OpenGL-assisted Visibility Queries of Large Polygonal Models

Tobias Hüttner, Michael Meißner, Dirk Bartz

WSI-98-6

ISSN 0946-3852
WSI/GRIS
University of Tübingen
Auf der Morgenstelle 10/C9
D72076 Tübingen, Germany

Email: {thuettne, meissner, bartz}@gris.uni-tuebingen.de
WWW: http://www.gris.uni-tuebingen.de

ABSTRACT

We present an OpenGL-assisted visibility culling algorithm to improve the rendering performance of large polygonal models. Using a combination of OpenGL-assisted frustum culling, hierarchical model-space partitioning, and OpenGL based occlusion culling, we achieve a significant better performance on general polygonal models than previous approaches. In contrast to these approaches, we only exploit common OpenGL features and therefore, our algorithm is also well suited for low-end OpenGL hardware.

Furthermore, we propose a small addition to the OpenGL rendering pipeline, to enhance the framebuffer's ability for faster visibility queries.

CR Categories:

I.3.5 [Computer Graphics]: Visibility Culling, Occlusion Culling
I.3.7 [Three-Dimensional Graphics and Realism]:
Hidden Line/Surface Removal

Keywords:

Occlusion culling, hierarchical data structures, OpenGL, sloppy n-ary space partitioning tree

1 Introduction

Hidden-line-removal and visibility are among the classic topics in computer graphics [7]. A large variety of algorithms are known to solve these visibility problems, including the z-buffer approach [3], the painter algorithm [7], and many more.

Recently, visibility has been of special interest for walkthroughs of architectural scenes [1, 22, 18] and rendering of large polygonal models [16, 9]. Unfortunately, these approaches are limited to cave-like scenes [16, 23], or require special hardware support [14].

In this paper, we present an algorithm for general visibility queries. This algorithm exploits several OpenGL features in order to obtain faster results for large polygonal models. To show the applicability of our algorithm even on low-end graphics workstations, we performed all measurements on a SGI O_2/R10000 workstation. Furthermore, we propose an extension to the OpenGL rendering pipeline to add features for improved general visibility culling.

In a pre-process, the model is subdivided into a sloppy n-ary space-partitioning-tree (snSP-tree). In contrast to ordinary partitioning-trees, like the BSP-tree [8], our subdivision is not a precise one; snSP-tree sibling nodes may not be disjunct. This is to prevent large numbers of small fractured polygons, which can cause numerical problems and increase the rendering load.

During the actual visibility culling, the OpenGL selection buffer is used to implement a view-frustum culling of the nodes of our subdivision tree. Thereafter, the remaining nodes of our snSP-tree are rendered into our implementation of a *virtual occlusion buffer* to determine the non-occluded nodes. Finally, the polygons of the snSP-nodes [24] considered non-occluded are rendered into the framebuffer.

Overall, our algorithm features:

- **Portability:** Only basic OpenGL-functionality is used for the implementation of the algorithm. No additional hardware support for texture-mapping or special visibility queries is necessary. Even low-end OpenGL supporting PC graphic hardware is able to use our visibility culling scheme.
- **Adaptability:** Due to the use of the OpenGL rendering pipeline, the presented algorithm adapts easily to any OpenGL graphics card. Features that are not supported in hardware can be disabled, or are realized in software by the OpenGL implementation.
- **Generality:** No assumption of the scene topology or restriction on the scene polygons are made.
- **Significant Culling:** Although high culling performance is always a trade-off between culling efficiency and speed efficiency, our algorithm obtains high culling performance, while keeping good rendering performance.
- **Well-balanced Culling:** Different computer systems introduce different graphics and cpu performance. The presented algorithm provides an adaptive balancing scheme for culling and rendering load.

Our paper is organized as follows: In Section 2, we briefly outline previous work that has been done in the field of visibility culling. Section 3 presents details of our algorithm. Section 4 analyses the results of our algorithm and provides some comparison to related algorithms. Finally, we state our conclusion and briefly describe future work.

2 Related Work

There are several papers which provide a survey of visibility algorithms. In [24], Zhang provides a brief recent overview with some comparison. Brechner surveys methods for interactive walkthroughs [2]. Visibility algorithms for flight simulation are surveyed in [19].

Early approaches are based on culling hierarchical subdivision blocks of scenes to the view-frustum [9]. Although this is a simple but effective scheme for close-ups, this approach is less suited for scenes that are densely occluded, but lie completely within the view-frustum.

In architectural model databases, the scene is usually subdivided into cells, where each cell is associated with a room of the building. For each potential view point of the cells, the potential visible set (PVS) is computed to determine the visibility. Several approaches have been proposed in [1, 22, 18]. However, it appears that the cell subdivision scheme is not suitable for general polygonal scenes without room-like subdivision. Therefore, these approaches are of no apparent importance for general visibility problems.

Several algorithms have been proposed in computational geometry. A brief overview can be found in [10]. Coorg and Teller proposed two object space culling algorithms. In [5], a conservative and simplified version of the aspect graph is presented. By establishing visibility changes in the neighborhood of single occluders using hierarchical data structures, the number of events in the aspect graph is significantly reduced.

Secondly, by combining a shadow-frustum-like visibility test of hierarchical subdivision blocks (e.g. octree blocks), the number of visibility queries is reduced [6]. However, both algorithms are neither suited for dense occluded scenes with rather small occluders (resulting in a large increase of queries), nor for dynamic scenes.

Cohen-Or proposed an ϵ-Visibility-Culling for distributed client/server walkthroughs [4]. Computing the shadow-frusta for a series of local view points and an occluder permits visibility queries on the local client. However, the algorithm does not seem to scale for very highly occluded scenes. Moreover, only two-dimensional visibility queries are possible in the current implementation.

In [17], an occluder database - a subset of the scene database - is selected. During the visibility culling, the shadow-frusta of the occluders are computed and a scene hierarchy is culled against these shadow-frusta. Overall, the surveyed computational geometry-based visibility approaches only deal with convex occluders, which limits their practical use severely.

In 1993, Greene et. al. proposed the hierarchical z-buffer algorithm [14, 13, 11], where a simplified version for anti-aliasing is used in [13]. After subdividing the scene into an octree, each of the octants is culled to the view-frustum as proposed in [9]. Thereafter, the silhouettes of the remaining octants are scan-converted into the framebuffer to check if these blocks are visible. If they are visible, their content is assumed to be visible too; if they are not visible, nothing of their content can be visible. The visibility query itself is performed by checking a z-value-image-pyramid for changes. Unfortunately, this query is not supported by common graphics hardware and therefore, becomes an expensive operation. However, we consider this algorithm as the parent for our approach, presented in Section 3.

In [12], Greene presents a hierarchical polygon tiling approach using coverage masks. This algorithm improves the visibility query of a hierarchical z-buffer, due to the two-dimensional character of the tiling. However, the main contribution of this algorithm is an anti-aliasing method, as the algorithm has advantages for very high-resolution images. The strict front-to-back order traversal of the polygons - necessary for the coverage masks - needs some data structure overhead. Building a hierarchy of an octree of BSP-trees limits the application of this algorithm to static scenes.

Naylor presented an algorithm, based on a 3D BSP-tree for the representation of the scene, a 2D BSP-tree as image representation, and an algorithm to project the 3D BSP-tree subdivided scene into the 2D BSP-tree image [20]. This approach can be considered as a generalization of Greene's hierarchical z-buffer algorithm [14].

Hong et. al. proposed a fusion between the hierarchical z-buffer algorithm [14] and the PVS-algorithm in [18]. In this z-buffer-assisted visibility algorithm, a human colon is first subdivided into a tube of cells in a pre-process. Thereafter, the visibility is determined on-the-fly by checking the connecting portals between these colon cells, exploiting the z-buffer and temporal coherence to obtain high culling performance [16]. Unfortunately, this approach is closely connected to the special tube-like topology of the colon and therefore, is not suited for general visibility problems.

In [23], a voxel-based visibility algorithm is presented. After classifying the scene on a grid of samples of the dataset as void-cells, solid-cells and data-cells, the visibility is determined in a pre-process for each potential view point. We expect good results for cave-like scenes, but a high memory and processing overhead for sparse scenes like the forest scene of Section 4. Therefore, this algorithm is not suited for a general visibility algorithm.

Last year, occlusion culling using hierarchical occlusion maps was presented [24]. Similar to [17], an occluder database is selected from the scene database. Using these occluders, bounding boxes of the potential occludees of the scene database are tested for overlaps, using the image hierarchy of the projected occluders. Further discussion of this very interesting approach can be found in Section 4.

Strategies for dynamic scenes are presented in [21] and [24]. Sudarsky and Gotsman propose a fast update of the hierarchical data structure, as the octree block of the hierarchical z-buffer. Zhang [24] et. al. suggest using each object of a scene as an occluder in the hierarchical occlusion maps algorithm.

3 OpenGL-assisted Visibility Culling

In this Section we present a novel technique to the visibility problem. Our algorithm is based on core OpenGL functionality and utilizes the available capabilities of OpenGL to check for occlusion. The basic strategy is to use a hierarchical spatial subdivision of the model and cull all not visible subdivision nodes. As a hierarchical representation of a scene, we use a sloppy n-ary Space Partitioning tree (snSP-tree), which is generated once per scene as a pre-processing step.

For each frame to be rendered, we perform view-frustum culling and occlusion culling. Figure 1 schematically illustrates the pipeline of our culling algorithm.

3.1 snSP-Tree

Our sloppy n-ary Space Partitioning tree stores the polygons of the model in its leaves. Each node within the tree hierarchy has a variable number of children (n-ary). The individual nodes represent a bounding volume being the superset of the bounding volumes of its child nodes. The sloppiness is given by the sloppy partitioning of the model, where the bounding volumes of sibling nodes may not be disjunct. Therefore, any given model can be stored in such a tree. No re-triangulation is necessary, due to a missing strict subdivision border for the polygons. Nevertheless, polygons which expand over the entire model, such as floors, should be subdivided into smaller polygons to ensure a well balanced tree. In general, leaves contain the polygons, branch nodes represent a hierarchy of bounding volumes, and the root node reflects the bounding volume of the entire model.

3.2 View-Frustum Culling

In contrast to other approaches, we use OpenGL to perform the view-frustum culling step. In detail, we use the *OpenGL selection mode* to detect whether a bounding volume interferes with the view-frustum. Therefore, the bounding volume, as convex hull, is transformed and clipped. Once a bounding volume intersects the view-frustum, we test whether the bounding volume resides entirely within the view-frustum. In this case, all subtrees of the bounding volume can be marked visible. Again, this test is performed using the selection mode of OpenGL, testing whether all vertices are still visible after transformation and clipping. As a result of the view-frustum culling step, leaves are tagged *possibly visible* or *definitely not visible*.

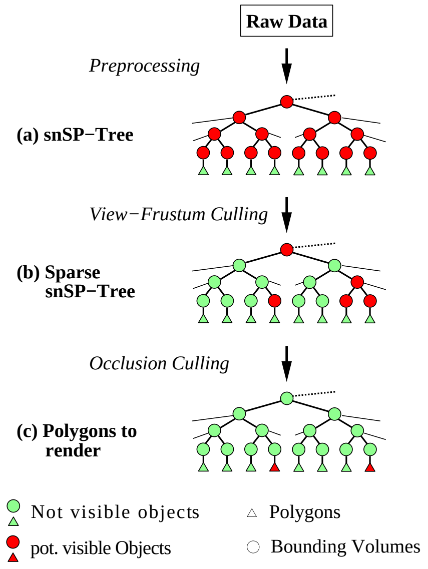

Figure 1: Survey of the basic algorithm.

3.3 Occlusion Culling

The task of an occlusion culling algorithm is to determine occlusion of objects in a model. We use a *virtual occlusion buffer*, being mapped onto the OpenGL framebuffer to detect possible contribution of any object to the framebuffer. In our implementation of the algorithm on a SGI O_2, we used the stencil buffer for this purpose[1]. Intentionally, the stencil buffer is used for advanced rendering techniques, like multi-pass rendering.

To test occlusion of a node, we send the triangles of its bounding volume to the OpenGL pipeline, use the z-buffer test while scan-converting the triangles, and redirect the output into the virtual occlusion buffer. Occluded bounding volumes will not contribute to the z-buffer and hence, will not cause any footprint in the virtual occlusion buffer.

Although reading the virtual occlusion buffer is fairly fast, it is a costly part of our algorithm. This is due to the time consumed during conversion of values inside the OpenGL pipeline, before they are finally returned to the user. For models subdivided into thousands of bounding volumes, this can lead to a less efficient operation. Furthermore, large bounding boxes require many read operations. Therefore, we implemented a progressive occlusion test, which reads spans of pixels from the virtual occlusion buffer using a double interleaving scheme, as illustrated in Figure 2. Although, the setup time for sampling small spans of the virtual occlusion buffer increases the time per sample, spans of ten pixels achieved

[1]Other buffers could be used as well but the stencil buffer is the least used buffer and has an empirically measured better read performance than the other buffers.

an almost similar speed-up as sampling entire lines of the virtual occlusion buffer. We assume that the compromised setup time for sampling small spans is amortized by the higher probability to find footprints, due to the additionally in Y interleaved scheme.

Sampling reflects the number of iterations needed to fully read the entire bounding box. By default, each frame performs only one iteration and hence, reads a $\frac{1}{Sampling}$th of each bounding box. During motion, this feature enables low culling cost without producing visible artifacts. Once the movement stops, the buffer will be read progressively until all values are tested. Basically, every $sampling$th horizontal line is read from the buffer, where the y-offset is incremented by $\frac{sampling}{2}$ for every second column of spans.

For the purpose of illustration, Figure 2 uses a smaller sampling factor as our actual implementation. However, it turned out that a sampling factor of ten is sufficient without compromising image quality. The sampling value can be adjusted adaptively.

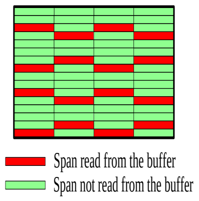

Figure 2: Progressive sampling of the virtual occlusion buffer using a sampling value of four hence, only a fourth of the buffer is read.

3.4 Adaptive Culling

For complex models with deep visibility, i.e., a forest scene, many almost occluded objects contribute only small parts to the final image. Knowing whether an object is visible does not introduce a measure of the quantity of contribution. To cull objects which are almost occluded and therefore, are barely noticeable, we introduce adaptive culling as our alternative to approximate culling [24].

Each object which generates a footprint on the virtual occlusion buffer needs to be evaluated. Therefore, we count the number of footprints of the object on the virtual occlusion buffer. We consider the depth of the object, the size of its 2D bounding box relative to the view plane, and the number of footprints. In other words, we calculate the percentage of footprints relative to the depth and size of the object.

$$Adap_{cull}(Obj) = \frac{SizeOf2DBoundingBox(Obj)}{SizeOfViewplane} * \frac{Dist(Eye) + Dist(Obj)}{Dist(Eye)} \quad (1)$$

where Dist(Obj) returns the minimal distance between the *Obj* and the view plane.

For each potentially visible object we evaluate Equation 1. If $Adap_{cull}(Obj)$ is smaller than the user defined threshold $Thres_{adap}$, we consider the object as occluded, even though it has a small contribution to the image.

Figure 3 shows some results of our adaptive culling mode, compared to the usual occlusion culling mode of our algorithm.

3.5 Further Optimizations

Interleaved Culling: It is obvious that a tree representing a model can be too deep to possibly test every bounding volume for occlusion. In the worst case, each leave could contain a single polygon. This is circumvented by generating well balanced trees, holding sufficient polygons in each leaf. Additionally, the view-frustum culling step and occlusion culling step are dynamically interleaved to exploit culling coherence - an already occluded bounding volume of a tree node does not require any further culling test for its child nodes.

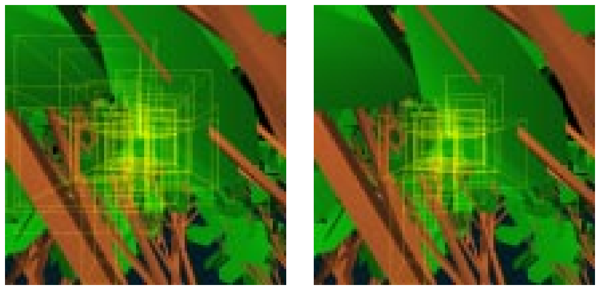

Figure 3: Alley of trees - bounding volumes of culled objects are marked yellow: (a) Adaptive culling. (b) Occlusion culling.

Cost-adaptive Culling: To obtain a good ratio between time spent for rendering and time spent for culling, we need to ensure that only a reasonable fraction of the rendering time is spent on culling. $F_{graphics}$, the factor which represents this ratio, is hardware dependent and needs to be determined empirically. On the SGI O$_2$, we determined $F_{graphics} = \frac{1}{3}$ as a good factor.

The cull depth adapts dynamically in order to meet the time budget. This budget is calculated using Equation 2, where T_{render} is the absolute amount of time spent for rendering the previous frame.

$$T_{budget} = T_{render} * F_{graphics} \quad (2)$$

Once our algorithm consumes more time than T_{budget}, the remaining nodes are simply culled against the view-frustum and sent to the rendering pipeline. Furthermore, once a node is detected to be entirely within the view-frustum, all leaves of this node can directly be sent to the rendering pipeline without further view-frustum culling the nodes inbetween.

Depth Ordered Culling: Front-to-back, or depth sorted order of the visibility tests produces an optimal filling of the virtual occlusion buffer. Therefore, it is important to process objects in depth sorted order. The z_{min} and z_{max} values for each bounding volume are returned by the view-frustum test for free. The bounding volumes interfering within the view-frustum are sorted by their z_{min} value into a *Depth-List*.

Our occlusion culling step tags each node as *visible* or *possibly occluded*. During motion, those tags have to be generated for every frame. As soon as the camera stops, only bounding volumes, in the previous iteration determined as possibly occluded are progressively refined. Nodes earlier marked visible will stay visible and can therefore be skipped, except for leaves - their polygons are directly sent to the rendering pipeline. This scheme changes once we determine a bounding volume to be visible, which has previously been marked as possibly occluded. In this case, we have to perform occlusion culling for all following nodes in the Depth-List, due to the changed visibility in the image.

Overall Refined Algorithm: To integrate these additional features, the basic algorithm is modified. The Depth-List is initialized with the visible child nodes of the uppermost visible node in the snSP-Tree. Unless the time budget is not entirely consumed, the head element of the Depth-List is transferred to our cull test. In an interleaved manner, view-frustum culling and occlusion culling for a single frame are performed as described in the following pseudo-code.

```
InitDepthList();

while (UsedTime < Budget)
  Node = DepthList->getHead();
  if (OccTest(Node) == VISIBLE)
    if (node == LEAF)
      render(Node->polygons);
      continue;
    forall children(Node)
      if (ViewFrustumTest(child) == VISIBLE)
        DepthList->add(child);
```

One advantage of this interleaved culling scheme is the reduced cost for sorting. For a well balanced snSP-Tree of depth twelve, we measured for the cathedral scene an average list length of eight and a maximum of 17.

4 Analysis

We examined our algorithm by processing four different scenes. One architectural scene of an array of gothic cathedrals, a city scene, a forest scene to demonstrate adaptive culling, and - similar to [24] - the content of a virtual garbage can of rather small objects.

In this Section, we discuss the performance of our algorithm on the test scenes described in Table 1. Note, the achieved percentage of model culled depends on the granularity of the snSP-tree. The more the individual objects of a scene are subdivided, the higher is the potential culling performance. Nevertheless, a higher culling performance does not imply a higher rendering performance. While culling up to 99% of many scenes is possible, the overall rendering performance would drop in most cases.

All measurements were performed rendering 650×650 images on a SGI O_2 workstation with 256 MB of memory and one 175 MHz R10000 CPU.

scene	#triangles	#objects	#triangles/object
cathedrals	3,334,104	8	416,763
city	1,056,280	300	3521
forest	452,981	12 + 1	28,500 + 110,981
garbage	5,331,146	2,500	about 2,100

Table 1: Model sizes.

4.1 Performance of the Algorithm

4.1.1 Cathedral Scene

Cathedral Scene

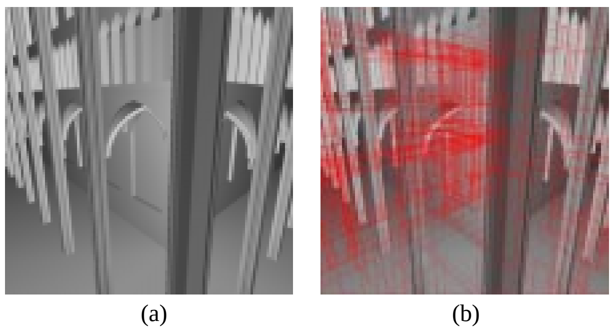

Figure 4: (a) Interior view of cathedral. (b) Bounding volumes of culled objects are marked in red.

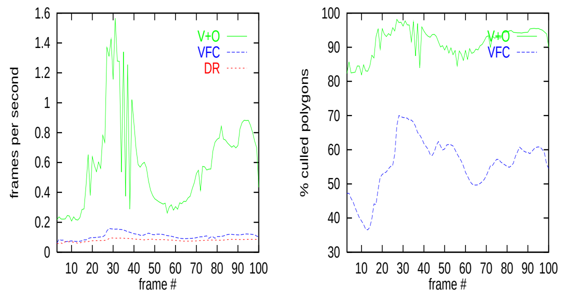

Figure 5: Frame rate and percentage of model culled: V+O denotes view-frustum culling and occlusion culling, VFC denotes view-frustum culling only, and DR denotes direct rendering without any culling.

In this scene, eight gothic cathedrals are aligned on a $2 \times 2 \times 2$ grid, where each cathedral consists of 416,763 polygons (Figure 11).

According to Section 3, we perform two different cull phases: First, a view-frustum culling (VFC); second, an occlusion culling. Figure 5 shows frame rate and percentage of model culled of our algorithm on the cathedral scene for a sequence of about 100 frames. For our performance tests, we measured three different modes: **direct rendering** (DR) - without any culling, **view-frustum culling** only (VFC), and **view-frustum and occlusion culling** (V+O).

The view-frustum-only mode culls only small portions of the eight cathedral model, because for most view points the other cathedrals are still within the view-frustum. However, occlusion culling is far more successful. Up to 65% of the model are culled away. Due to the visibility culling, we obtained an average speed-up of seven.

4.1.2 City Model

The city model is constructed out of three-hundred buildings. Each building contains some interior furniture.

City Model

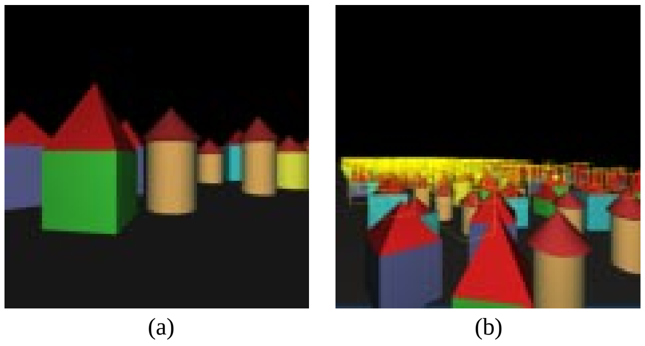

Figure 6: City model is rendered using V+O culling: (a) Visitor's perspective. (b) Bird's perspective of visitor's view - all yellow bounding volumes are not rendered due to visibility culling. Only 0.2% of the geometry is actually rendered at an average frame rate of two frames per second, if the view point is near the ground.

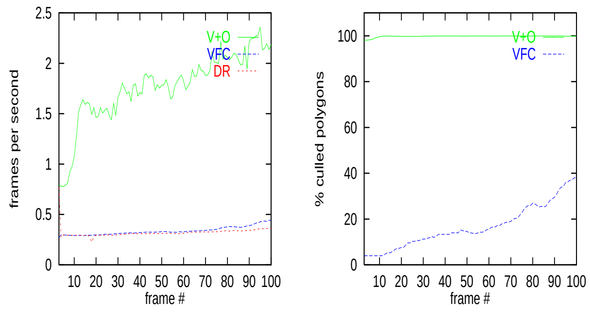

Figure 7: Frame rate and percentage of model culled: V+O denotes view-frustum culling and occlusion culling, VFC denotes view-frustum culling only, and DR denotes direct rendering without any culling.

Figure 7 shows frame rate and percentage of model culled of the city model. Three culling modes were measured while rendering a sequence of 100 frames: **direct rendering** (DR) - no culling, **view-frustum culling only** (VFC), and **view-frustum and occlusion culling** (V+O).

4.1.3 Forest Scene

The forest scene compounds of 12 leaf tree objects - each consists of 28,500 polygons - and one "Castle del Monte" of 110,981 polygons behind the trees (Figure 8). The scattered, yet dense occluded structure of the leaf trees has special demands for a visibility algorithm. Depending on the subdivision of those trees, we achieve higher additional culling, due to adaptive culling; Figure 9 shows an average additional reduction of 11% of the geometry using adaptive culling (AC), compared to the usual V+O culling of our algorithm.

Forest Scene

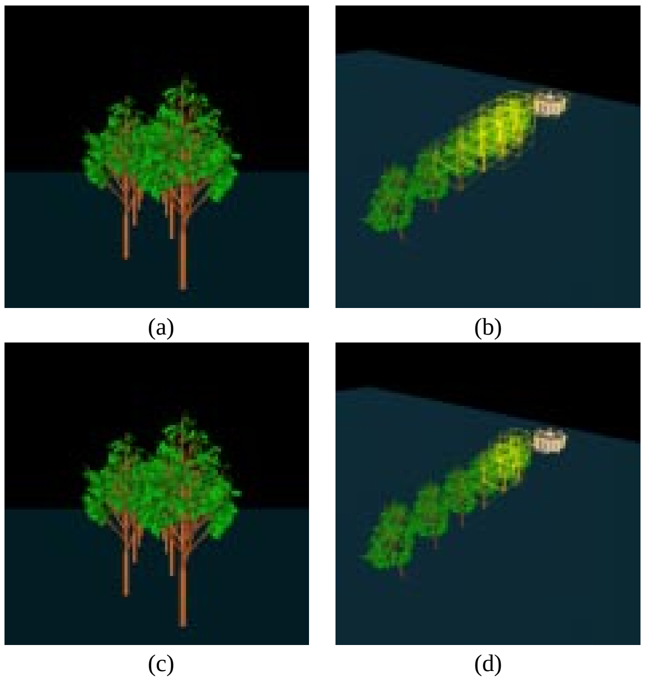

Figure 8: The forest scene is rendered using adaptive culling which culled 88% of the structure: (a) Front view. (b) Overview - all culled bounding volumes are marked yellow. (c) The forest scene is rendered using V+O culling which culled 77% of the structure. (d) Overview - all culled bounding volumes are marked yellow.

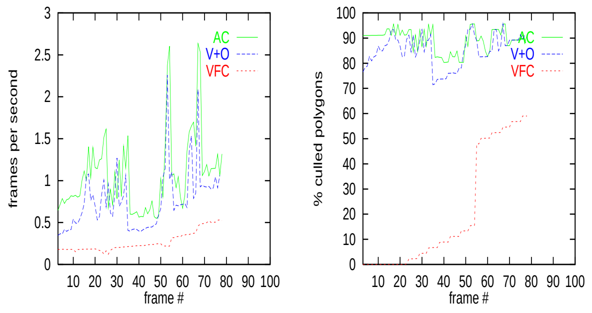

Figure 9: Frame rate and percentage of model culled: V+O denotes view-frustum culling and occlusion culling, VFC denotes view-frustum culling only, and AC denotes adaptive culling.

4.1.4 Virtual Garbage Can Scene

To cull dynamic scenes, a special mode can be used. Only the leaf level of the snSP-tree is used to check the visibility. We show the performance of this mode on a scene of the content of a virtual garbage can (Figure 10). 2,500 independent, potentially moving objects of an average size of about 2,100 polygons are contained in the scene. 96% of the total 5,331,146 polygons are culled. The average obtained speed-up is still larger than seven.

Virtual Garbage Can Scene

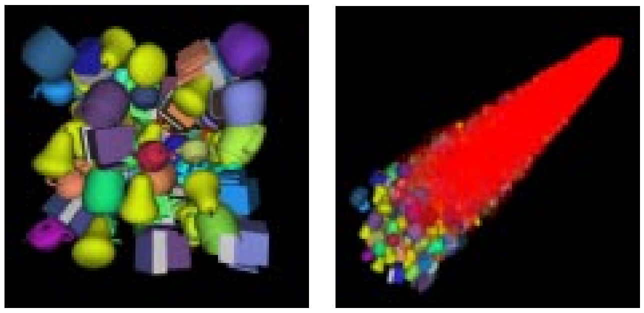

Figure 10: (a) Front view. (b) Bird's perspective of front view - all red bounding volumes are not rendered due to visibility culling. Direct rendering took more than 28 seconds, while rendering using our algorithm took less than four seconds.

scene	#triangles	culling	speed-up
cathedrals	3,334,104	91.3%	4.2
city	1,056,280	99.8%	4.8
forest AC	452,981	89.0%	3.8
V+C	452,981	84.7%	2.6
garbage	5,331,146	96.0%	7

Table 2: Average performance of OVC-algorithm compared to view-frustum only culling. The forest scene reflects comparison of adaptive culling (AC) and V+O culling to view-frustum culling.

4.2 Discussion and Comparison

In this Section, the pros and cons of our algorithm are discussed. In particular, we compare our OpenGL-assisted Visibility Culling algorithm (OVC) with three approaches for general visibility[2].

4.2.1 Hierarchical Z-buffer (HZB)

The HZB-algorithm combines a view-frustum culling of the octree block cubes with a z-value image pyramid for visibility queries [14]. In detail, each octree block cube is scan-converted into the z-buffer. Thereafter, all z-values that have changed are propagated through the image hierarchy. By checking the appropriate z-pyramid level, the visibility of the cube can be determined. If the cube is not visible, all the octree blocks which are children of the block associated with this cube, are not visible too. If the cube is visible, the children are recursively checked, until all children are either not visible or their associated polygons are rendered.

As mentioned before, we consider the basic idea of this algorithm as parent of our approach. However, we vary some of the ideas. In contrast to the HZB, we use a sloppy nSP-tree as object-space subdivision scheme. Furthermore, we use the OpenGL-framebuffer as virtual occlusion buffer to determine the occlusion of the tree elements; no image-space hierarchy is used to speed-up the queries.

The occlusion query is a weak point of the HZB-algorithm. Although it is tuned using a z-pyramid, basically an expensive array search through several pyramid levels is performed. Our OVC-algorithm only checks portions of the virtual occlusion buffer for the footprints of our subdivision nodes. This operation is fast, while achieving an equal culling performance.

Greene proposed the use of a "Z-query" feature to speed-up the performance of the occlusion test [14]. Unfortunately, this feature is not implemented on common graphics hardware.

To summarize, the close relationship to the HZB-algorithm enables the OVC-algorithm to combine many advantages of the HZB (i.e. temporal coherence, high model reduction due to culling) with our more efficient occlusion query on common OpenGL-graphics hardware.

4.2.2 Hierarchical Tiling using Coverage Masks (HT)

The hierarchical tiling approach [12] basically differs from HZB in two ways. While the HZB-algorithm uses a z-pyramid to establish the visibility stage of a particular subdivision block, this algorithm uses a hierarchical tiling algorithm. After tiling the silhouette of a subdivision block using pre-computed coverage-masks, the silhouette is tested against the current coverage pyramid of already rendered (and tiled) scene polygons of visible blocks. If the current block is visible, all its polygonal content is tiled into the coverage pyramid. This occlusion test requires a strict front-to-back traversal of the scene. To obtain this order, the spatial subdivision hierarchy is an octree of BSP-trees, which contain the front-to-back sorted scene polygons.

Although Greene pointed out some strategies for dynamic scenes [12], the previously mentioned strict front-to-back order limits the applicability for dynamic scenes.

Performance comparison of the HT visibility algorithm to our algorithm is difficult, because only a 4096$\times$4096 image of a really large dataset (167 million quadrilaterals) in a non-interactive environment was presented. Our models are much smaller, yet can be treated with a frame rate which enables some interactivity. However, we believe that the initial overhead filling the coverage pyramid is significant, especially for smaller scenes. Furthermore, the OVC-algorithm does not require a front-to-back order to establish the visibility, though it would produce better occluders. The rendering of high-resolution images is the domain of the hierarchical tiling algorithm. The occlusion query of our algorithm depends strongly on the size of the virtual occlusion buffer; hence, it performs less efficient generating high-resolution images.

4.2.3 Hierarchical Occlusion Maps (HOM)

The HOM-algorithm uses objects with high occlusion potential to generate a hierarchical occlusion map. For each frame, a bounding volume hierarchy of the remaining scene is checked against this occlusion map for overlap. If they do not overlap, the geometry of the volume is visible. If they do overlap, occlusion is determined in a conservative depth test [24].

[2]We encountered several problems comparing the numerous approaches. A serious, yet simple problem is the use of proprietary models to demonstrate the performance of the individual algorithms. Defining and providing a set of standard models would allow a better comparison of different algorithms.

In contrast, our OVC-algorithm does not use an occluder selection to initiate occlusion. Instead, the bounding volumes of the subdivision nodes are tested directly against the relevant parts of the virtual occlusion buffer, which represents the current visible objects. Effectively, this leads to a dynamic occluder selection from the complete scene database.

To compute the image hierarchy of occlusion maps, texture mapping hardware of high-end graphics workstations is exploited to speed-up the necessary filter-operation. For the OVC-algorithm, no image hierarchy is necessary. In addition, we do not use any special graphics hardware, besides the basic OpenGL-supported features[3].

The OVC-algorithm is closely related to the HZB-algorithm [14]. Therefore, the advantages of this algorithm compared to the HOM-algorithm are valid for our algorithm too, like the less conservative culling and possible exploitation of temporal coherence.

For approximate culling, Zhang et. al. claim to increase the culling performance of the algorithm [24]. Unfortunately, no data was presented to support that claim. Adaptive culling, our alternative to approximate culling, obtains on average an additional reduction of 5% of the geometry. On our test scene for adaptive culling, the forest scene, the average frame rate was increased by a quarter of a frame per second.

A drawback of our algorithm is its inability to treat non-polygonal objects, like textures. The algorithm is strictly polygon oriented; therefore, only textures closely associated with a polygon can be treated.

To compare the performance, note that the performance measurements of the HOM-algorithm were performed on a SGI Infinite-Reality graphics, which is about ten times more powerful than the O_2, and on a SGI MaximumImpact graphics, which is about three times more powerful than the O_2.

Additionally, our models are of significant higher polygon count than the HOM-algorithm models. Considering these different modalities, we are approximately two times faster for the city model. All other models can not fairly be compared, due to different topology and motion (garbage model).

4.3 Extending OpenGL

There are many limiting factors of current OpenGL to visibility culling. Probably most important is the lack of a distinctive visibility culling stage in the rendering pipeline.

The way we determine whether a subdivision node is visible, depends very much on the actual implementation of the virtual occlusion buffer, in our case the stencil buffer. Right now, we check the relevant part of the stencil buffer in a special interleaved mode for the identifier of this node. In many cases, this node is not visible. Therefore, it takes a long time to establish its visibility state.

We propose an extension to OpenGL to implement the virtual occlusion buffer in basically three ways:

- **Occlusion buffer.** Reading the framebuffer is a costly operation hence, we need a buffer query which is faster than the standard framebuffer read operation.

- **Footprint flag.** Most effort is spent checking the buffer for a modification, since the last action. Adding a modification flag to the virtual occlusion buffer would improve the performance tremendously.

- **Footprint counter.** Adaptive culling requires a measure how much of an object is visible, i.e. a building through a hole in a wall. The number of modified footprints of the virtual occlusion buffer could be such a measure. Extending OpenGL by this feature would simplify this task greatly.

[3]Currently, we use the stencil buffer as implementation of the virtual occlusion buffer.

The extension proposed in this Section is very simple and easy to implement in hardware. We hope that the members of the OpenGL Architecture Review Board are aware of the importance of the problem and consider a reasonable extension to OpenGL.

Recently, HP proposed an extension to OpenGL for visibility queries [15]. Although only a preliminary version without any details of the proposal is available, it underlines the growing importance of visibility culling for high-performance rendering of large models.

5 Conclusion and Future Work

In this paper we have presented a visibility culling algorithm based on core OpenGL functionality. By combining different framebuffer features and sloppy n-ary Space Partitioning-trees - as model-space subdivision - significant culling performance and reasonable frame rates were obtained. On average, the culling performance exceeded 90%, while a rendering speed-up factor of almost five - compared to view-frustum culling - was achieved.

Furthermore, we proposed to add features for visibility culling to the OpenGL rendering pipeline. Specifically, we suggest to add an occlusion buffer, including a footprint flag, and a footprint counter.

There are quite a few areas for future work on this algorithm. Among the most important are

- **Multi-resolution:** Large model databases usually use multi-resolution methods to represent different levels of detail. Incorporating these level-of-detail functionality is important for the rendering of large-scale scenes.

- **Parallelization:** Using a multi-threaded implementation for the visibility queries promises faster traversal of our model-space snSP-tree. However, using multiple threads for the processing of OpenGL primitives adds a potential bottleneck into our visibility stage.

- **Optimization of the subdivision scheme:** Different scenes of different topology require subdivision schemes of different topology and depth. However, optimizing these structures is not a trivial task and needs further investigation.

Besides these areas for future work, it seems promising to exploit the presented selection-buffer-based frustum culling for visibility queries based on shadow-frusta. Unfortunately, not all possible frusta can be specified by the projection matrix of OpenGL. However, shadow-frusta of less suited occluders could be approximated by assembling square footprints which can be specified using the matrix stack of OpenGL.

References

[1] J. Airey, J. Rohlf, and F. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In *Symposium on Interactive 3D Graphics*, pages 41–50, 1990.

[2] R. Brechner. Interactive walkthroughs of large geometric databases. In *SIGGRAPH'96 course notes*, 1996.

[3] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, 1974.

[4] D. Cohen-Or and E. Zadicario. On-line conservative ϵ-visibility culling for client/server walkthroughs. In *Late Breaking Hot Topics Proceedings, IEEE Visualization'97*, pages 37–40, 1997.

[5] S. Coorg and S. Teller. Temporally coherent conservative visibility. In *Proc. of the 12th ACM Symposium on Computational Geometry*, 1996.

[6] S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. In *Proc. of the 1997 ACM Symposium on Interactive 3D Graphics*, pages 83–90,189, 1997.

[7] J. Foley, A. Van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, Reading, Mass., 2nd edition, 1996.

[8] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. In *Proc. of ACM SIGGRAPH*, pages 124–133, 1980.

[9] B. Garlick, D. Baum, and J. Winget. Interactive viewing of large geometric databases using multiprocessor graphics workstations. In *SIGGRAPH'90 course notes: Parallel Algorithms and Architectures for 3D Image Generation*, 1990.

[10] J.E. Goodman and J. O'Rourke. *Handbook of Discrete and Computational Geometry*. CRC Press, Boca Raton, 1997.

[11] N. Greene. *Hierarchical Rendering of Complex Environments*. PhD thesis, Computer and Information Science, University of California, Santa Cruz, 1995.

[12] N. Greene. Hierarchical polygon tiling with coverage masks. In *Proc. of ACM SIGGRAPH*, pages 65–74, 1996.

[13] N. Greene and M. Kass. Error-bounded antialiased rendering of complex environments. In *Proc. of ACM SIGGRAPH*, pages 59–66, 1994.

[14] N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. In *Proc. of ACM SIGGRAPH*, pages 231–238, 1993.

[15] Hewlett-Packard. Occlusion test, preliminary. http://www.opengl.org/Developers/Documentation/Version1.2/HPspecs/occlusion_test.txt, 1997.

[16] L. Hong, S. Muraki, A. Kaufman, D. Bartz, and T. He. Virtual voyage: Interactive navigation in the human colon. In *Proc. of ACM SIGGRAPH*, pages 27–34, 1997.

[17] T Hudson, D. Manocha, J. Cohen, M. Lin, Kenneth E. Hoff, and H. Zhang. Accelerated occlusion culling using shadow frusta. In *Proc. of ACM Symposium on Computational Geometry*, 1997.

[18] D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proc. of ACM Interactive 3D Graphics Conference*, 1995.

[19] C. Mueller. Architectures of image generators for flight simulators. Technical Report TR95-015, Department of Computer Science, University of North Carolina, Chapel Hill, 1995.

[20] B. Naylor. Partitioning tree image representation and generation from 3d geometric models. In *Proc. of Graphics Interface'92*, pages 201–212, 1992.

[21] O. Sudarsky and C. Gotsman. Output-sensitive visibility algorithms for dynamic scenes with applications to virtual reality. In *Proc. of Eurographics'96 conference*, pages 249–258, 1996.

[22] S. Teller and C.H. Sequin. Visibility pre-processing for interactive walkthroughs. In *Proc. of ACM SIGGRAPH*, pages 61–69, 1991.

[23] R. Yagel and W. Ray. Visibility computations for efficient walkthrough of complex environments. *PRESENCE*, pages 1–16, 1996.

[24] H. Zhang, D. Manocha, T. Hudson, and Kenneth E. Hoff. Visibility culling using hierarchical occlusion maps. In *Proc. of ACM SIGGRAPH*, pages 77–88, 1997.

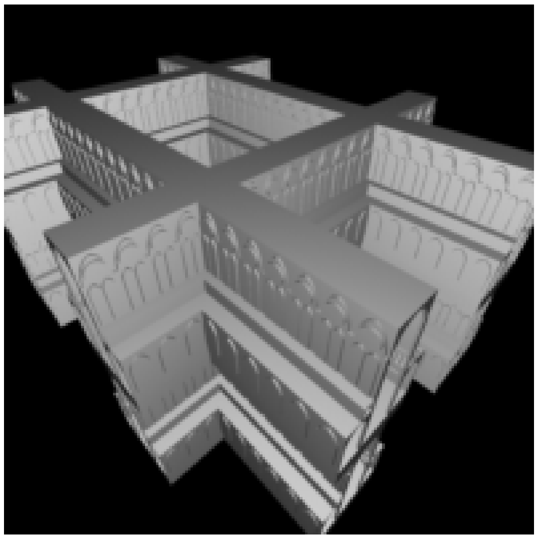
Figure 11: Overview of the cathedral scene.

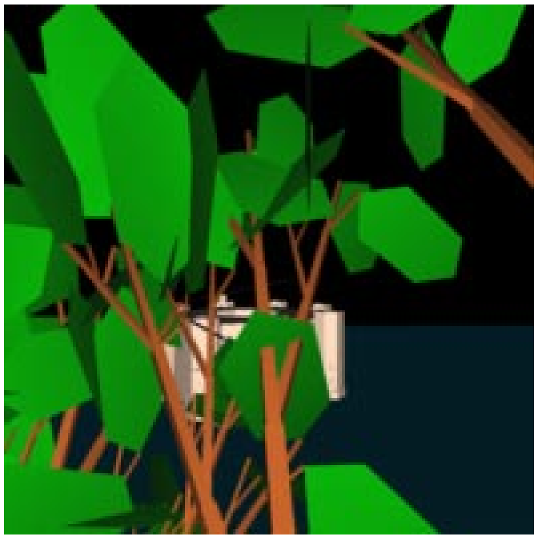
Figure 12: "Suddenly, the old castle appeared behind the branches of the trees ....."