than once. The overdraw causes a waste of computational effort both in the pixel and the vertex processing stages of modern graphic hardware. The elimination of occluded objects is addressed by *occlusion culling*. In an optimized rendering pipeline, occlusion culling complements other rendering acceleration techniques such as levels of detail or impostors.

Occlusion culling can either be applied offline or online. When applied offline as a preprocess, we compute a potentially visible set (PVS) for cells of a fixed subdivision of the scene. At runtime, we can quickly identify a PVS for the given viewpoint. However, this approach suffers from four major problems: (1) the PVS is valid only for the original static scene configuration, (2) for a given viewpoint, the corresponding cell-based PVS can be overly conservative, (3) computing all PVSs is computationally expensive, and (4) an accurate PVS computation is difficult to implement for general scenes. Online occlusion culling can solve these problems at the cost of applying extra computations at each frame. To make these additional computations efficient, most online occlusion culling methods rely on a number of assumptions about the scene structure and its occlusion characteristics (e.g. presence of large occluders, occluder connectivity, occlusion by few closest depth layers).

Recent graphics hardware natively supports an *occlusion query* to detect the visibility of an object against the current contents of the z-buffer. Although the query itself is processed quickly using the raw power of the graphics processing unit (GPU), its result is not available immediately due to the delay between issuing the query and its actual

processing in the graphics pipeline. As a result, a naive application of occlusion queries can even decrease the overall application performance due the associated CPU stalls and GPU starvation. In this paper, we present an algorithm that aims to overcome these problems by reducing the number of issued queries and eliminating the CPU stalls and GPU starvation. To schedule the queries, the algorithm makes use of both the spatial and the temporal coherence of visibility. A major strength of our technique is its simplicity and versatility: the method can be easily integrated in existing real-time rendering packages on architectures supporting the underlying occlusion query.

2. Related Work

With the demand for rendering scenes of ever increasing size, 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, we distinguish between *from-point* and *from-region* visibility algorithms. From-region algorithms compute a PVS and are applied offline in a preprocessing phase [ARB90, TS91, LSCO03]. From-point algorithms are applied online for each particular viewpoint [GKM93, HMC*97, ZMHH97, BHS98, WS99, KS01]. In our further discussion we focus on online occlusion culling methods that exploit graphics hardware.

A conceptually important online occlusion culling method is the hierarchical z-buffer introduced by Greene et al. [GKM93]. It organizes the z-buffer as a pyramid, where the standard z-buffer is the finest level. At all other levels, each z-value is the farthest in the window corresponding to the adjacent finer level. The hierarchical z-buffer allows to quickly determine if the geometry in question is occluded. To a certain extent this idea is used in the current generation of graphics hardware by applying early z-tests of fragments in the graphics pipeline (e.g., Hyper-Z technology of ATI or Z-cull of NVIDIA). However, the geometry still needs to be sent to the GPU, transformed, and coarsely rasterized even if it is later determined invisible.

Zhang [ZMHH97] proposed hierarchical occlusion maps, which do not rely on the hardware support for the z-pyramid, but instead make use of hardware texturing. The hierarchical occlusion map is computed on the GPU by rasterizing and down sampling a given set of occluders. The occlusion map is used for overlap tests whereas the depths are compared using a coarse depth estimation buffer. Wonka and Schmalstieg [WS99] use occluder shadows to compute from-point visibility in $2\frac{1}{2}$D scenes with the help of the GPU. This method has been further extended to online computation of from-region visibility executed on a server [WWS01].

Bartz et al. [BMH98] proposed an OpenGL extension for occlusion queries along with a discussion concerning a potential realization in hardware. A first hardware implementation of occlusion queries came with the VISUALIZE fx graphics hardware [SOG98]. The corresponding OpenGL extension is called HP_occlusion_test. A more recent OpenGL extension, NV_occlusion_query, was introduced by NVIDIA with the GeForce 3 graphics card and it is now also available as an official ARB extension.

Hillesland et al. [HSLM02] have proposed an algorithm which employs the NV_occlusion_query. They subdivide the scene using a uniform grid. Then the cubes are traversed in slabs roughly perpendicular to the viewport. The queries are issued for all cubes of a slab at once, after the visible geometry of this slab has been rendered. The method can also use nested grids: a cell of the grid contains another grid that is traversed if the cell is proven visible. This method however does not exploit temporal and spatial coherence of visibility and it is restricted to regular subdivision data structures. Our new method addresses both these problems and provides natural extensions to balance the accuracy of visibility classification and the associated computational costs.

Recently, Staneker et al. [SBS04] developed a method integrating occlusion culling into the OpenSG scene graph framework. Their technique uses occupancy maps maintained in software to avoid queries on visible scene graph nodes, and temporal coherence to reduce the number of occlusion queries. The drawback of the method is that it performs the queries in a serial fashion and thus it suffers from the CPU stalls and GPU starvation.

On a theoretical level, our paper is related to methods aiming to exploit the temporal coherence of visibility. Greene et al. [GKM93] used the set of visible objects from one frame to initialize the z-pyramid in the next frame in order to reduce the overdraw of the hierarchical z-buffer. The algorithm of Coorg and Teller [CT96] restricts the hierarchical traversal to nodes associated with visual events that were crossed between successive viewpoint positions. Another method of Coorg and Teller [CT97] exploits temporal coherence by caching occlusion relationships. Chrysanthou and Slater have proposed a probabilistic scheme for view-frustum culling [SC97].

The above mentioned methods for exploiting temporal coherence are tightly interwoven with the particular culling algorithm. On the contrary, Bittner et al. [BH01] presented a general acceleration technique for exploiting spatial and temporal coherence in hierarchical visibility algorithms. The central idea, which is also vital for this paper, is to avoid repeated visibility tests of interior nodes of the hierarchy. The problem of direct adoption of this method is that it is designed for the use with instantaneous CPU based occlusion queries, whereas hardware occlusion queries exhibit significant latency. The method presented herein efficiently overcomes the problem of latency while keeping the benefits of

a generality and simplicity of the original hierarchical technique. As a result we obtain a simple and efficient occlusion culling algorithm utilizing hardware occlusion queries.

The rest of the paper is organized as follows: Section 3 discusses hardware supported occlusion queries and a basic application of these queries using a kD-tree. Section 4 presents our new algorithm and Section 5 describes several additional optimizations. Section 6 presents results obtained by experimental evaluation of the method and discusses its behavior. Finally Section 7 concludes the paper.

3. Hardware Occlusion Queries

Hardware occlusion queries follow a simple pattern: To test visibility of an occludee, we send its bounding volume to the GPU. The volume is rasterized and its fragments are compared to the current contents of the z-buffer. The GPU then returns the number of visible fragments. If there is no visible fragment, the occludee is invisible and it need not be rendered.

3.1. Advantages of hardware occlusion queries

There are several advantages of hardware occlusion queries:

- *Generality of occluders.* We can use the original scene geometry as occluders, since the queries use the current contents of the z-buffer.
- *Occluder fusion.* The occluders are merged in the z-buffer, so the queries automatically account for occluder fusion. Additionally this fusion comes for free since we use the intermediate result of the rendering itself.
- *Generality of occludees.* We can use complex occludees. Anything that can be rasterized quickly is suitable.
- *Exploiting the GPU power.* The queries take full advantage of the high fill rates and internal parallelism provided by modern GPUs.
- *Simple use.* Hardware occlusion queries can be easily integrated into a rendering algorithm. They provide a powerful tool to minimize the implementation effort, especially when compared to CPU-based occlusion culling.

3.2. Problems of hardware occlusion queries

Currently there are two main hardware supported variants of occlusion queries: the HP test (HP_occlusion_test) and the more recent NV query (NV_occlusion_query, now also available as ARB_occlusion_query). The most important difference between the HP test and the NV query is that multiple NV queries can be issued before asking for their results, while only one HP test is allowed at a time, which severely limits its possible algorithmic usage. Additionally the NV query returns the number of visible pixels whereas the HP test returns only a binary visibility classification.

The main problem of both the HP test and the NV query is the latency between issuing the query and the availability of the result. The latency occurs due to the delayed processing of the query in a long graphics pipeline, the cost of processing the query itself, and the cost of transferring the result back to the CPU. The latency causes two major problems: CPU stalls and GPU starvation. After issuing the query, the CPU waits for its result and does not feed the GPU with new data. When the result finally becomes available, the GPU pipeline can already be empty. Thus the GPU needs to wait for the CPU to process the result of the query and to feed the GPU with new data.

A major challenge when using hardware occlusion queries is to avoid the CPU stalls by filling the latency time with other tasks, such as rendering visible scene objects or issuing other, independent occlusion queries (see Figure 1)

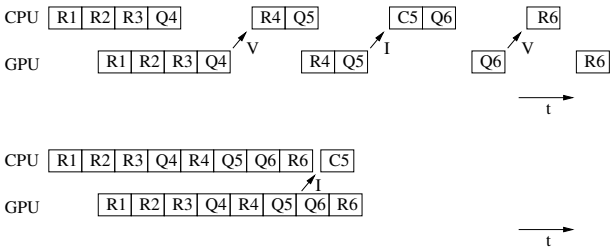

Figure 1: *(top) Illustration of CPU stalls and GPU starvation. Qn, Rn, and Cn denote querying, rendering, and culling of object n, respectively. Note that object 5 is found invisible by Q5 and thus not rendered. (bottom) More efficient query scheduling. The scheduling assumes that objects 4 and 6 will be visible in the current frame and renders them without waiting for the result of the corresponding queries.*

3.3. Hierarchical stop-and-wait method

Many rendering algorithms rely on hierarchical structures in order to deal with complex scenes. In the context of occlusion culling, such a data structure allows to efficiently cull large scene blocks, and thus to exploit spatial coherence of visibility and provide a key to achieving output sensitivity.

This section outlines a naive application of occlusion queries in the scope of a hierarchical algorithm. We refer to this approach as the *hierarchical stop-and-wait* method. Our discussion is based on kD-trees, which proved to be efficient for point location, ray tracing, and visibility culling [MB90, HMC*97, CT97, BH01]. The concept applies to general hierarchical data structures as well, though.

The hierarchical stop-and-wait method proceeds as follows: Once a kD-tree node passes view-frustum culling, it is tested for occlusion by issuing the occlusion query and waiting for its result. If the node is found visible, we continue by recursively testing its children in a front-to-back order. If the node is a leaf, we render its associated objects.

The problem with this approach is that we can continue the tree traversal only when the result of the last occlusion query becomes available. If the result is not available, we have to stall the CPU, which causes significant performance penalties. As we document in Section 6, these penalties together with the overhead of the queries themselves can even decrease the overall application performance compared to pure view-frustum culling. Our new method aims to eliminate this problem by issuing multiple occlusion queries for independent scene parts and exploiting temporal coherence of visibility classifications.

4. Coherent Hierarchical Culling

In this section we first present an overview of our new algorithm. Then we discuss its steps in more detail.

4.1. Algorithm Overview

Our method is based on exploiting temporal coherence of visibility classification. In particular, it is centered on the following three ideas:

- We initiate occlusion queries on nodes of the hierarchy where the traversal terminated in the last frame. Thus we avoid queries on all previously visible interior nodes [BH01].
- We assume that a previously visible leaf node remains visible and render the associated geometry without waiting for the result of the corresponding occlusion query.
- Issued occlusion queries are stored in a query queue until they are known to be carried out by the GPU. This allows interleaving the queries with the rendering of visible geometry.

The algorithm performs a traversal of the hierarchy that is terminated either at leaf nodes or nodes that are classified as invisible. Let us call such nodes the *termination nodes*, and interior nodes that have been classified visible the *opened nodes*. We denote sets of termination and opened nodes in the i-th frame $\mathcal{T}_i$ and $\mathcal{O}_i$, respectively. In the i-th frame, we traverse the kD-tree in a front-to-back order, skip all nodes of $\mathcal{O}_{i-1}$ and apply occlusion queries first on the termination nodes $\mathcal{T}_{i-1}$. When reaching a termination node, the algorithm proceeds as follows:

- For a previously visible node (this must be a leaf), we issue the occlusion query and store it in the query queue. Then we immediately render the associated geometry without waiting for the result of the query.
- For a previously invisible node, we issue the query and store it in the query queue.

When the query queue is not empty, we check if the result of the oldest query in the queue is already available. If the query result is not available, we continue by recursively processing other nodes of the kD-tree as described above. If the query result is available, we fetch the result and remove the node from the query queue. If the node is visible, we process its children recursively. Otherwise, the whole subtree of the node is invisible and thus it is culled.

```
Algorithm: Traversal of the kD-tree
TraversalStack.Push(kDTree.Root);
while ( not TraversalStack.Empty() or
        not QueryQueue.Empty() ) {
  //--- PART 1: processing finished occlusion queries
  while ( not QueryQueue.Empty() and
          (ResultAvailable(QueryQueue.Front()) or
           TraversalStack.Empty()) ) {
    N = QueryQueue.Dequeue();
    // wait if result not available
    visiblePixels = GetOcclussionQueryResult(N);
    if ( visiblePixels > VisibilityThreshold ) {
      PullUpVisibility(N);
      TraverseNode(N);
    }
  }
  //--- PART 2: kd-tree traversal
  if ( not TraversalStack.Empty() ) {
    N = TraversalStack.Pop();
    if ( InsideViewFrustum(N) ) {
      // identify previously visible nodes
      wasVisible = N.visible && (N.lastVisited == frameID -1);
      // identify previously opened nodes
      opened = wasVisible && !IsLeaf(N);
      // reset node's visibility classification
      N.visible = false;
      // update node's visited flag
      N.lastVisited = frameID;
      // skip testing all previously opened nodes
      if ( !opened ) {
        IssueOcclusionQuery(N); QueryQueue.Enqueue(N);
      }
      // traverse a node unless it was invisible
      if ( wasVisible )
        TraverseNode(N);
    }
  }
}
TraverseNode(N) {
  if ( IsLeaf(N) )
    Render(N);
  else
    TraversalStack.PushChildren(N);
}
PullUpVisibility(N) {
  while (!N.visible) { N.visible = true; N = N.parent; }
}
```

Figure 2: *Pseudo-code of coherent hierarchical culling.*

In order to propagate changes in visibility upwards in the hierarchy, the visibility classification is *pulled up* according to the following rule: An interior node is invisible only if all its children have been classified invisible. Otherwise, it remains visible and thus opened. The pseudo-code of the complete algorithm is given in Figure 2. An example of the be-

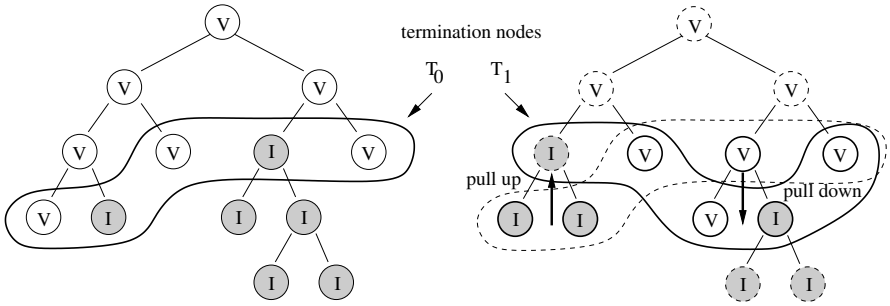

Figure 3: *(left) Visibility classification of a node of the kD-tree and the termination nodes. (right) Visibility classification after the application of the occlusion test and the new set of termination nodes. Nodes on which occlusion queries were applied are depicted with a solid outline. Note the pull-up and pull-down due to visibility changes.*

havior of the method on a small kD-tree for two subsequent frames is depicted Figure 3.

The sets of opened nodes and termination nodes need not be maintained explicitly. Instead, these sets can be easily identified by associating with each node an information about its visibility and an id of the last frame when it was visited. The node is an opened node if it is an interior visible node that was visited in the last frame (line 23 in the pseudocode). Note that in the actual implementation of the pull up we can set all visited nodes to invisible by default and then pull up any changes from invisible to visible (lines 25 and line 12 in Figure 2). This modification eliminates checking children for invisibility during the pull up.

4.2. Reduction of the number of queries

Our method reduces the number of visibility queries in two ways: Firstly, as other hierarchical culling methods we consider only a subtree of the whole hierarchy (opened nodes + termination nodes). Secondly, by avoiding queries on opened nodes we eliminate part of the overhead of identification of this subtree. These reductions reflect the following coherence properties of scene visibility:

- *Spatial coherence.* The invisible termination nodes approximate the occluded part of the scene with the smallest number of nodes with respect to the given hierarchy, i.e., each invisible termination node has a visible parent. This induces an adaptive spatial subdivision that reflects spatial coherence of visibility, more precisely the coherence of occluded regions. The adaptive nature of the subdivision allows to minimize the number of subsequent occlusion queries by applying the queries on the largest spatial regions that are expected to remain occluded.
- *Temporal coherence.* If visibility remains constant the set of termination nodes needs no adaptation. If an occluded node becomes visible we recursively process its children (pull-down). If a visible node becomes occluded we propagate the change higher in the hierarchy (pull-up). A pull-down reflects a spatial growing of visible regions. Similarly, a pull-up reflects a spatial growing of occluded regions.

By avoiding queries on the opened nodes, we can save $1/k$ of the queries for a hierarchy with branching factor k (assuming visibility remains constant). Thus for the kD-tree, up to half of the queries can be saved. The actual savings in the total query time are even larger: the higher we are at the hierarchy, the larger boxes we would have to check for occlusion. Consequently, the higher is the fill rate that would have been required to rasterize the boxes. In particular, assuming that the sum of the screen space projected area for nodes at each level of the kD-tree is equal and the opened nodes form a complete binary subtree of depth d, the fill rate is reduced $(d+2)$ times.

4.3. Reduction of CPU stalls and GPU starvation

The reduction of CPU stalls and GPU starvation is achieved by interleaving occlusion queries with the rendering of visible geometry. The immediate rendering of previously visible termination nodes and the subsequent issuing of occlusion queries eliminates the requirement of waiting for the query result during the processing of the initial depth layers containing previously visible nodes. In an optimal case, new query results become available in between and thus we completely eliminate CPU stalls. In a static scenario, we achieve exactly the same visibility classification as the hierarchical stop-and-wait method.

If the visibility is changing, the situation can be different: if the results of the queries arrive too late, it is possible that we initiated an occlusion query on a previously occluded node A that is in fact occluded by another previously occluded node B that became visible. If B is still in the query queue, we do not capture a possible occlusion of A by B since the geometry associated with B has not yet been rendered. In Section 6 we show that the increase of the number of rendered objects compared to the stop-and-wait method is usually very small.

4.4. Front-to-back scene traversal

For kD-trees the front-to-back scene traversal can be easily implemented using a depth first traversal [BH01]. However, at a modest increase in computational cost we can also use a more general breadth-first traversal based on a priority queue. The priority of the node then corresponds to an inverse of the minimal distance of the viewpoint and the bounding box associated with the given node of the kD-tree [KS01, SBS04].

In the context of our culling algorithm, there are two main advantages of the breadth-first front-to-back traversal :

- *Better query scheduling.* By spreading the traversal of the scene in a breadth-first manner, we process the scene in depth layers. Within each layer, the node processing order is practically independent, which minimizes the problem of occlusion query dependence. The breadth-first traversal interleaves occlusion-independent nodes, which can provide a more accurate visibility classification if visibility changes quickly. In particular, it reduces the problem of false classifications due to missed occlusion by nodes waiting in the query queue (discussed in Section 4.3).
- *Using other spatial data structures.* By using a breadth-first traversal, we are no longer restricted to the kD-tree. Instead we can use an arbitrary spatial data structure such as a bounding volume hierarchy, octree, grid, hierarchical grid, etc. Once we compute a distance from a node to the viewpoint, the node processing order is established by the priority queue.

When using the priority queue, our culling algorithm can also be applied directly to the scene graph hierarchy, thus avoiding the construction of any auxiliary data structure for spatial partitioning. This is especially important for dynamic scenes, in which maintenance of a spatial classification of moving objects can be costly.

4.5. Checking the query result

The presented algorithm repeatedly checks if the result of the occlusion query is available before fetching any node from the traversal stack (line 6 in Figure 2). Our practical experiments have proven that the cost of this check is negligible and thus it can used frequently without any performance penalty. If the cost of this check were significantly higher, we could delay asking for the query result by a time established by empirical measurements for the particular hardware. This delay should also reflect the size of the queried node to match the expected availability of the query result as precise as possible.

5. Further Optimizations

This section discusses a couple of optimizations of our method that can further improve the overall rendering performance. In contrast to the basic algorithm from the previous section, these optimizations rely on some user specified parameters that should be tuned for a particular scene and hardware configuration.

5.1. Conservative visibility testing

The first optimization addresses the reduction of the number of visibility tests at the cost of a possible increase in the number of rendered objects. This optimization is based on the idea of skipping some occlusion tests of visible nodes. We assume that whenever a node becomes visible, it remains visible for a number of frames. Within the given number of frames we avoid issuing occlusion queries and simply assume the node remains visible [BH01].

This technique can significantly reduce the number of visibility tests applied on visible nodes of the hierarchy. Especially in the case of sparsely occluded scenes, there is a large number of visible nodes being tested, which does not provide any benefit since most of them remain visible. On the other hand, we do not immediately capture all changes from visibility to invisibility, and thus we may render objects that have already become invisible from the moment when the last occlusion test was issued.

In the simplest case, the number of frames a node is assumed visible can be a predefined constant. In a more complicated scenario this number should be influenced by the history of the success of occlusion queries and/or the current speed of camera movement.

5.2. Approximate visibility

The algorithm as presented computes a conservative visibility classification with respect to the resolution of the z-buffer. We can easily modify the algorithm to cull nodes more aggressively in cases when a small part of the node is visible. We compare the number of visible pixels returned by the occlusion query with a user specified constant and cull the node if this number drops below this constant.

5.3. Complete elimination of CPU stalls

The basic algorithm eliminates CPU stalls unless the traversal stack is empty. If there is no node to traverse in the traversal stack and the result of the oldest query in the query queue is still not available, it stalls the CPU by waiting for the query result. To completely eliminate the CPU stalls, we can speculatively render some nodes with undecided visibility. In particular, we select a node from the query queue and render the geometry associated with the node (or the whole subtree if it is an interior node). The node is marked as rendered but the associated occlusion query is kept in the queue to fetch its result later. If we are unlucky and the node remains invisible, the effort of rendering the node's geometry is wasted. On the other hand, if the node has become visible, we have used the time slot before the next query arrives in an optimal manner.

To avoid the problem of spending more time on rendering invisible nodes than would be spent by waiting for the result of the query, we select a node with the lowest estimated rendering cost and compare this cost with a user specified constant. If the cost is larger than the constant we conclude that it is too risky to render the node and wait till the result of the query becomes available.

6. Results

We have incorporated our method into an OpenGL-based scene graph library and tested it on three scenes of different types. All tests were conducted on a PC with a 3.2GHz P4, 1GB of memory, and a GeForce FX5950 graphics card.

6.1. Test scenes

The three test scenes comprise a synthetic arrangement of 5000 randomly positioned teapots (11.6M polygons); an urban environment (1M polygons); and the UNC power plant model (13M polygons). The test scenes are depicted in Figure 8. All scenes were partitioned using a kD-tree constructed according to the surface-area heuristics [MB90].

Although the teapot scene would intuitively offer good occlusion, it is a complicated case to handle for occlusion culling. Firstly, the teapots consist of small triangles and so only the effect of fused occlusion due to a large number of visible triangles can bring a culling benefit. Secondly, there are many thin holes through which it is possible to see quite far into the arrangement of teapots. Thirdly, the arrangement is long and thin and so we can see almost half of the teapots along the longer side of the arrangement.

The complete power plant model is quite challenging even to load into memory, but on the other hand it offers good occlusion. This scene is an interesting candidate for testing not only due to its size, but also due to significant changes in visibility and depth complexity in its different parts.

The city scene is a classical target for occlusion culling algorithms. Due to the urban structure consisting of buildings and streets, most of the model is occluded when viewed from the streets. Note that the scene does not contain any detailed geometry inside the buildings. See Figure 4 for a visualization of the visibility classification of the kD-tree nodes for the city scene.

6.2. Basic tests

We have measured the frame times for rendering with only view-frustum culling (VFC), the hierarchical stop-and-wait method (S&W), and our new coherent hierarchical culling method (CHC). Additionally, we have evaluated the time for an "ideal" algorithm. The ideal algorithm renders the visible objects found by the S&W algorithm without performing any visibility tests. This is an optimal solution with respect

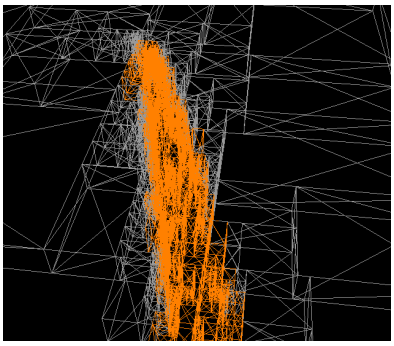

Figure 4: *Visibility classification of the kD-tree nodes in the city scene. The orange nodes were found visible, all the other depicted nodes are invisible. Note the increasing size of the occluded nodes with increasing distance from the visible set.*

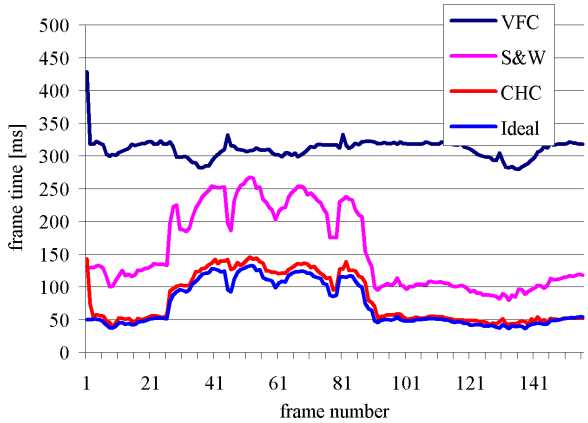

Figure 5: *Frame times for the teapot scene.*

to the given hierarchy, i.e., no occlusion culling algorithm operating on the same hierarchy can be faster. For the basic tests we did not apply any of the optimizations discussed in Section 5, which require user specified parameters.

For each test scene, we have constructed a walkthrough which is shown in full in the accompanying video. Figures 5, 6, and 7 depict the frame times measured for the walkthroughs. Note that Figure 7 uses a logarithmic scale to capture the high variations in frame times during the power plant walkthrough. To better demonstrate the behavior of our algorithm, all walkthroughs contain sections with both restricted and unrestricted visibility. For the teapots, we viewed the arrangement of teapots along the longer side of the arrangement (frames 25–90). In the city we elevated the viewpoint above the roofs and gained sight over most of the city (frames 1200–1800). The power plant walkthrough contains several viewpoints from which a large part of the model is visible (spikes in Figure 7 where all algorithms are slow),

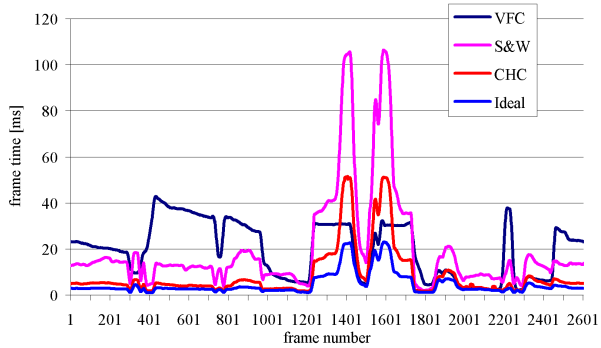

Figure 6: *Frame times for the city walkthrough. Note the spike around frame 1600, where the viewpoint was elevated above the roofs, practically eliminating any occlusion.*

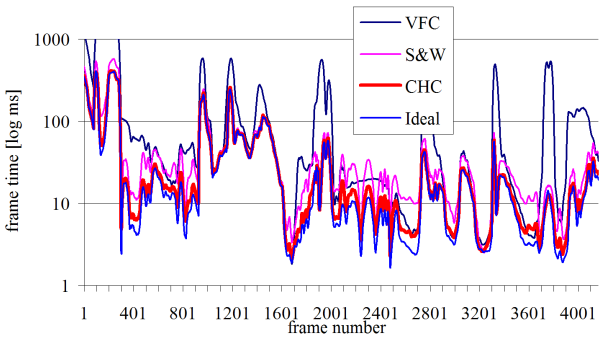

Figure 7: *Frame times for the power plant walkthrough. The plot shows the weakness of the S&W method: when there is not much occlusion it becomes slower than VFC (near frame 2200). The CHC can keep up even in these situations and in the same time it can exploit occlusion when it appears (e.g. near frame 3700).*

viewpoints along the border of the model directed outwards with low depth complexity (holes in Figure 7 where all algorithms are fast), and viewpoints inside the power plant with high depth complexity where occlusion culling produces a significant speedup over VFC (e.g. frame 3800).

As we can see for a number frames in the walkthroughs, the CHC method can produce a speedup of more than one order of magnitude compared to VFC. The maximum speedup for the teapots, the city, and the power plant walkthroughs is 8, 20, and 70, respectively. We can also observe that CHC maintains a significant gain over S&W and in many cases it almost matches the performance of the ideal algorithm. In complicated scenarios the S&W method caused a significant slowdown compared to VFC (e.g. frames 1200–1800 of Figure 6). Even in these cases, the CHC method maintained a good speedup over VFC except for a small number of frames.

Next, we summarized the scene statistics and the average values per frame in Table 1. The table shows the number of issued occlusion queries, the wait time representing the CPU stalls, the number of rendered triangles, the total frame time, and the speedup over VFC.

We can see that the CHC method practically eliminates the CPU stalls (wait time) compared to the S&W method. This is paid for by a slight increase in the number of rendered triangles. For the three walkthroughs, the CHC method produces average speedups of 4.6, 4.0, and 4.7 over view frustum culling and average speedups of 2.0, 2.6, and 1.6 over the S&W method. CHC is only 1.1, 1.7, and 1.2 times slower than the ideal occlusion culling algorithm. Concerning the accuracy, the increase of the average number of rendered triangles for CHC method compared to S&W was 9%, 1.4%, and 1.3%. This increase was always recovered by the reduction of CPU stalls for the tested walkthroughs.

6.3. Optimizations

First of all we have observed that the technique of complete elimination of CPU stalls discussed in Section 5.3 has a very limited scope. In fact for all our tests the stalls were almost completely eliminated by the basic algorithm already (see wait time in Table 1). We did not find constants that could produce additional speedup using this technique.

The measurements for the other optimizations discussed in Section 5 are summarized in Table 2. We have measured the average number of issued queries and the average frame time in dependence on the number of frames a node is assumed visible and the pixel threshold of approximate visibility. We have observed that the effectiveness of the optimizations depends strongly on the scene. If the hierarchy is deep and the geometry associated with a leaf node is not too complex, the conservative visibility testing produces a significant speedup (city and power plant). For the teapot scene the penalty for false rendering of actually occluded objects became larger than savings achieved by the reduction of the number of queries. On the other hand since the teapot scene contains complex visible geometry the approximate visibility optimization produced a significant speedup. This is however paid for by introducing errors in the image proportional to the pixel threshold used.

6.4. Comparison to PVS-based rendering

We also compared the CHC method against precalculated visibility. In particular, we used the PVS computed by an offline visibility algorithm [WWS00]. While the walkthrough using the PVS was 1.26ms faster per frame on average, our method does not require costly precomputation and can be used at any general 3D position in the model, not only in a predefined view space.

scene	method	#queries	wait time [ms]	rendered triangles	frame time [ms]	speedup
Teapots	VFC	—	—	11,139,928	310.42	1.0
11,520,000 triangles	S&W	4704	83.19	2,617,801	154.95	2.3
21,639 kD-Tree nodes	**CHC**	**2827**	**1.31**	**2,852,514**	**81,18**	**4.6**
	Ideal	—	—	2,617,801	72.19	5.2
City	VFC	—	—	156,521	19.79	1.0
1,036,146 triangles	S&W	663	9.49	30,594	19.9	1.5
33,195 kD-Tree nodes	**CHC**	**345**	**0.18**	**31,034**	**8.47**	**4.0**
	Ideal	—	—	30,594	4.55	6.6
Power Plant	VFC	—	—	1,556,300	138.76	1.0
12,748,510 triangles	S&W	485	16.16	392,962	52.29	3.2
18,719 kD-Tree nodes	**CHC**	**263**	**0.70**	**397,920**	**38.73**	**4.7**
	Ideal	—	—	392,962	36.34	5.8

Table 1: *Statistics for the three test scenes. VFC is rendering with only view-frustum culling, S&W is the hierarchical stop and wait method, CHC is our new method, and Ideal is a perfect method with respect to the given hierarchy. All values are averages over all frames (including the speedup).*

scene	t_{av}	n_{vp}	#queries	frame time [ms]
Teapots	0	0	2827	81.18
	2	0	1769	86.31
	2	25	1468	55.90
City	0	0	345	8.47
	2	0	192	6.70
	2	25	181	6.11
Power Plant	0	0	263	38.73
	2	0	126	31.17
	2	25	120	36.62

Table 2: *Influence of optimizations on the CHC method. t_{av} is the number of assumed visibility frames for conservative visibility testing, n_{vp} is the pixel threshold for approximate visibility.*

7. Conclusion

We have presented a method for the optimized scheduling of hardware accelerated occlusion queries. The method schedules occlusion queries in order to minimize the number of the queries and their latency. This is achieved by exploiting spatial and temporal coherence of visibility. Our results show that the CPU stalls and GPU starvation are almost completely eliminated at the cost of a slight increase in the number of rendered objects.

Our technique can be used with practically arbitrary scene partitioning data structures such as kD-trees, bounding volume hierarchies, or hierarchical grids. The implementation of the method is straightforward as it uses a simple OpenGL interface to the hardware occlusion queries. In particular, the method requires no complicated geometrical operations or data structures. The algorithm is suitable for application on scenes of arbitrary structure and it requires no preprocessing or scene dependent tuning.

We have experimentally verified that the method is well suited to the NV_occlusion_query supported on current consumer grade graphics hardware. We have obtained an average speedup of 4.0–4.7 compared to pure view-frustum culling and 1.6–2.6 compared to the hierarchical stop-and-wait application of occlusion queries.

The major potential in improving the method is a better estimation of changes in the visibility classification of hierarchy nodes. If nodes tend to be mostly visible, we could automatically decrease the frequency of occlusion tests and thus better adapt the method to the actual occlusion in the scene. Another possibility for improvement is better tuning for a particular graphics hardware by means of more accurate rendering cost estimation. Skipping occlusion tests for simpler geometry can be faster than issuing comparably expensive occlusion queries.

Acknowledgements

This work has been supported by the *Kontakt OE/CZ* grant no. 2004/20. Many thanks to Stefan Jeschke for his help with importing the Power Plant model.

References

[ARB90] AIREY J. M., ROHLF J. H., BROOKS, JR. F. P.: Towards image realism with interactive update rates in complex virtual building environments. In *1990*

Figure 8: *The test scenes: the teapots, the city, and the power plant.*

Symposium on Interactive 3D Graphics (1990), ACM SIGGRAPH, pp. 41–50. 2

[BH01] BITTNER J., HAVRAN V.: Exploiting coherence in hierarchical visibility algorithms. *Journal of Visualization and Computer Animation, John Wiley & Sons 12* (2001), 277–286. 2, 3, 4, 6

[BHS98] BITTNER J., HAVRAN V., SLAVÍK P.: Hierarchical visibility culling with occlusion trees. In *Proceedings of Computer Graphics International '98 (CGI'98)* (1998), IEEE, pp. 207–219. 2

[BMH98] BARTZ D., MEISSNER M., HÜTTNER T.: Extending graphics hardware for occlusion queries in opengl. In *Proceedings of the 1998 Workshop on Graphics Hardware* (1998), pp. 97–104. 2

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

[COCSD03] COHEN-OR D., CHRYSANTHOU Y., SILVA C., DURAND F.: A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics 9*, 3 (2003), 412–431. 2

[CT96] COORG S., TELLER S.: Temporally coherent conservative visibility. In *Proceedings of the Twelfth Annual ACM Symposium on Computational Geometry* (May 1996), pp. 78–87. 2

[CT97] COORG S., TELLER S.: Real-time occlusion culling for models with large occluders. In *Proceedings of the Symposium on Interactive 3D Graphics* (1997), ACM Press, pp. 83–90. 2, 3

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical Z-buffer visibility. In *Computer Graphics (Proceedings of SIGGRAPH '93)* (1993), pp. 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), ACM Press, pp. 1–10. 2, 3

[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. 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 (Oct. 2001), 365–379. 2, 6

[LSCO03] LEYVAND T., SORKINE O., COHEN-OR D.: Ray space factorization for from-region visibility. *ACM Transactions on Graphics (Proceedings of SIGGRAPH '03) 22*, 3 (July 2003), 595–604. 2

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *Visual Computer 6*, 6 (1990), 153–65. 3, 7

[SBS04] STANEKER D., BARTZ D., STRASSER W.: Occlusion culling in OpenSG PLUS. *Computers and Graphics 28*, 1 (Feb. 2004), 87–92. 2, 6

[SC97] SLATER M., CHRYSANTHOU Y.: View volume culling using a probabilistic caching scheme. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST'97)* (1997), ACM Press, pp. 71–78. 2

[SOG98] SCOTT N. D., OLSEN D. M., GANNETT E. W.: An overview of the VISUALIZE fx graphics accelerator hardware. *Hewlett-Packard Journal: technical information from the laboratories of Hewlett-Packard Company 49*, 2 (May 1998), 28–34. 2

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

[WS99] WONKA P., SCHMALSTIEG D.: Occluder shadows for fast walkthroughs of urban environments. In *Computer Graphics Forum (Proceedings of EUROGRAPHICS '99)* (Sept. 1999), pp. 51–60. 2

[WWS00] WONKA P., WIMMER M., SCHMALSTIEG D.: Visibility preprocessing with occluder fusion for urban walkthroughs. In *Proceedings of EUROGRAPHICS Workshop on Rendering* (2000), pp. 71–82. 8

[WWS01] WONKA P., WIMMER M., SILLION F. X.: Instant visibility. In *Computer Graphics Forum (Proceedings of EUROGRAPHICS '01)* (2001), pp. 411–421. 2

[ZMHH97] ZHANG H., MANOCHA D., HUDSON T., HOFF III K. E.: Visibility culling using hierarchical occlusion maps. In *Computer Graphics (Proceedings of SIGGRAPH '97)* (1997), pp. 77–88. 2