OpenSG Symposium (2003)
D. Reiners (Editor)

An Occlusion Culling Toolkit for OpenSG PLUS

D. Staneker†

WSI/GRIS, University of Tübingen, Germany

Abstract

Image-space occlusion culling is an useful approach to reduce the rendering load of large polygonal models in scientific computing, mechanical engineering, or virtual medicine. Like most large model techniques, occlusion culling trades overhead costs with the rendering costs of the possibly occluded geometry.
In this paper we present an occlusion culling toolkit for OpenSG. The toolkit includes three different image space techniques utilizing graphics hardware. These techniques are supported by other software techniques to optimize the occlusion culling queries. All techniques are conservative and usable with dynamic scenes, because no precomputing is necessary.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Picture/Image Generation]: Viewing Algorithms, Occlusion Culling; I.3.4 [Graphics Utilities]: Application Packages, Graphics Packages; I.3.7 [Three-Dimensional Graphics and Realism]: Hidden Line/Surface Removal;

1. Introduction

Faster visualization of large datasets in scientific computing, mechanical engineering, or virtual medicine are in the focus of several techniques. Most of them reduce the number of polygons, others are using sampling techniques like point sampling or ray tracing. To reduce the number of polygons, level-of-detail [5] or impostor techniques can be used. Occlusion culling is another approach for faster visualization of large datasets. Hereby hidden parts of a scene are detected and excluded from the rendering process. In this paper we will present a toolkit for occlusion culling for OpenSG. We will present the base techniques of the toolkit to calculate occlusion and further approaches to enhance these base techniques.

OpenSG[13] is a portable scene graph toolkit with the focus on real time rendering. With the OpenSG PLUS project, OpenSG will be enhanced with Large Scene Support, High Level Primitives and High Level Shading. The presented occlusion culling techniques are part of the Large Scene Support.

This paper is organized as follows; the next section briefly reviews related toolkits for visualization and occlusion culling techniques. The third section describes the toolkit and its features. Results are shown in Section 4, followed by the conclusions.

2. Previous Work

Scene graph programming toolkits are widely available, e.g. Open Inventor, IRIS Performer, Cosmo3D, but most of them have no support for occlusion culling. One of the scene graph programming toolkits, which have support for occlusion culling is Jupiter[8, 2]. Jupiter focuses on large model rendering and provides different concepts to manage large amount of data. For occlusion culling Jupiter uses only the HP Occlusion Flag[7].

A lot of occlusion culling algorithms are available. Cohen-Or et al.[3] give a recent overview on the various occlusion culling techniques. While they can be classified in object-space[4] and image-space techniques, we are focusing on image space-techniques, but an object-space technique is also available in the toolkit for OpenSG. Some of the occlusion culling techniques need extensive preprocessing or they need special scenes. These are not in the scope of this paper. In the taxonomy of Cohen-Or et al.[3] we are focusing on conservative, from-point image-space approaches for generic scenes and generic occluders with extra support for object-space shadow frustra[11].

† staneker@gris.uni-tuebingen.de

One of the well known image-space algorithm is the Hierarchical z-Buffer[6], which uses hierarchical data structures for the depth buffer and the scene. Other image-space algorithms are occlusion maps[18] or virtual occlusion buffers[1]. Many other algorithms are available, some of them are using OpenGL to accelerate occlusion calculations, like the following discussed in the next section.

3. Our proposal in detail

3.1. Base Functionality

The base functionality provides some generic, image-space algorithms to get visibility information of an given object. All of them are using the hardware in some way to get visibility information.

The approach is always the same. The occlusion test is initialized (e.g. disabling z-buffer writes), then multiple occlusion queries can be performed (each request gets an index) and after all queries the results can be requested with the corresponding index. There is no restriction in the geometry of the bounding volume for the occlusion test, however we are using only the bounding boxes from the scene graph for the tests. No precomputing to get a special hierarchy or special data structures is needed, only the features and data structures of the OpenSG scene graph are used, thus dynamic scenes are also supported.

The following techniques build the base occlusion test techniques[16] for the toolkit.

3.1.1. OpenGL Extensions for Occlusion Culling

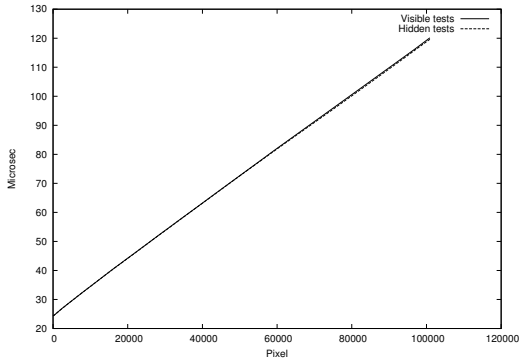

Figure 1: *Latency for the HP Occlusion Flag on an Intel P4@2400MHz with a NVidia GeForce4Ti 4600 running Linux.*

The HP Occlusion Flag[7] is a hardware extension, which returns information of the visibility of an object. The idea is, to render a bounding volume through the pipeline with disabled color- and z-buffer writes. If at least one pixel of the bounding volume triggers a z-buffer write, the HP Occlusion Flag is set to true otherwise to false. If the result is true (at least one pixel of the bounding volume is visible), the content of the bounding volume has to be rendered. The HP Occlusion Flag provides a very easy and one of the fastest ways for doing occlusion culling, but has the drawback, that each request to the HP Occlusion Flag is synchronous. A new request can only be started after the finish of the previous one. This problem is adressed by the HP Visibility Extension[9] and by the more well known NVidia Occlusion Query[12]. Both extensions support multiple queries at once. Additionally, the NVidia extension returns the amount of visible pixels of each tested bounding volume instead of a simple flag. This can be used for Level-of-detail selection or screen space culling.

The HP Occlusion Flag was introduced with the HP Visualize fx[15] graphics subsystem. It is also available beside the NVidia Occlusion Query extension on NVidia Geforce3/4Ti or newer graphics subsystems. Modern OpenGL 2.0 or DirectX 9.0 capable systems like the ATI Radeon 9700 or the NVidia Geforce fx have to support the occlusion extensions. Both extensions, NVidia and HP, are now supported by the toolkit for OpenSG.

The performance of the hardware extensions depends on the fillrate of the z-buffer. Larger bounding volumes need more time for the test, because the whole bounding volume passes always the z-buffer stage of the rendering pipeline. Figure 1 shows the correlation between the size of a bounding volume in screen-space and the latency for an occlusion test request. With enabled backface culling the test is almost twice as fast as without, because with backface culling only one scan through the z-buffer for the front-face is done. The graphics hardware rasters always the complete bounding volume, but the rasterization could be stopped after the first visible pixel, when using the HP extension. With the NVidia extension, the hardware has to raster always the whole bounding volume to determine the full amount of visible pixels. This is a drawback, especially for large bounding volume. We address this problem with the software depth buffer (Section 3.2.2) and the Occupancy Map (Section 3.2.1.)

3.1.2. Using the Stencil Buffer

Bartz et al.[1] described a technique that the stencil-buffer can be used to compute visibility informations. During rasterization writing to the frame- and z-buffer is disabled. For each pixel of the bounding volume the z-buffer test is applied. If the pixel would be visible, a value is written to the stencil-buffer (see Figure 2) by using `glStencilOp()`. After rasterizing the bounding volume, the stencil-buffer is read and sampled in software. Occluded bounding volumes will not contribute to the z-buffer, hence will not cause a respective entry in the stencil-buffer.

The actual implementation reads the whole region of the covered zone by the bounding volume. This could be optimized like the fragments in Section 3.1.3 or with the inter-

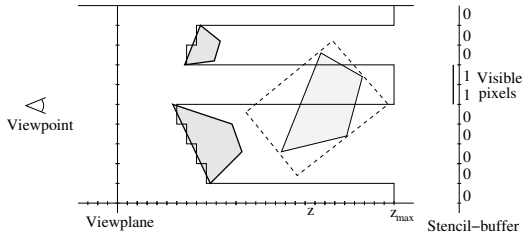

Figure 2: *Occlusion test with the stencil-buffer.*

leaving scanning scheme from Bartz et al.[1]. Multiple queries are possible, if the stencil buffer supports more than one bit. Also the amount of visible pixels can be counted, but usually the test is stopped after the first or a necessary amount of visible pixels.

3.1.3. Hardware Depth Buffer

The OpenGL z-buffer can be used to get the visibility information of an bounding volume, since it always holds the correct depth-value for every pixel. To test occlusion, the depth-values of the bounding volume are computed and tested against the values of a z-buffer maintained in software. A `glReadPixels()` to read the OpenGL z-buffer is quite expensive, hence this operation is split in fragments. Each fragment has the same size, which is a multiple of the databus width to exploit memory alignment on the graphics subsystem. A fragment is only read, if it is necessary for a pixel test[10]. The test stopps after at least one or a necessary amount of visible pixel.

Every fragment holds two flags, an *invalid* and an *unused* flag. At the beginning of every frame all the *unused* flags are true and a tested pixel against this fragments leads always to a visible pixel without reading the OpenGL z-buffer. If a pixel is visible, the *invalid* bits of the corresponding fragments are enabled, because the geometry of the bounding volume will be rendered and the content of the z-buffer may change. For pixels inside fragments with a true *invalid* bit, we read the z-buffer and disable the *invalid* bit.

In many scenes it is not necessary to render every detail. For this approach a minimum of visible pixels for a bounding box can be set. Only if at least this minimum of pixels is visible, the complete bounding box is set as visible. This leads to a speed-up with a miner reduction in rendering quality.

3.1.4. Traversal and Depth Sorting

All of our presented image space techniques are using the hardware z-buffer in some way for the occlusion test, thus accurate z-buffer values are needed to get correct occlusion results. To ensure accurate values, we are using a front-to-back sorted rendering of the geometry objects of the scene graph. Front-to-back sorting is not really necessary, but one of the simplest ways to render occluders first. The front-to-back sorting is done by the front-most corner of the object's bounding box, which leads to an adequate sorting for occlusion culling.

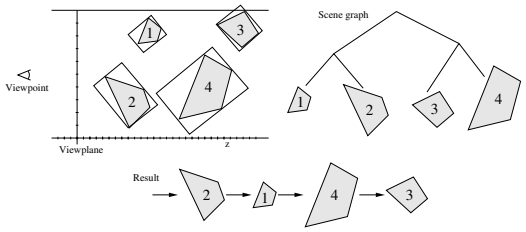

Figure 3: *Front-to-back sorted traversal.*[16]

For this paper we used our own traversal and depth sorting scheme due to the lack of depth-sorted traversal in first OpenSG releases, but this will change[14] in future releases. The traversal, sorting and rendering is realized in a pipelined fashion. First, the scene graph is traversed and the geometry nodes are collected. While they are collected, the depth sorting is realized with a hash function. After the traversal and sorting, the geometry nodes are rendered or occlusion culled in an interleaved step. Our traversal and depth sorting is a bottle neck in large scene graphs due to the use of the `OSG::DrawAction` and has space for a lot of optimizations, which is discussed in a later section.

3.2. Extensions

To optimize the use of the base functionality and to exploit occlusion information, we developed further techniques:

3.2.1. Occupancy Map

One problem of the approaches is, that they always have to read the hardware z-buffer in some way, but from viewpoints with low occlusion a lot of occlusion tests are unnecessary, because they return a visible result, so that the rendering of the corresponding geometry gets more expensive. The Occupancy Map[17] is a small data structure which manages occupied regions of the screen space. Only in regions with already rendered (occupied) geometry, an occlusion test makes sense. In not occupied regions an occlusion test will always return visible pixels due to the lack of occluding pixels. The Occupancy Map saves these unnecessary accesses to the hardware z-buffer.

The Occupancy Map is realized only as a small bit field. Each bit represents an occupied or unoccupied region. Storing depth values or other information is not necessary, because the requests occur in a depth sorted order. The Occupancy Map is updated with the bounding boxes of the rendered objects, which is not exact, but conservative.

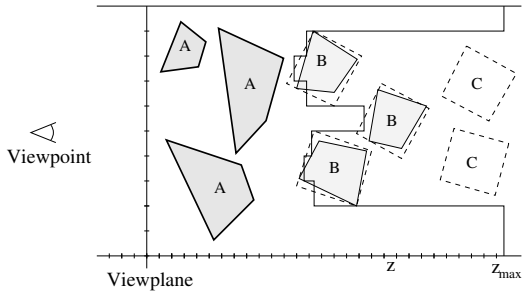

Figure 4: *Occlusion test with the software depth buffer; (A) visible geometry, (B) tested with an OpenGL test, (C) tested by the software approach.*

3.2.2. Software Depth Buffer

For scenes with high depth complexity, occlusion tests can be saved by a software implementation of a depth buffer. Rendering of the scene geometry in software is too expensive, but hidden bounding boxes can be used as an approximation. Thus we are rendering the bounding boxes of previous hardware occlusion tests into the software depth buffer. Before another bounding box is tested by the hardware tests, it can be tested with the software depth buffer. Fillrate for the software depth buffer can be saved by a lower resolution than the hardware depth buffer. If the application knows occluder, they can be rendered into the software depth buffer before starting any other tests.

3.2.3. Shadow Frustra

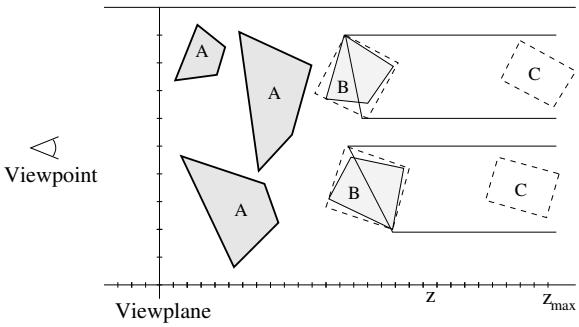

Figure 5: *Occlusion test with Shadow Frustra. (A) Visible geometry, (B) tested with an OpenGL test, (C) tested by the software approach.*

Shadow Frustra are presented by Hudson et al.[11]. In contrast to the other used techniques in OpenSG, Shadow Frustra are working in object space. The technique can be used in two different ways. Usually the application defines some shadow frustra of known (virtual) occluders. The shadow frustrum itself is defined by multiple sets of planes. Each set defines an inner or outer region. The bounding box of an occludee has to be complete inside an inner region, otherwise it is not occluded. Another way is to define the shadow frustra automatically. We are using shadow frustra of hidden bounding boxes (see Figure 5) from previous image space occlusion tests. This is a very simple approach and not as powerful as the software depth buffer because of the lack of occluder fusion, but a lot of fillrate can be saved, if the application knows large occluders. See Figure 13 for such a scene with a large occluder in the front.

3.2.4. The Toolkit

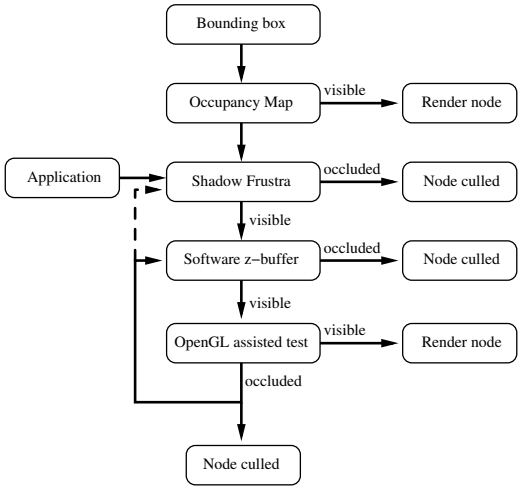

Figure 6: *Architecture of the occlusion culling process.*

The occlusion culling toolkit provides an abstract, object-oriented access to the before described occlusion culling approaches. Multiple queries are available through the methods, however they are internally synchronized if the underlying technique doesn't support multiple queries, like the HP Occlusion Flag. The access to an occlusion test class looks as follows:

A. Setup of global configurations for all tests and frames, like used viewport, resolution of buffers...
B.1 Frame initialization of special values for each frame, e.g. software z-buffer clear or setting of flags for internal data structures.
B.1.1 Occlusion test initialization - state changes for occlusion test, e.g. disabling of z-buffer writes, stencil-buffer setup...
B.1.2 Occlusion test perform for a given bounding volume assigned to index *i*.
B.1.3 Occlusion test result returns the visibility information of bounding volume *i*.
B.1.4 Occlusion test exit restores state changes from B.1.1, update flags for next tests...
B.2 Frame exit - memory cleanups...

All techniques can be combined and used at the same

time. Not every combination makes sense. E.g. the hardware z-buffer algorithm does not benefit very much from a software z-buffer, because both use almost the same data structure and tests would be redundant. Also automatically generated shadow frustra from hidden bounding boxes in conjunction with the software depth buffer do not make sense, because both solve the same problem, but shadow frustra have no occluder fusion. However, shadow frustra defined by the application can work together with the software depth buffer.

Due to the very different characteristic of the available graphics boards and the data sets, the application has to decide, which approaches are used.

4. Results

For all our tests we used the OSGViewer application[16] with a cotton picker model (see Figure 14) and a camera path with 80 different frames. All frames were rendered at a high resolution of 1480×1016 with 24 bits color depth and three light sources.

In some frames, some parts of the scenes are view frustrum culled by the internal OpenSG view frustrum culler. The cotton picker is a large model with over 13,000 geometry nodes containing almost 11mio of polygons. We used a medium class PC with Intel P4@2.4 GHz and a NVidia Geforce4Ti 4600 running Linux for our measurements.

4.1. Traversal and Sorting

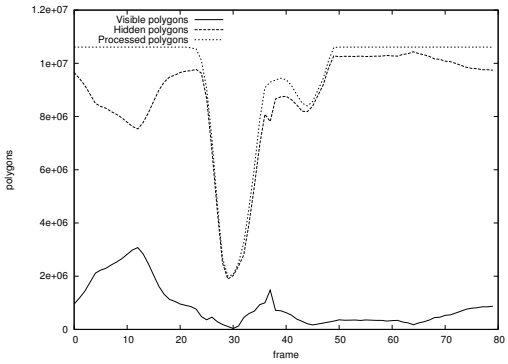

Figure 7: *Visible/hidden polygons of our camera path.*

First we analysed our traversal and sorting scheme. Due to the two step approach, first traversal and sorting, second rendering and occlusion culling, we can simply distinguish the time for traversal and sorting, and the time for rendering and occlusion culling. Figure 9 shows, that in many frames the rendering with occlusion culling needs less time than the traversal. An average time of 325 ms was measured for the cotton picker, this is equivalent to a limit of 3 frames per second without rendering. OpenSG can travers and render faster with the `OSG::RenderAction`, but it was not possible to use this class for occlusion culling without modifications to the OpenSG code base. The change to a better traversal scheme in the next OpenSG releases will close this problem in the future.

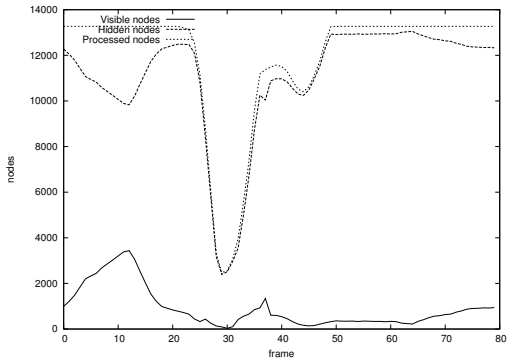

Figure 8: *Visible/hidden nodes of our camera path.*

In Figure 7 the amount of visible and hidden polygons (detected by a bounding box occlusion test) for each frame is shown. Approximately between frame 25 and frame 50 the view frustrum culler removes some nodes. An average of almost 880,000 polygons in 905 nodes are visible and 8,670,000 polygons in 11,000 nodes are hidden in our camera path. Frame 30 has the lowest complexity with 34,611 visible and 2,022,003 hidden polygons.

4.2. Rendering and Occlusion Culling

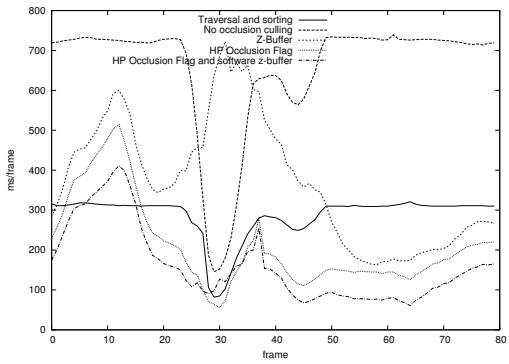

Figure 9: *Performance timings of the rendering with different techniques.*

Because of the slow traversal, we only timed rendering and occlusion culling to show the difference between the approaches:

We used a software z-buffer with a quarter resolution of the viewport for the tests. The performance of the z-buffer read depends strongly on the depth complexity because of the high cost of the read of the OpenGL z-buffer. The HP

	Rendering time	Speedup
No occlusion culling	654 ms	0%
With z-buffer read	374 ms	75%
With HP Flag test	212 ms	208%
With HP Flag + soft z-buffer	162 ms	304%

Table 1: *Comparison of the average performance timings.*

Occlusion Flag or the NVidia Occlusion Query with support from the software z-buffer gaves best results for the used camera path.

4.3. Occupancy Map and Software z-Buffer

Figure 10 shows the percentage of the savings of the Occupancy Map and the software z-buffer from all occlusion tests in the frames of our camera path. The Occupancy Map saves 19% of occlusion tests with a visible result and only 1% of all occlusion tests. In scenes with lower occlusion density, these values are higher. The software z-buffer saves 80% of occlusion tests with a hidden result and 75% of all occlusion tests. Of course, these values are lower in scenes with lower depth complexity.

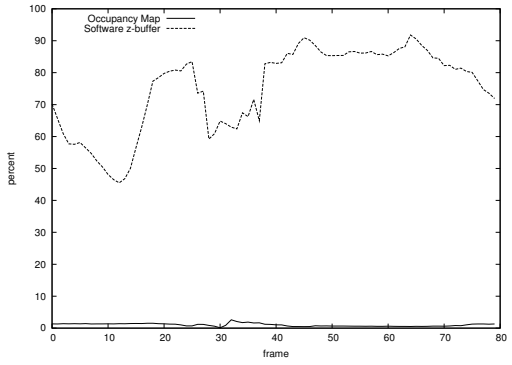

Figure 10: *Savings of extra occlusion tests by the Occupancy Map and the software z-buffer.*

5. Conclusions

In this paper we have presented the occlusion culling toolkit for OpenSG with the different approaches for doing occlusion culling. Different approaches are necessary to support the wide range of graphics subsystems. Momentary is the fastest way for doing occlusion culling the HP Occlusion Flag or the NVidia Occlusion Query in combination with the software z-buffer (above 300% speed-up.) Using these extensions without special assistance can be fast, but not as fast as with software z-buffer. However to support a lot of very different hardware platforms, all of the other techniques have their account.

5.1. Future Work

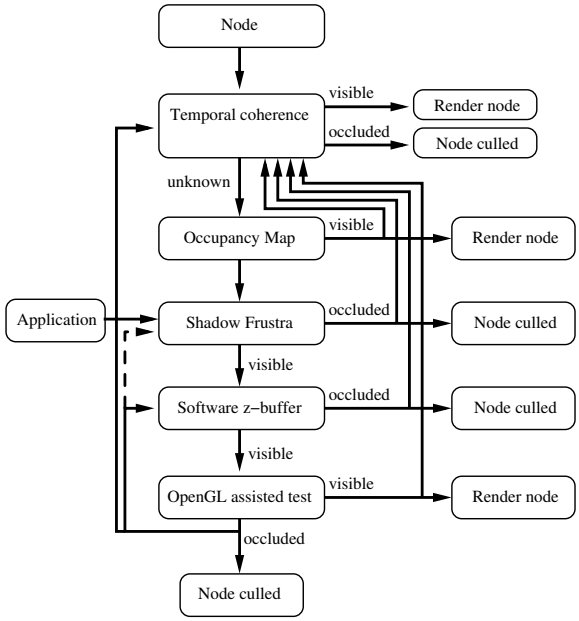

Figure 11: *Future architecture of the toolkit.*

Traversal and sorting has to be optimized, especially for very large scene graphs. In this paper we used our own, slow implementation. Future versions will use a more powerful and faster strategy of OpenSG. This will also enable hierarchical occlusion culling, which is not possible with the current scheme. Also state changes of the graphics pipeline have to be minimized to get further speed-ups.

Temporal coherence could also improve the rendering speed and could be easily integrated in the actual toolkit. This will be a major point of further development (see Figure 11).

Precomputing was out of the focus, because dynamic scenes without assumptions on the scene graph have to work. In further releases this could become a more interesting point to speed-up rendering of static or special scenes.

The tests are working in a serial fashion, but could easily be parallelized, so that the software techniques, like the shadow frustra or the software depth buffer are working parallel to the NVidia Occlusion Query or the HP Occlusion Flag. This would result in a better load balancing between the main processor and the graphics subsystem and thus, to higher frame rates.

An automatic selection of the techniques, so that the application do not have to take care about occlusion culling will be developed.

Acknowledgements

This work is supported by the OpenSG PLUS project of the bmb+f in Germany. The cotton picker dataset is a courtesy of Engineering Animation Inc.

We would like to thank Dirk Reiners, Gerrit Voss and Johannes Behr for their help in OpenSG programming.

References

1. BARTZ, D., MEISSNER, M., AND HÜTTNER, T. OpenGL-assisted Occlusion Culling of Large Polygonal Models. *Computers & Graphics 23*, 5 (1999), 667–679.
2. BARTZ, D., STANEKER, D., STRASSER, W., CRIPE, B., GASKINS, T., ORTON, K., CARTER, M., JOHANNSEN, A., AND TROM, J. Jupiter: A Toolkit for Interactive Large Model Visualization. In *Proc. of Symposium on Parallel and Large Data Visualization and Graphics* (2001), pp. 129–134.
3. COHEN-OR, D., CHRYSANTHOU, Y., DURAND, F., AND SILVA, C. Visibility: Problems, Techniques, and Application. In *ACM SIGGRAPH Course 4* (2000).
4. COORG, S., AND TELLER, S. Temporally Coherent Conservative Visibility. In *Proc. of ACM Symposium on Computational Geometry* (1996), pp. 78–87.
5. GARLAND, M. Multiresolution Modeling: Survey and Future Opportunities. In *Eurographics STAR report 2* (1999).
6. GREENE, N., KASS, M., AND MILLER, G. Hierarchical Z-Buffer Visibility. In *Proc. of ACM SIGGRAPH* (1993), pp. 231–238.
7. HEWLETT-PACKARD. Occlusion Test, Preliminary. http://www.opengl.org/Developers/Documentation/Version1.2/HPspecs/occlusion_test.txt, 1997.
8. HEWLETT-PACKARD. Jupiter 1.0 Specification. Tech. rep., Hewlett Packard Company, Corvallis, OR, 1998.
9. HEWLETT-PACKARD. HP Visibility Test. http://dune.mcs.kent.edu/~ farrell/distcomp/graphics/hpopengl/Reference/glVisibilityBufferHP.html , 1999.
10. HEY, H., AND TOBLER, R. F. Real-time occlusion culling with a lazy occlusion grid. In *Proc. of Eurographics Workshop on Rendering* (2001).
11. HUDSON, T., MANOCHA, D., COHEN, J., LIN, M., HOFF, K. E., AND ZHANG, H. Accelerated Occlusion Culling Using Shadow Frusta. In *Proc. of ACM Symposium on Computational Geometry* (1997), pp. 2–10.
12. NVIDIA. NVidia Occlusion Query. http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion_query.txt, 2001.
13. OPENSG-FORUM. OpenSG - Open Source Scenegraph. http://www.opensg.org, 2000.
14. REINERS, D. A Flexible and Extensible Traversal Framework for Scenegraph Systems. http://www.opensg.org/ OpenSGPLUS/symposium/, 2002.
15. SCOTT, N., OLSEN, D., AND GANNETT, E. An Overview of the VISUALIZE fx Graphics Accelerator Hardware. *The Hewlett-Packard Journal*, May (1998), 28–34.
16. STANEKER, D. A First Step towards Occlusion Culling in OpenSG PLUS. http://www.opensg.org/ OpenSG-PLUS/symposium/, 2002.
17. STANEKER, D., BARTZ, D., AND MEISSNER, M. Using Occupancy Maps for better Occlusion Culling Efficiency. Poster, Eurographics Workshop on Rendering, 2002.
18. ZHANG, H., MANOCHA, D., HUDSON, T., AND HOFF, K. E. Visibility Culling Using Hierarchical Occlusion Maps. In *Proc. of ACM SIGGRAPH* (1997), pp. 77–88.

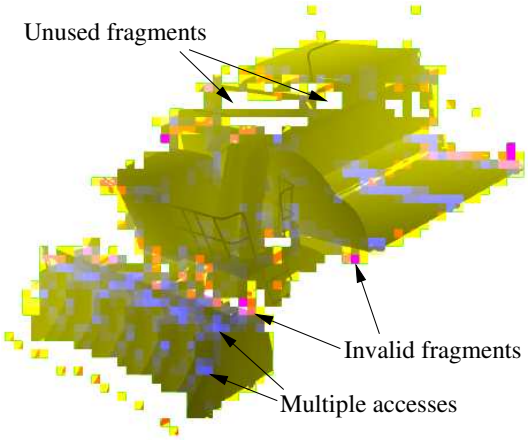

Figure 12: *Z-buffer with marked fragments.*

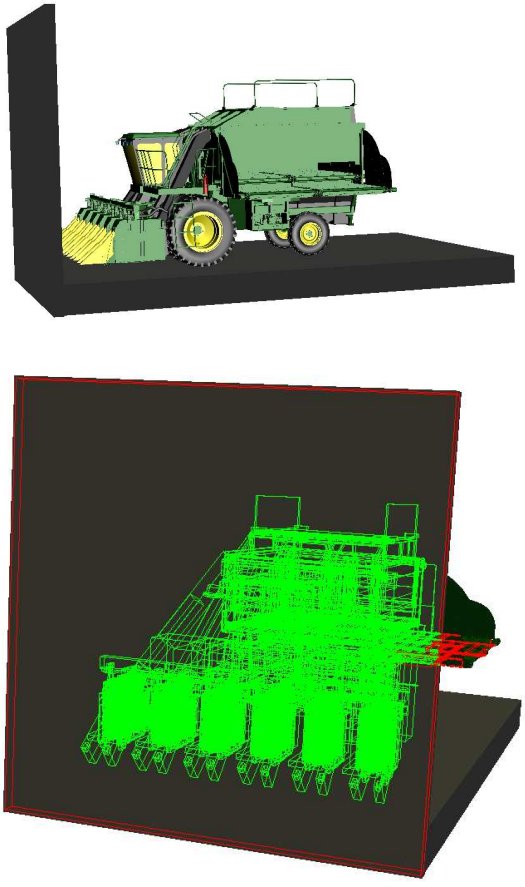

Figure 13: *Shadow Frustra test scene. The red boxes tested by the HP Occlusion Flag and the green ones with the shadow frustrum of the front box.*

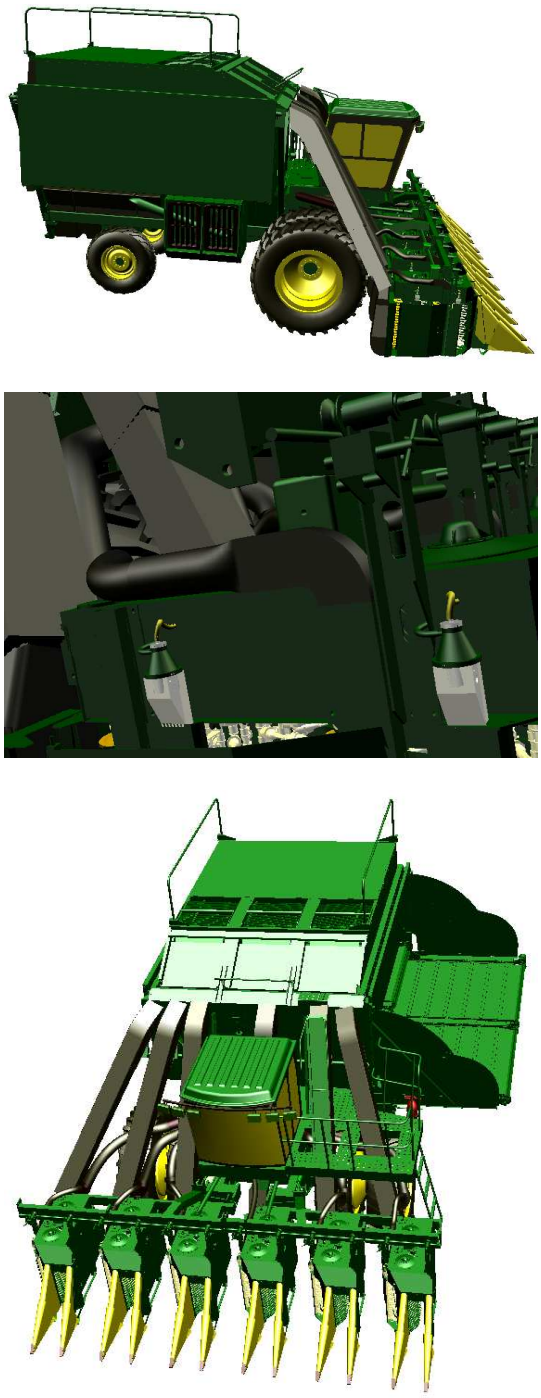

Figure 14: *Cotton picker from different viewpoints of the camera path.*