EG UK Theory and Practice of Computer Graphics (2006)
M. McDerby, L. Lever (Editors)

Fast Rendering of Complex Dynamic Scenes

V. Kovalčík and J. Sochor

Faculty of Informatics, Masaryk University, Brno, Czech Republic

Abstract

We present a novel algorithm capable of rendering complex dynamic scenes at high frame rates. The key part of the algorithm is occlusion culling which is performed by an optimized usage of the hardware occlusion queries. The spatial organization of the scene using 2-level BSP-like hierarchy helps to speed up evaluating full and partial occlusion of the objects. The algorithm handles both static and dynamic objects and places no restrictions on the shape of objects.

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

1. Introduction

The computer graphics is evolving and more and more complex worlds are becoming common. Recently, high quality models were used mainly in CAD applications, architectural visualizations or scientific simulations. Nowadays it is possible to find very complex environments in many applications, e.g. in common computer games. Unfortunately (never ending story), the raw power of the computers is not increasing fast enough and it is not possible to display such environments directly using just the power of a current 3D graphics hardware. More advanced algorithms have to be used to achieve acceptable frame rates. These algorithms include level-of-detail reduction of complexity and visibility culling. In this paper we will focus on the latter approach.

The visibility culling algorithms are used to detect primitives that are not visible from the current point of view. Most of the commonly used algorithms are based on frustum culling and backface culling. Both approaches process each polygon independently, are easy to implement and are supported by major graphic APIs. Occlusion culling, which can eliminate objects occluded by other objects, is much more problematic. This type of visibility determination cannot process graphic primitives independently but has to take objects relations into account.

If done properly, the occlusion culling can accelerate the rendering significantly. Previous research efforts have gone far in this area, especially for large static scenes (e.g. urban areas) and moving camera. Most of the algorithms published till now use some kind of preprocessing of the scene remaining the same for all the time of the existence of the scene.

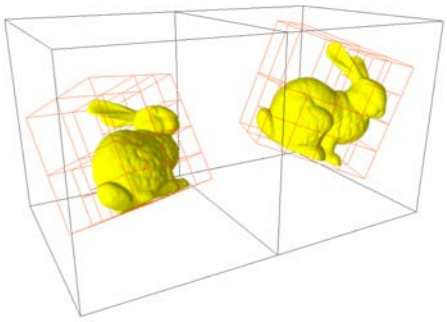

Figure 1: *Two level BSP tree structure: objects in the scene are organized in main BSP-like tree (black) and any object can be partitioned and have its own BSP tree (red).*

Usually some support for moving objects can be implemented. However, if most objects in the scene were moving, the time requirements for the change of preprocessed data would be too high. We want to fill this gap by presenting an algorithm based on BSP-like trees allowing movement of many complex objects and also supporting partial occlusion of objects.

Our main goal was to design a flexible unbiased algorithm

© The Eurographics Association 2006.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

for complex scenes. The algorithm described in this paper has the following properties:

- There are no restrictions on the shapes of objects in a scene.
- All objects serve as potential occluders.
- Entire object as well as only a part of it can be eliminated due to occlusion culling.
- All objects in a scene can move.

A preprocessing of objects is still necessary, but runtime updates of the internal structures are simple and can be done quickly.

2. Related work

In the last decade, many algorithms for accelerating the visibility task were published. These algorithms fall into two main categories: visibility *from-point* or visibility *from-cell*.

From-cell algorithms calculate potentially visible set of objects (PVS) for a region. PVS is usually determined in preprocessing phase and just a few computations are executed at runtime. The typical representative of this group is portal rendering (see Luebke et al. [LG95]), which is frequently used in computer games. Scene is divided into convex cells, which are connected through portals. As the user can see different cells only through the portals, the set of visible cells can be determined by performing only several simple tests in each frame. Hua et al. [HBPF02] and Koltun et al. [KCCO00] proposed different algorithms that divide scene into cells and for every cell several occluders are created. The occluders are very simple objects approximating fusion of several smaller objects and which are used, instead of actual objects, to accomplish occlusion culling at runtime. The *from-cell* algorithms are useful in specific types of environments, such as indoor scenes, but their drawback is a little support of highly dynamic scenes.

Visibility in dynamic scenes can be solved by using *from-point* algorithms. These algorithms determine PVS that is valid for the current point of view. Usually, PVS has to be detected for every change of view, and special arrangements are necessary to lower the cost of this operation. Hudson et al. [HMC*97] created "shadow" volumes that extend away from the camera and are used for culling the intersecting objects. Greene et al. [GKM93] and Zhang et al. [ZMHH97] presented algorithms based on a screen-space hierarchical buffer. The latter uses Hierarchical Occluder Map (HOM), a pyramid of images with decreasing resolution. Every pixel in the HOM represents an average opacity of respective region in image space. The pyramid is generated from several occluders which are selected automatically each frame. This algorithm relies on an appropriate selection of the occluders. If wrong occluders are chosen, the performance can decrease dramatically. However, the careful selection of the occluders is not so crucial in algorithms which use occlusion query functions available in current graphic cards.

Hey at al. [HTP01] proposed an algorithm that constructs "lazy occlusion" grid in screen space and uses the occlusion queries to resolve the visibility of the individual cells if necessary. Hillesland et al. [HSLM02] divide the scene into a 3D grid (either uniform or non-uniform) and the visibility of the cells is queried during the rendering. For a visible cell, the geometry intersecting it is rendered. Bittner et al. [BWPP04] optimized the usage of occlusion queries. Their algorithm performs a traversal of the hierarchy represented as kD-tree. Considering the temporal coherence of previously visible and invisible nodes, unnecesary queries are not issued and the rendering is speeded up significantly.

Yoon et al. [SEYM03] combined occlusion culling with view-dependent rendering (an extension of the progressive meshes [Hop96]) to allow fast rendering with slight image imperfections. This algorithm gives good results for moderate dynamic changes, however it has some drawbacks. If the movement of the camera exhibits only a low temporal coherence (fast camera rotation or a sudden change in position), it would be difficult to find proper occluders and the performance would decrease.

The common problems of above-mentioned algorithms are high memory requirements caused by loading all data in advance. Correa et al. [CKS03] solved it by computing an approximate set of visible objects for the next frame and prefetching the geometry just before it was used. Their method uses much less memory than other approaches, without serious impact on the performance.

More comprehensive surveys of visibility algorithms were presented by Cohen-Or et al. [COCSD00] and Bittner et al. [BW03].

3. Algorithm overview

We present a novel algorithm capable of rendering complex dynamic scenes at high frame rates. The algorithm is based on three key principles: Adaptive space partitioning, dividing objects into parts and optimized issuing of occlusion queries. These will be desribed in detail later.

The algorithm assumes a scene composed of objects which are independent on each other, can be moved freely and can be of an arbitrary complexity. The current version of algorithm can process objects represented as triangular surfaces without any topological constraints.

The basic scheme of the algorithm is as follows: At first, each object is processed individually using its original coordinate system. Each object that is too complex, is divided into parts and a local axes-aligned BSP tree is created. A simple criterion based on the maximum number of triangles in leaves is used to terminate a construction of the tree (details in 3.1).

Afterwards the whole scene is organized globaly using another tree structure. Complex objects with their BSP trees,

© The Eurographics Association 2006.

as well as simpler objects without BSP trees, are positioned into a scene. Axis-aligned BSP-like tree is constructed in a similar way, see 3.2. As the result, a 2-level hierarchy is prepared in the preprocessing phase.

At runtime, the hierarchical spatial structure is traversed in a top-down manner, leaves visited in a front-to-back order with respect to the current position of a camera. The visibility of each node is tested and the nodes found visible are processed further. Objects in the node without local BSP tree are directly rendered by sending them to the graphic card, which draws them using conventional Z-buffer algorithm. The complex objects having their own BSP tree are processed similarly as the main scene tree. The overall scheme is outlined in the Algorithm 1.

Algorithm 1: Overview of the algorithm

```
/* Preprocessing phase                          */
foreach object do
    if TooComplex (object) then
        Divide object;
        Create local BSP tree;
    end
end
Create global BSP-like tree containing all transformed
objects;
/* Runtime phase                                */
foreach frame do
    // Possible changes in the scene
    Update global BSP-like structure;
    Traverse tree hierarchy front-to-back;
    foreach node do
        Try to estimate the visibility;
        if CannotEstimate then
            Issue occlusion query;
        if Visible (node) then
            Render appropriate object parts;
    end
end
```

The scheme is presented in a very simplified form as it does not show overlapping of processing at runtime. The details of the algorithm are discussed in the following sections.

3.1. Object partitioning

Complex objects consisting of many triangles are partitioned in space using an axis-aligned BSP trees. The object is divided recursively by planes until a number of triangles in parts drops below threshold level. The triangles intersecting the plane are split up. Although this process increases a total number of triangles, the overall effect is negligible as large parts of objects can be eliminated by the occlusion culling at runtime.

Whereas a world coordinate system of a scene is fixed, objects in the scene may be positioned and move independently of each other. It implies that the local BSP trees must be also transformed together with their objects. An additional transformation matrix is stored in a root of every local tree node and evaluated after every dynamical change in the scene.

3.2. Scene partitioning

An axis-aligned BSP-like tree is employed for organizing the objects in the scene. The main difference to object partitioning described in 3.1 is that objects are not split, but only tied to a respective node. An object intersecting the splitting plane between two children of a node, is placed in the parent node. Consequently any object belongs exactly to one node, which is the smallest node denoting the subspace the object is fully contained in. Resulting structure is similar to BSP tree, with objects not only in leaves but also in inner nodes.

Analogously to local BSP trees, bounding boxes of subspaces are stored in nodes for additional visibility testing.

In some cases, objects may be placed into the root of the whole tree hierarchy, which means thay have to be processed each frame regardless of the camera position and orientation. The number of such objects is usually not high and it may be lowered using adequate splitting or other form of decomposition.

But even if not, one can assume that such a large object would consist of many triagles and a local BSP tree would be constructed which would support efficient traversal.

To make dynamic changes available, the scene tree is checked at the beginning of each frame. The objects are tested with respect to space subdivision and if neccessary, they are relinked to respective node moving thus up or down in the scene tree. Space subdivision is not changed. As the tree is built using only bounding boxes of objecs and no splitting is accomplished, this operation is fast and inexpensive. Simultaneously we perform "lazy" reconstruction of the scene tree, i.e. after several frames the scene subdivision tree is locally adjusted to reflect the dynamic changes.

3.3. Traversing the structures

The traversing starts at the root of the scene tree. For each visited node the potential visibility is resolved (see 3.4). If the node is visible, its contents are processed further in the following order: The child node closer to a camera is processed first, then the objects stored in the node and finally the child node further from the camera. The objects stored directly in the node are sorted by the distance of their bounding boxes' centroids from the camera. Whenever an object is to be rendered it is first checked for existence of the local BSP tree. An object without tree is immediately rendered on the screen. If the tree is present, the algorithm traverses it in the same way as the scene tree. As mentioned before,

© The Eurographics Association 2006.

respective local transformation must be applied to the local BSP tree as well as to objects rendered. Remaining work can be done in the same way as when dealing with the scene tree. The process is illustrated in Figure 2.

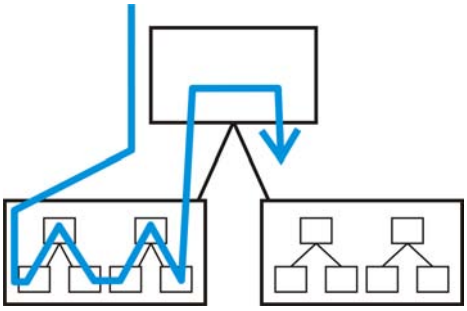

Figure 2: *Traversing the scene tree and the individual object trees.*

3.4. Estimating the visibility

When traversal of the scene tree or the object tree begins the processing of a node, it needs to determine its visibility.

In order to minimize the number of unnecessary occlusion queries, we first perform frustum culling and reject culled nodes. For any other node we should issue occlusion queries to resolve visibility exactly. However, by using a temporal coherence many queries could be saved. For each node we store the results of several recent queries in previous frames and use it to estimate the visibility of the node in the current frame. If the node was not visible in the last frame, we have to issue query to detect possible dynamic change of visibility. If visible, we assume the node will be probably visible again. In this case we use the results of previous queries stored in the node, and count the number of "visible" responses returned over certain time period. The higher the number, the more frames we postpone the query and render the node directly (see Figure 3). This temporal-coherence technique decreases the number of issued queries and speeds up rendering.

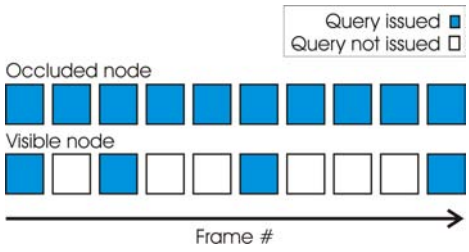

Figure 3: *The more queries returned the result "visible" the more frames we will wait before issuing other query. Meanwhile the node will be assumed visible.*

Furthermore, the queries are issued in parallel if possible. If there are several nodes waiting to be queried for visibility, their bounding boxes are projected onto screen and bounding rectangles are created. The bounding boxes corresponding to non-intersecting rectangles are then sent to graphics card as occlusion queries. Nodes with overlapping projections have to wait until the results of previous queries are returned. To accomplish this we use an auxiliary structure - a list of active nodes waiting to be processed. This list includes both nodes from the scene tree and from the objects' trees. The list is sorted approximately by the distance from the viewer and it is continuously refined as the algorithm traverses the trees. For pseudo-code of the core part of the algorithm see Algorithm 2.

Algorithm 2: Rendering of a frame

```
while not Finished () do
    /* Finish processing of active
       queries                                */
    foreach query in activeNodeQueries do
        if result == Visibile then
            ProcessVisibleNode(query->node);
        end
    end
    /* Process as many nodes as
       possible at once                       */
    foreach node in activeNodes do
        visResult =
        FirstVisibilityTest(node);
        if visResult == NeedQuery then
            if maximum queries reached then
                break;
            Issue query;
        end
        else if visResult == Visibile then
            ProcessVisibleNode (node);
        end
        else if visResult == Invisibile then
            /* Nothing to do...               */
        end
        Delete node from activeNodes ;
    end
end
```

The main part uses two auxiliary functions. *FirstVisibilityTest* function performs several simple tests to roughly estimate the visibility. The pseudo-code can be seen in the Algorithm 3. The last major function is *ProcessVisibleNode* function, which is called for every node that is found visible in the current frame (Algorith 4).

Another speedup might be gained by tightening the boxes used to query the visibility. Instead of querying the bounding box corresponding to a node, we shrink the box so it fits tightly the contents of a node. As this operation could

© The Eurographics Association 2006.

Algorithm 3: **FirstVisibilityTest** function

Input : node
Output: return NeedQuery, Visibile, Invisibile

```
if FrustumCulled() then
    return Invisibile;
if ViewerInside() then
    return Visibile;
if WasInvisibleLastTime() then
    return NeedQuery;
if ExamineRecentQueryResults() then
   /* The more times the object was
      visible recently, the less
      likely is the query issued    */
    return Visibile ;
return NeedQuery ;
```

Algorithm 4: **ProcessVisibleNode** function

Input: aNode

Render aNode->objects without trees;
In activeNodes replace aNode with aNode->child1, aNode->child2;
sort aNode->objects with trees;
foreach *aNode->object with tree* **do**
    insert aNode->object into activeNodes before aNode->child2;
end

be computationally expensive, it is executed only for nodes whose contents have not changed for several frames.

4. Experiments and results

All test were performed on a computer with AMD Sempron 3000+ processor, 2 GB of RAM and ATI Radeon 9500 graphic card with 128 MB of memory.

We have tested the algorithm on two scenes:

- The power plant model containing 1185 objects and about 13 million triangles in total. (Figure 4)
- A scene composed of 300 Stanford bunny models, each having nearly 70 thousand triangles. The bunnies were placed very densely and were moving around freely in a box-shaped area. No collision detection was performed and surfaces were allowed to penetrate without any restriction, see Figure 5.

To determine the performance of rendering algorithm we defined a path through the testing scene and let the camera fly along it. The path was sampled reguraly and the same sequence of viewpoints was evaluated in each test run. The dynamic objects were moving along straight line pathes with the initial positions and movement vectors set by pseudorandom number generator. The positions of the objects at sampled viewpoints were the same each test. We have measured the performace of the following three algorithms:

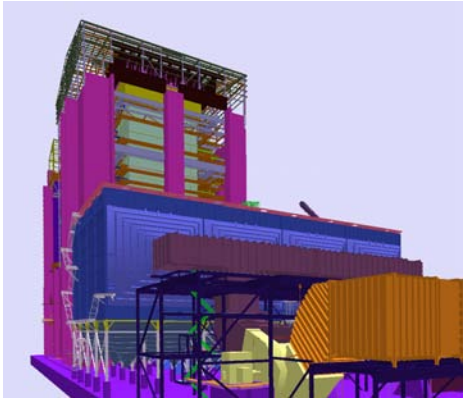

Figure 4: *UNC power plant model.*

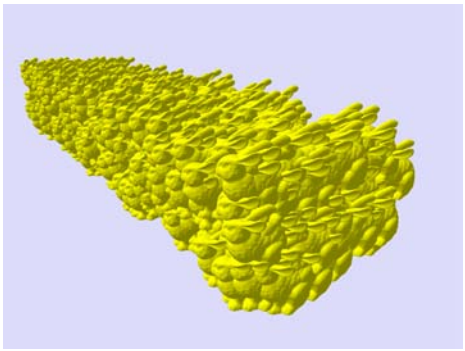

Figure 5: *A scene with 300 moving bunny models.*

- Full - the algorithm described in this paper with occlusion culling and optimized issuing of occlusion queries.
- Simple OC - a simpler version of the algorithm that does not issue occlusion queries in parallel and does not use the recent query results to estimate the visibility. It traverses the BSP tree and a query is issued for every node in the frustum.
- VFC only - The objects are eliminated by view-frustum culling, but no occlusion culling is done and every object in front of the camera is rendered.

Before rendering a dynamic scene we run the preprocessor to subdivide the individual objects and to create the objects' trees. It took about 8 minutes to prepare BSP trees for the objects in the power plant model (1185 objects and about 13 million triangles in total). The Standford bunny was prepared in 5 seconds. None optimization and sophisticated splitting methods were applied. On the other hand, creating and updating a scene tree during rendering phase was very fast (in the order of milisecond) and was plausible to be completed before the rendering of the next frame. The pre-

© The Eurographics Association 2006.

processing increased number of triangles. For example, the Stanford bunny was divided into 23 parts with the total number of triangles to be nearly 77,000. The bunny originally has nearly 70,000 triangles, so the increase is about 10 %.

The results are summarized in Table 1. Note the long time per frame for the "Simple OC" algorithm when applied to the bunny scene. In this case, a huge number of objects and their parts require frequent occlusion queries. Resolving the visibility using the occlusion queries was time costly and in some frames this algorithm preformed worse than brute force approach ("VFC only").

The following graphs show the progress of the powerplant fly-through test. The graphs in Figure 6 and Figure 7 compare the performance of the algorithms described above. Notice that although the "Full" algorithm is the fastest in most frames, there are some frames, where the "VFC Only" suits best. It happens in situations when the camera is looking to an area with low occlusion. The occlusion queries fail and only increase the rendering time needed.

Figure 8 shows number of object parts rendered in individual frames. The best algorithm in this comparison is the "Simple OC", because it determines the occlusion of all nodes for every frame. However, it does not optimize queries, which means it does not use shrinked bounding boxes. That explains some cases when the algorithm performs slightly worse than "Full".

The graph in Figure 9 compares the number of queries issued in each frame. The difference between "Full" and "Simple OC" is the consequence of the estimation of unnecessary queries described in section 3.4.

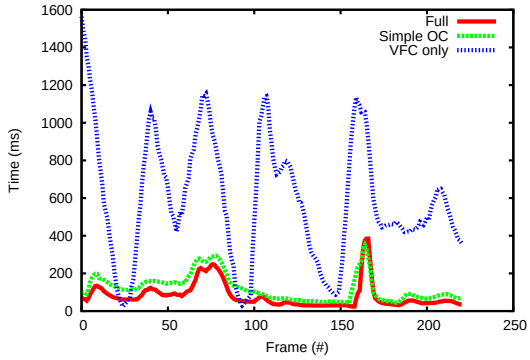

Figure 6: *Rendering time of each frame.*

5. Conclusion

We present an algorithm capable of performing occlusion culling with HW occlusion queries efficiently. It can be applied to any kind of the scene, including dynamic environments. Because of the usage of a two-level BSP-like structure it gives very good performance, yet the implementation remains straightforward.

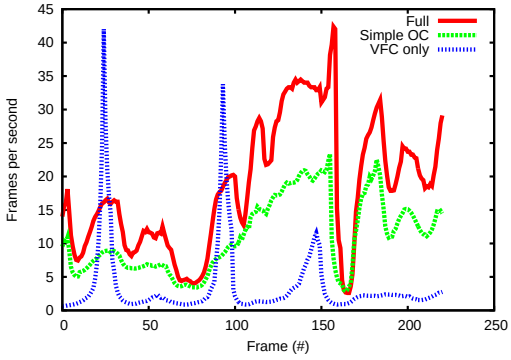

Figure 7: *Rendering speed in fps.*

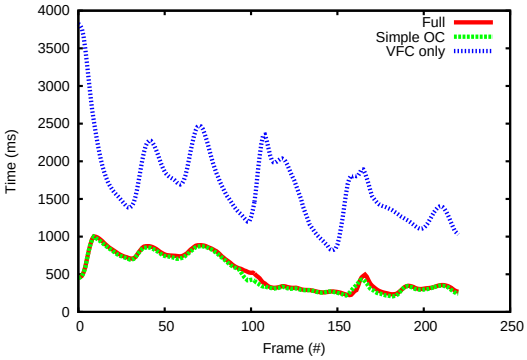

Figure 8: *Number of objects rendered each frame.*

Our algorithm have some weak spots that could be eliminated in the future. The main problem of the algorithm is a creation of new triangles while preparing BSP trees for the individual objects. While the increase of the number of the triangles is not dramatic, in might be troublesome in some cases. This problem could be fixed by using different structure than BSP.

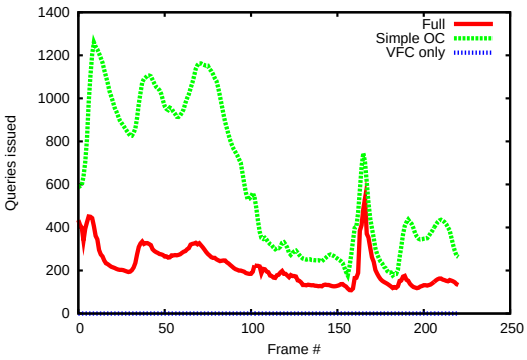

Figure 9: *Number of queries issued each frame.*

© The Eurographics Association 2006.

Scene	Time per frame (ms)			# of rendered object parts			# of occlusion queries		
	Full	Simple OC	VFC only	Full	Simple OC	VFC only	Full	Simple OC	VFC only
Power plant	788	1194	5942	522	504	1678	216	616	0
Moving bunnies	826	3168	3836	6260	2584	22344	2173	14408	0

Table 1: *Comparison of the new algorithm performing both frustum culling and occlusion culling (Full) with the algorithm performing simple occlusion culling without any optimizations of the query issuing (Simle OC) and the algorithm performing view-frustum culling only (VFC only). The "Time per frame" column shows average times spend on rendering one frame. The second column shows the number of parts of an object that have been rendered. The last column shows an average number of queries issued per frame.*

The optimization strategy could be also improved. One possibility is to work adaptively according to the performance of the GPU (for example: On different graphic cards the cost of occlusion query is different).

Finally, the memory requirements could be reduced significantly by using approach similar to [CKS03]. Only necesary objects and kept in the memory and objects estimated to be visible in the subsequent frames are loaded in advance.

6. Acknowledgement

This work was funded by the Ministry of Education, Czech Republic, grant no. LC06008.

References

[BW03] BITTNER J., WONKA P.: Visibility in computer graphics. *Journal of Environment and Planning B: Planning and Design 30*, 5 (2003), 725 – 729.

[BWPP04] BITTNER J., WIMMER M., PIRINGER H., PURGATHOFER W.: Coherent hierarchical culling: Hardware occlusion queries made useful. In *Proceedings of Eurographics 2004* (2004).

[CKS03] CORREA W. T., KLOSOWSKI J. T., SILVA C. T.: Visibility-based prefetching for interactive out-of-core rendering. In *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (Washington, DC, USA, 2003), IEEE Computer Society, p. 2.

[COCSD00] COHEN-OR D., CHRYSANTHOU Y., SILVA C. T., DURAND F.: A survey of visibility for walkthrough applications. In *Proc. of EUROGRAPHICS'00, course notes* (2000).

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical z-buffer visibility. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1993), ACM Press, pp. 231–238.

[HBPF02] HUA W., BAO H., PENG Q., FORREST A. R.: The global occlusion map: a new occlusion culling approach. In *VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology* (New York, NY, USA, 2002), ACM Press, pp. 155–162.

[HMC*97] HUDSON T., MANOCHA D., COHEN J., LIN M., HOFF K., ZHANG H.: Accelerated occlusion culling using shadow frusta. In *SCG '97: Proceedings of the thirteenth annual symposium on Computational geometry* (New York, NY, USA, 1997), ACM Press, pp. 1–10.

[Hop96] HOPPE H.: Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), ACM Press, pp. 99–108.

[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, 2002.

[HTP01] HEY H., TOBLER R. F., PURGATHOFER W.: *Real-Time Occlusion Culling with a Lazy Occlusion Grid*. Tech. Rep. TR-186-2-01-02, Institute of Computer Graphics and Algorithms, Vienna University of Technology, January 2001.

[KCCO00] KOLTUN V., CHRYSANTHOU Y., COHEN-OR D.: Virtual occluders: An efficient intermediate pvs representation. *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering* (2000), 59–70.

[LG95] LUEBKE D., GEORGES C.: Portals and mirrors: simple, fast evaluation of potentially visible sets. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics* (New York, NY, USA, 1995), ACM Press, pp. 105–ff.

[SEYM03] S.-E. YOON B. S., MANOCHA D.: Interactive view-dependent rendering with conservative occlusion culling in complex environments. In *IEEE Visualization* (2003).

[ZMHH97] ZHANG H., MANOCHA D., HUDSON T., HOFF III K. E.: Visibility culling using hierarchical occlusion maps. *Computer Graphics 31*, Annual Conference Series (1997).

© The Eurographics Association 2006.