Culling an Object Hierarchy to a Frustum Hierarchy

Nirnimesh, Pawan Harish, and P.J. Narayanan

Center for Visual Information Technology
International Institute of Information Technology
Hyderabad, India
{nirnimesh@research., harishpk@research., pjn@}iiit.ac.in

Abstract. Visibility culling of a scene is a crucial stage for interactive graphics applications, particularly for scenes with thousands of objects. The culling time must be small for it to be effective. A hierarchical representation of the scene is used for efficient culling tests. However, when there are multiple view frustums (as in a tiled display wall), visibility culling time becomes substantial and cannot be hidden by pipelining it with other stages of rendering. In this paper, we address the problem of culling an object to a hierarchically organized set of frustums, such as those found in tiled displays and shadow volume computation. We present an adaptive algorithm to unfold the twin hierarchies at every stage in the culling procedure. Our algorithm computes from-point visibility and is conservative. The precomputation required is minimal, allowing our approach to be applied for dynamic scenes as well. We show performance of our technique over different variants of culling a scene to multiple frustums. We also show results for dynamic scenes.

1 Introduction

Visibility culling of a scene is central to any interactive graphics application. The idea is to limit the geometry sent down the rendering pipeline to only the geometry with a fair chance of finally becoming visible. It is important for the culling stage to be fast for it to be effective; otherwise the performance gain achieved will be overshadowed. Hierarchical scene structures are commonly used to speed up the process. Hierarchical culling of bounding boxes to a view frustum is fast and sufficient in most applications. Assarsson et al. [1] described several optimizations for view frustum culling. Bittner et al. [2] exploited temporal coherence to minimize the number of occlusion queries for occlusion culling to a view frustum.

Fast frustum culling is particularly crucial for rendering to multiple frustums simultaneously. (1) CAVE [3] is a multi-display virtual-reality environment which requires visibility culling to multiple frustums. (2) Another application using multiple frustums involves occlusion culling of a scene by eliminating objects in the frustum shadows formed by all principal occluders, as proposed by Hudson et al. [4]. (3) Cluster-based tiled displays require fast culling to multiple frustums corresponding to each tile in the display (Figure 7). (4) Multi-projector

P. Kalra and S. Peleg (Eds.): ICVGIP 2006, LNCS 4338, pp. 252–263, 2006.

display systems [5] use several overlapping frustums corresponding to each of the projectors. (5) Multiple frustums are also required to compute visibility for architectural environments [6,7,8].

Any real-world interactive visualization application typically deals with scenes with millions of triangles. An effective way of arranging the scene involves scene graphs. The spatial hierarchy of a scene graph greatly reduces the number of checks for visibility culling. Similarly, when the number of frustums is large, it is natural to also treat them hierarchically. In the most general case, we would want to cull any general object hierarchy to any general frustum hierarchy.

In this paper, we use a hierarchical representation of view frustums to cull the scene to all the frustums coherently. Our method adaptively merges the two hierarchies – the scene hierarchy and the frustum hierarchy – for visibility culling. To this end, we present an algorithm which determines which hierarchy to traverse and when. To our knowledge, this is the first work which considers this decision to be important and effective for coherent culling to multiple frustums. Here, we address the specific problem of culling an object hierarchy to a frustum hierarchy for a tiled display wall (Figure 7). Our tiled display wall system [9] uses a number of commodity systems in a cluster, each powering a tile. The system uses a scene graph (Open Scene Graph [10]) representation of a massive scene. The network resources limit the amount of data that can be transmitted, thereby making efficient visibility culling an important requirement. The individual frustums in the display wall have a fixed arrangement with respect to each other and have a common viewpoint. Such a tight arrangement of frustums motivates our visibility culling algorithm to perform coherent computations which are both fast and scalable. We are able to bring down the culling time for a hierarchical version of UNC's power plant model for a 4×4 tiled display from about 14 ms using the traditional approach to about 5 ms using our adaptive algorithm.

Our visibility culling approach performs *from-point* visibility as opposed to *from-region* visibility performed by several other culling techniques [11,12]. Besides, our culling approach is conservative, as opposed to other probabilistic or approximate culling techniques [13,14,15], which can lead to serious rendering artifacts. This is critical for the kind of applications in which the multiple frustums come into use. For a cluster-based tiled display wall, for instance, the load on the network needs to be minimized and the interactivity needs to be preserved. Culling determines the geometry that will be cached on the rendering nodes. Approximate culling techniques lead to probabilistic prefetching, often leading to freezes during rendering.

We present experimental results from our visibility culling algorithm for the Fatehpur Sikri model and UNC's Power plant model. We have focused only on fast culling to multiple frustums, and have therefore not discussed the later stages in the rendering pipeline. We compare the results with different variants for culling to multiple frustums. We also investigate the performance of our culling technique for a dynamic scene, when many objects change position. This involves additional overheads in updating the bounding boxes at many nodes before the culling can be performed.

2 Related Work

Visibility determination has been a fundamental problem in computer graphics [16] since the scene is typically much larger than the graphics rendering capabilities. Cláudio et al. [17] and Durand et al. [18] have presented comprehensive visibility surveys. View-frustum culling algorithms avoid rendering geometry that is outside the viewing frustum. Hierarchical techniques have been developed [19], as well as other optimizations [1,14]. Funkhouser et al. [20] described the first published system that could support models larger than main memory, based on the from-region visibility algorithm of Teller and Sequin [6]. Aliaga et al. [12] described MMR, the first published system to handle models with tens of millions of polygons at interactive frame rates, although it did require an expensive high-end multi-processor graphics workstation.

Assarsson et al. [1] presented several optimizations for fast view frustum culling, using different kinds of bounding boxes and bounding sphere. For their octant test, they split the view frustum in half along each axes, resulting in eight parts, like the first subdivision of an octree. Using bounding sphere for objects, it is sufficient to test for culling against the outer three planes of the octant in which the center of the bounding sphere lies. This can be extended to general bounding volumes as well [21]. Our frustum hierarchy approach is inspired by this idea of subdividing the view-frustum into octants. However, Assarsson et al. divide the view-frustum only once, whereas we complete this procedure to construct a full frustum hierarchy. Bittner et al. [2] used hardware occlusion query techniques to exploit temporal coherence and reduce CPU-GPU stalls for occlusion culling. Since the occlusion culling information holds good for all frustums for our specific case of tiled display walls, separate occlusion culling for each frustum is not necessary.

Another way to look at occlusion relationships is to use the fact that a viewer cannot see the occludee if it is inside the shadow generated by the occluder. Hudson et al. [4] proposed an approach based on dynamically choosing a set of occluders, and computing their shadow frusta, which is used for culling the bounding boxes of a hierarchy of objects. Bittner et al. [22] improved this method by combining the shadow frusta of the occluders into an occlusion tree. This method has an advantage over Hudson et al. as the comparison in the latter is done on a single tree as opposed to each of the m frustums individually, hence improving the time complexity from $O(m)$ to $O(\log m)$. Our approach of constructing a tree of view frustums resembles this technique of handling frustums. We go even further to combine the frustum hierarchy with object hierarchy.

3 Object Hierarchy and Frustum Hierarchy

We work with two hierarchies in our culling technique. The first is the spatial hierarchy of the scene, represented using a scene graph (OpenGL Performer [23], Open Scene Graph [10]). In the Object Hierarchy (OH), each node has a

bounding volume such that the bounding volume of an internal node entirely encloses the bounding volumes of all its children. Only leaf nodes contain actual geometry. A well-formed scene graph would have compact geometry nodes so that bounding volumes can be used to provide accurate visibility tests.

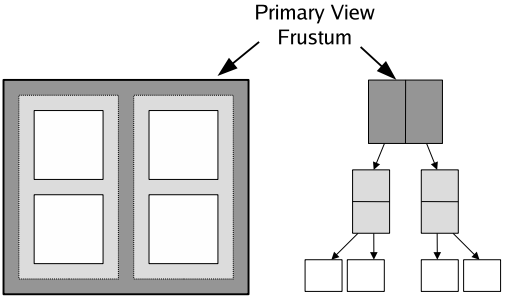


Fig. 1. Frustum Hierarchy (FH). White boxes represent view frustums. Their hierarchical grouping for 3 levels is shown on the right. The bisection plane at each internal node is also shown. Note that near and far planes are not shown.

The second hierarchy we deal with is that of view frustums (Figure 1). Our frustum Hierarchy (FH) is analogous to a BSP-like division. In the most general scenario, a number of independent view frustums in 3D are grouped together hierarchically. Every internal node's bounding volume encloses that of its children. A plane bisects each internal node's volume into half-spaces containing its children. The leaf nodes in the hierarchy correspond to individual view frustums. For a tiled display wall application, each rendering node corresponds to one view-frustum. The root node in the frustum hierarchy corresponds to the primary view-frustum (shown in Figure 1). The case of overlapping view frustums, commonly used for multi-projector displays, is easily handled by treating the overlapping regions as additional independent frustums.

4 Adaptive Traversal of Object and Frustum Hierarchies

Ideal traversal through OH and FH is crucial for optimal performance. The preprocessing step required is discussed in Section 4.1. In Section 4.2, we first discuss several schemes for traversing these hierarchies, and then present our adaptive algorithm.

4.1 Preprocessing

Oriented bounding boxes (OBB) give a compact representation of object's geometry and orientation in space. It is desirable that culling be performed to OBBs only as opposed to the whole geometry since it is fast and conservative. During the preprocessing stage, the scene graph is loaded into main memory. For a set of 3D points, their eigen vectors represent their orientation. Therefore, at

the leaf nodes of OH, the eigen vectors of the geometry points provide oriented bounding boxes. However, at the internal nodes, we compute the eigen vectors using just the children's bounding box vertices. This is a fast approximation of an oriented bounding box for the internal node.

A Frustum hierarchy, FH, is constructed, with each internal node having a bisection plane. Beginning at the root node, `root` in OH, call Algorithm 1 as `preprocess(root)`. Preprocessing takes place in a bottom-up fashion. The bounding box information thereby computed is stored with each of the nodes in the scene graph.

Algorithm 1. preprocess(OH_Node)

```
1: G ⇐ φ
2: if not leaf(OH_Node) then
3:    for all child c of OH_Node do
4:       preprocess(c)
5:       G ⇐ G+ bounding box vertices of c
6:    end for
7: else
8:    G ⇐ G+ OH_Node.getGeometry()
9: end if
10: e ⇐ compute eigen vectors of G
11: BBOX ⇐ compute OBB from e
12: OH_Node.save_bbox(BBOX)
```

4.2 Frustum Culling Approaches

Our culling procedure involves a first level culling to the primary view frustum, so as to eliminate objects completely outside the view. The next step involves classifying these n objects to m view frustums. A *naive approach* involves testing each of these objects with all the view frustums. The expected time complexity for this approach is $O(mn)$. We now discuss several other hierarchical variations to this approach, followed by our adaptive algorithm.

OH without FH: This is a commonly used approach, wherein the scene graph is culled to all the view frustums one by one. It does not exploit any hierarchical arrangement of frustums, and therefore performs poorly with large number of view frustums. This approach has a average time complexity of $O(m \log n)$.

FH without OH: If the frustum hierarchy only is utilized, each object has to be tested against it, beginning from the root. For every internal node in the FH, if an object is present entirely on one side of its bisection plane, its visibility can be safely eliminated from all frustums lying in the other half-space. Therefore, we can potentially eliminate half the number of frustums at each node in the hierarchy. Hence, the average case time complexity is $O(n \log m)$.

Adaptive OH and FH: In both the above cases, the two hierarchies (OH and FH) are used independent of each other, i.e. when the OH is traversed, frustums are treated non-hierarchically and when FH is traversed, objects are treated non-hierarchically. Hence, adaptive merging of the two hierarchies leads to substantial reduction in computations. Consider the different cases:

- At leaf nodes in OH, only FH traversal remains.
- At leaf nodes in FH, only OH traversal remains.
- At all internal nodes, decide whether to further traverse FH or OH.

The precomputed data stored during preprocessing stage (Section 4.1) is utilized to arrive at the above decision. If an OH-node is not intersected by an FH-node's bisection plane, the frustum hierarchy should be unfolded further, keeping the OH intact. Unfolding OH here leads to a large number of OH-nodes to deal with in the next iteration. If the bisection plane of an FH-node intersects the bounding box of an OH-node, OH should be unfolded, thereby breaking the object to its constituents. We classify the children into three groups (L=Left, C=Cuts, R=Right) depending on their position with respect to the FH-node's bisection plane. The group L contains all the children lying completely in negative half-space, R contains those lying in the positive and C contains the rest (the objects that cut the plane).

For each node `OH_Node` determined to be visible in the primary frustum, the algorithm `adaptive_OHandFH_Cull(OH_Node, FH_root)` (Algorithm 2) is called, where `FH_root` is the FH root. `ClassifyLCR()` is an accessory function which categorizes `OH_Node`'s children into the sets L, C and R according to their position with respect to an FH node's current bisection plane. The algorithm recurses for the members in L and R to the corresponding child-frustum (Algorithm 2: lines 11, 14) while the members of C is recursed for both the child-frustums (Algorithm 2: lines 7–8). When the FH is exhausted, the remaining objects are marked to be visible in the corresponding view frustum. The objects in set C need to be checked with both the half-spaces. However, the number of frustums under consideration get reduced by half for objects in set L and R, thereby potentially halving the computations. The number of children to deal with might increase because `classifyLCR()` breaks up an OH node into its children. At this stage, there are two options. We can carry on with each object independently or can regroup the objects in sets L and R into pseudo-groups. This involves recomputing the bounding box for the pseudo-group. Pseudo-groups do not really exist in the scene graph but can reduce the computations required for further stages. Our experiments show that this regrouping is advantageous only for scene graphs with very high branching factor. Otherwise, the overhead of forming the pseudo-group overshadow the gain achieved. Note that pseudo regrouping is not shown in Algorithm 2.

Our adaptive algorithm follows an $O(m\, n^{\log_p q} + (p-q)\log m)$ time complexity, on a quick analysis, where p is the average branching factor of OH and q is the average number of nodes in set C. Exact analysis is difficult as it depends on the goodness of the branching and the spatial separation of child nodes at each level. Hence, p and q depend heavily on the scene structure, the view frustums

arrangement and the viewpoint. The average case time complexity follows a sub-linear pattern. In the worst case, the complexity becomes $O(m\ n)$ when $q = p$, when all OH leaves fall in all FH leaves, and the hierarchy is inconsequential. In the best cast, the complexity is $O(\log m)$ when $q = 0$. This is the situation when only one FH leaf contains the entire OH. In practice, the algorithm is able to adapt to variations in complexity of the visible scene, which is very common during interactive walkthroughs.

Algorithm 2. adaptive_OHandFH_Cull(OH_Node, FH_Node)

```
 1: if leaf(FH_Node) then
 2:    Mark OH_Node as visible to FH_Node
 3:    return
 4: end if
 5: [L, C, R] ⇐ ClassifyLCR(OH_Node, FH_Node.plane)
 6: for all c in set C do
 7:    adaptive_OHandFH_Cull(c, FH_Node.neg)
 8:    adaptive_OHandFH_Cull(c, FH_Node.pos)
 9: end for
10: for all l in set L do
11:    adaptive_OHandFH_Cull(l, FH_Node.neg)
12: end for
13: for all r in set R do
14:    adaptive_OHandFH_Cull(r, FH_Node.pos)
15: end for
```

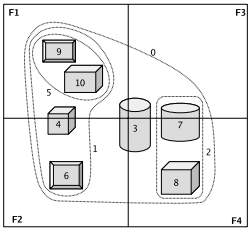


Fig. 2. Hierarchy of objects as visible to a 2×2 tiled arrangement of view frustums. The grouping of objects is shown. F1, F2, F3 and F4 represent view frustums. Their adaptive culling is shown in Figure 3.

Line 2 of Algorithm 2 marks the `OH_node` as visible to the `FH_node`. Line 5 performs the classification of the object node to L, C and R. Lines 7, 8 recurses for every child in C, the set of objects cut by the plane. Lines 11 and 14 recurse to the next stage of FH.

The algorithm can easily deal with dynamic scenes as well, since the preprocessing stage involves cheap eigen-vector calculations only. The visibility determination remains unchanged. Besides, very little extra data is stored for the algorithm execution. Note that bisection using a plane is possible in Algorithm 2

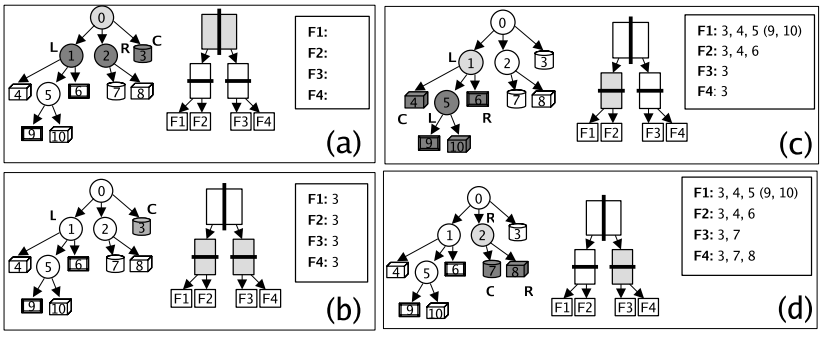


Fig. 3. Adaptive culling of the scene structure in Figure 2. The object and frustum hierarchies are shown along with already determined visibility list. Working nodes are shown as light-gray. Dark gray objects are the ones that need to be recursed further. (a) OH root is classified as per the bisection plane of the FH root. L, C, R classification is shown. (b) Continuing culling for set C. (c) Continuing culling for set L. (d) Continuing culling for set R.

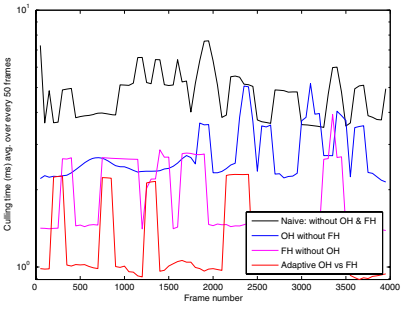
for an application such as tiled display walls because the tile sizes are uniform and the frustum space ultimately divides to form the individual tile frustums. This might not be true for non-uniform frustums. However, a hierarchy of frustums can still be built. Only, in such a case, the terminal frustum in line 2 of Algorithm 2 will further involve a check for visibility before marking an object as visible.

5 Experimental Results

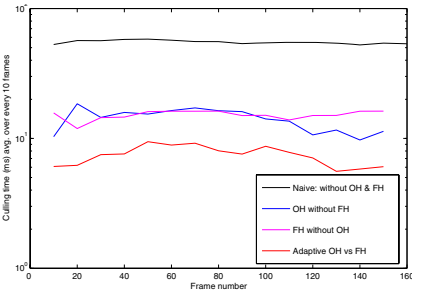
We perform several walkthrough experiments on models of different scene complexities to test the performance of the adaptive algorithm. We used a hierarchical model of Fatehpur Sikri, which has 1.6 M triangles spread over 770 nodes (288 internal + 482 leaf), with an average branching factor of 2.67. We also used a hierarchical model of UNC's power plant, which has geometry spread over 5037 nodes (1118 internal + 3919 leaf), with an average branching factor of 4.5.

Figure 4(a) shows a logarithmic plot of culling time taken by various algorithms discussed in Section 4.2 for a 4000 frame walkthrough on the Fatehpur model. The walkthrough is such that the entire scene is visible. This is a worst-case situation; typical walkthroughs perform better. Our adaptive algorithm (Algorithm 2) takes the least time almost throughout the walkthrough. This is followed by the *FH without OH* approach. The *OH without FH* approach performs worse than both these two.

Figure 4(b) shows the culling time for a walkthrough on the power plant model. The plots for *OH without FH* and *FH without OH* approaches coincide, as opposed to lagging performance by *OH without FH* approach in Figure 4(a). This is because the high branching factor in OH makes the *OH without FH* approach more significant. However, the adaptive algorithm performs significantly better than all the other approaches.

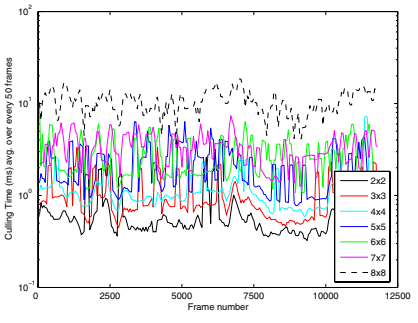


(a) A 4000 frame walkthrough on the Fatehpur Sikri model

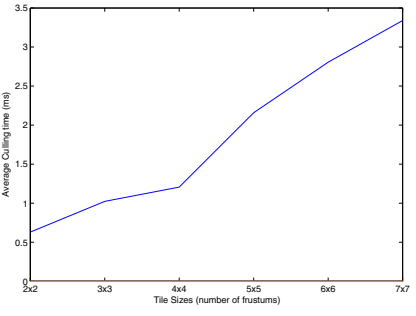


(b) Walkthrough on the power plant model

Fig. 4. Culling performance for various approaches. The lower the time, the better. Our adaptive algorithm outperforms others almost throughout the walkthrough.



(a) Culling scalability plot for a 11000 frame walkthrough

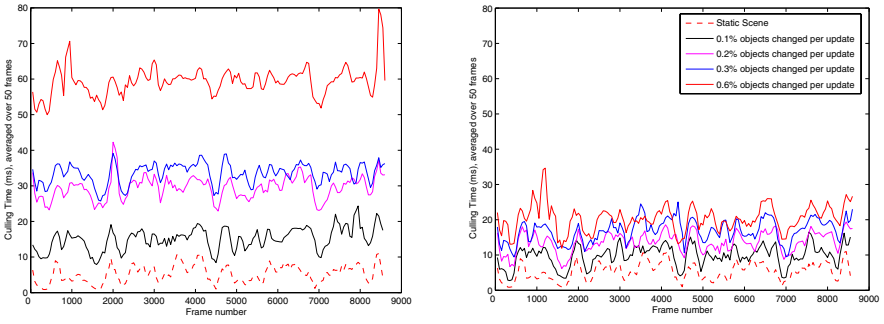


(b) Plot showing average culling time against different tile configurations

Fig. 5. Culling scalability performance on the Fatehpur Sikri model

We conducted scalability tests with respect to the tile size for different configurations of tiled displays. Figure 5 shows the plots for our adaptive algorithm for an 11000 frame walkthrough of the Fatehpur Sikri model. The algorithm takes about 11 ms, on an average, for culling to an 8×8 configuration, thereby making the culling applicable for setting up display walls of such configuration. Otherwise culling time limits the overall frame rate achievable on a server-managed display wall such as ours(Figure 7, [9]), where the rendering is done by client machines and data-transmission can be performed in parallel with the culling of the next frame.

Figure 6 shows the performance of our adaptive algorithm for a dynamic scene. Different percentages of the scene is changed prior to every update. Dynamic scenes have objects moving in space. The bounding boxes of these objects and their parents till the OH root need to be recomputed. Fast OBB computations (Section 4.1) permit dynamic scenes to be culled at interactive frame rates. Although speed can be further increased with axis-aligned bounding boxes (AABB), it comes at the cost of poor visibility culling. In an optimal situation,



(a) OBBs used at dynamic nodes (b) AABBs used at dynamic nodes

Fig. 6. Culling performance on the power plant scene for different percentages of dynamic objects. The model has a total of 5037 objects. The performance with AABB is better than with OBB but at the cost of over-conservative visibility culling.

Fig. 7. A 4×4 display wall rendering of UNC's Powerplant. The combined resolution is 12 MPixels. Efficient rendering to a display wall requires fast visibility culling of the scene to all the frustums. Adaptive culling by merging of the object and frustum hierarchies makes this possible for even bigger tile configurations.

a hybrid of both OBBs and AABBs should be used. It is beneficial to compute AABBs for dynamic portions of the scene. Note that the percentage of dynamic objects shown in Figure 6 are extreme case situations. In practice, the scenes are less dynamic and so the adaptive algorithm performs even better.

6 Conclusions and Future Work

We presented a conservative, from-point visibility culling approach for culling a large scene to a hierarchy of view-frustums. We presented an adaptive algorithm

which determines the optimal path, merging object and frustum hierarchies. The algorithm performs logarithmically in practice. Due to this, it can scale well for large number of frustums which is critical for the application scenario of a tiled display wall. The performance gain by our algorithm is shown on several walkthrough experiments. The algorithm makes culling for very large tile display setups feasible. Huge models can be handled at interactive frame rates. We also showed that the adaptive algorithm is applicable for dynamic scenes as well.

Though we showed the performance on a two-dimensional, tight-fit array of frustums, the results can be extended to other hierarchies of frustums. We are currently extending it to other typical situations like a 2D array of frustums with small overlap used in multi-projector displays. We are also working on culling to general frustum hierarchies needed for applications like shadow volume computations. A BSP-tree like partitioning of the frustums, very similar to our current approach, will be needed in such cases.

Acknowledgements. This work was partially funded by Microsoft Research through their University Relations, India.

References

1. Assarsson, U., Möller, T.: Optimized View Frustum Culling Algorithms for Bounding Boxes. Journal of Graphics Tools: JGT **5** (2000) 9–22
2. Bittner, J., Wimmer, M., Piringer, H., Purgathofer, W.: Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful. Comput. Graph. Forum **23** (2004) 615–624
3. Cruz-Neira, C., Sandin, D.J., DeFanti, T.A.: Surround-screen projection-based virtual reality: the design and implementation of the CAVE. In: SIGGRAPH. (1993)
4. Hudson, T., Manocha, D., Cohen, J., Lin, M., Hoff, K., Zhang, H.: Accelerated occlusion culling using shadow frusta. In: Symposium on Computational geometry. (1997)
5. Raskar, R., Brown, M.S., Yang, R., Chen, W.C., Welch, G., Towles, H., Seales, W.B., Fuchs, H.: Multi-projector displays using camera-based registration. In: IEEE Visualization. (1999) 161–168
6. Teller, S.J., Sequin, C.H.: Visibility preprocessing for interactive walkthroughs. In: SIGGRAPH. (1991)
7. Airey, J.M.: Increasing update rates in the building walkthrough system with automatic model-space subdivision and potentially visible set calculations. PhD thesis (1990) Director-Frederick P. Brooks, Jr.
8. Airey, J.M., Rohlf, J.H., Frederick P. Brooks, J.: Towards image realism with interactive update rates in complex virtual building environments. In: Symposium on Interactive 3D graphics. (1990)
9. Nirnimesh, Narayanan, P.J.: Scalable, Tiled Display Wall for Graphics using a Coordinated Cluster of PCs. In: 14th Pacific Conference on Computer Graphics and Applications (Pacific Graphics). (2006)
10. Burns, D., Osfield, R.: OpenSceneGraph A: Introduction, B: Examples and Applications. In: IEEE Virtual Reality Conference. (2004)

11. Funkhouser, T.A.: Database Management for Interactive Display of Large Architectural Models. In: Graphics Interface. (1996)
12. Aliaga, D.G., Cohen, J., Wilson, A., Baker, E., Zhang, H., Erikson, C., III, K.E.H., Hudson, T., Stürzlinger, W., Bastos, R., Whitton, M.C., Jr., F.P.B., Manocha, D.: MMR: an interactive massive model rendering system using geometric and image-based acceleration. In: Symposium on Interactive 3D Graphics. (1999)
13. Corrêa, W.T., Klosowski, J.T., Silva, C.T.: Visibility-Based Prefetching for Interactive Out-Of-Core Rendering. In: IEEE Symposium on Parallel and Large-Data Visualization and Graphics. (2003)
14. Klosowski, J.T., Silva, C.T.: The Prioritized-Layered Projection Algorithm for Visible Set Estimation. IEEE Trans. Vis. Comput. Graph. **6** (2000) 108–123
15. Slater, M., Chrysanthou, Y.: View volume culling using a probabilistic caching scheme. In: Virtual Reality Software and Technology. (1997)
16. Sutherland, I.E., Sproull, R.F., Schumacker, R.A.: A characterization of ten hidden-surface algorithms. ACM Comput. Surv. **6** (1974) 1–55
17. Cohen-Or, D., Chrysanthou, Y., Silva, C.T., Durand, F.: A survey of visibility for walkthrough applications. IEEE Trans. Vis. Comput. Graph. **9** (2003) 412–431
18. Durand, F.: 3D Visibility: analytical study and applications. PhD thesis, Université Joseph Fourier, Grenoble I (1999)
19. Clark, J.H.: Hierarchical Geometric Models for Visible Surface Algorithms. Commun. ACM **19** (1976) 547–554
20. Funkhouser, T.A., Séquin, C.H., Teller, S.J.: Management of large amounts of data in interactive building walkthroughs. In: Symposium on Interactive 3D Graphics. (1992)
21. Assarsson, U., Möller, T.: Optimized View Frustum Culling Algorithms. Technical Report 99–3, Department of Computer Engineering, Chalmers University of Technology (1999)
22. Zhang, H., Manocha, D., Hudson, T., Kenneth E. Hoff, I.: Visibility culling using hierarchical occlusion maps. In: SIGGRAPH. (1997)
23. Rohlf, J., Helman, J.: IRIS Performer: a high performance multiprocessing toolkit for real-time 3D graphics. In: SIGGRAPH. (1994)