Extending Graphics Hardware For Occlusion Queries In OpenGL

Dirk Bartz Michael Meißner Tobias Hüttner

Computer Graphics Lab *
University of Tübingen

Abstract

For interactive rendering of large polygonal objects, fast visibility queries are necessary to quickly decide whether polygonal objects are visible and need to be rendered. None of the numerous published algorithms provide visibility performance for interactive rendering of large models.

In this paper, we propose an OpenGL extension for fast occlusion queries. Added after the depth test stage of the OpenGL rendering pipeline, our algorithm provides fast queries to establish the occlusion of polygonal objects. Furthermore, hardware aspects of this proposal are discussed and possible implementations on two different graphics architectures are presented.

CCS Categories: I.3.1 [Hardware Architectures]: Graphics processors, Raster Display Devices; I.3.3 [Picture/Image Generation]: View Algorithms; I.3.7 [Three-Dimensional Graphics and Realism]: Hidden Line/Surface Removal;

Keywords: Visibility, occlusion culling, hierarchical data structures, OpenGL.

1 Introduction

Hidden-line-removal and visibility are among the classic topics in computer graphics [4]. A large variety of algorithms are known to solve these visibility problems, including the z-buffer approach [14, 3], the painter algorithm [4], and many more.

Recently, visibility has been of special interest for walkthroughs of architectural scenes [1, 15] and rendering of large polygonal models [9, 5, 17]. Unfortunately, these approaches are limited to cave-like scenes [9], require not commonly available hardware support [7], or do not provide interactive rendering (more than 10 frames/second) of large models on mid-range graphics hardware [17].

We believe that an extension to graphic API's like OpenGL is essential to deal economically with large polygonal models in an interactive way.

* University of Tübingen, WSI/GRIS
Auf der Morgenstelle 10/C9, D72076 Tübingen, Germany
Email: {bartz, meissner, thuettne}@gris.uni-tuebingen.de

In [10], we proposed a new visibility algorithm. This algorithm exploits basic OpenGL functionality for fast visibility queries of large polygonal models. Frequently, 90% of a scene were culled due to occlusion at no losses in visual quality. Considering large scenes of millions of polygons, the achieved speed-up is significant and enables interactive handling of these scenes. In Figure 8, a scene of $1,056,280$ polygons was rendered at approximately 1.8 frames per second on a SGI O_2, where almost 98% of the polygons were culled, due to occlusion. Using view-frustum culling only, on average 0.6 frames per second were achieved.

Following the hierarchical z-buffer approach [7], a space-partitioning scheme (sloppy n-ary space-partitioning tree) was used to accelerate the queries. In contrast to [7], we did not use a z-pyramid as an image space hierarchy to accelerate occlusion queries. Instead, standard OpenGL buffers were exploited to implement a virtual occlusion buffer, which improved the performance of occlusion queries. Still, query performance was limited by searching the virtual occlusion buffer for changes. Consequently, an extension for visibility queries within OpenGL was proposed. In this paper, we present a detailed discussion of our extension and outline two possible hardware implementations.

Our paper is organized as follows: In Section 2, we briefly outline previous work that has been done in the field of hardware support for occlusion culling. Section 3 presents details of our proposed extension to the OpenGL rendering pipeline. Section 4 discusses implementations on different hardware platforms. In Section 5, we outline different additional applications of the proposed extension. 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 [17], Zhang provides a brief recent overview with some comparison. Brechner surveys methods for interactive walkthroughs [2]. However, we focus on papers which propose visibility algorithms using special hardware support.

In 1993, Greene et al. proposed the hierarchical z-buffer algorithm [7, 6]. After subdividing the scene into an octree, each of the octants is culled to the view-frustum as proposed in [5]. 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. Usually, the respective levels of the z-value-image-pyramid are searched for z-value changes, a feature which is commonly not supported in hardware. In [7], a hardware implementation of this query on a Kubota Pacific Titan 3000 workstation using a Denali GB graphics hardware is discussed. Still, most time of the visibility query is spent performing this "Z query".

Hong et al. [9] proposed a fusion between the hierarchical z-buffer algorithm [7] and the PVS-algorithm in [12]. 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 hardware z-buffer and temporal coherence to obtain high culling performance . 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 1997, occlusion culling using hierarchical occlusion maps was presented [17]. An occluder database is selected from the scene database. Using these occluders, screen bounding boxes of the potential occludees of the scene database are tested for overlaps, using an image hierarchy of the projected occluders (hierarchical occlusions maps). Basically, two features of this algorithm were supported in hardware; first, the construction process of the hierarchical occlusion maps can be supported by modern texture-mapping hardware. Second, the alternative use of a z-buffer as the depth estimation buffer for the overlap test.

Last year, Hewlett-Packard proposed an OpenGL extension for occlusion culling [8]. Similar to the hierarchical z-buffer approach, graphic primitives, which represent a more complex geometry, are rendered within an occlusion test mode to determine their visibility. Depending on the result, all underlying geometry is rendered or skipped.

3 Embedding Occlusion Queries in the OpenGL pipeline

The main source of performance problems of visibility algorithms like the hierarchical z-buffer [7] is the framebuffer-like design of the z-buffer. Most of the effort is spent tracking down the changes due to non-occlusion. This information would be easily obtained by directly catching the write enable signals of the depth buffer test. In contrast to the z-buffer, this information is data-sensitive and straightforward to process.

Generally, our strategy for occlusion-driven rendering of a given hierarchical subdivided scene is based on three steps. For each subdivision entity, we first render the entity (e.g. an octree block) in a special occlusion mode, which does not affect the content of the framebuffer, similar to the OpenGL selection mode. Second, we establish occlusion of the individual subdivision entity by using our occlusion extension. Finally, depending on the occlusion information, the actual graphic primitives, which are represented by the not occluded subdivision entity are rendered into the framebuffer.

Please note, for the correct computation of occlusion, backface culling must be enabled. Furthermore, the necessary counting of pixels of the subdivision entities is only correct, if the objects compound of these polygons are convex.

In this Section, we describe an extension to the OpenGL pipeline and API of step two. Basically, three features are provided by the extension.

- **Non-Occlusion Hit Counter (NOHC)**. This is used to quantify all not occluded pixels of the scan-converted subdivision entity. This provides simple analysis of the non-occlusion hits; how many, on which area of the viewport (we call this a occlusion tile).

- **Projection Hit Counter (PHC)**. This counts the number of pixels of the projection of the object to be rendered. Projection hits together with non-occlusion hits can provide information about how much of the projection of an object is not occluded.

  Further discussion on the use of the PHC can be found in Section 5.

- **Multiple Occlusion Tiles.** The complete viewport can be limited to smaller portions, or refined into a hierarchy of tiles. Alternatively to run a hierarchy of occlusion tests, multiple occlusion tiles can split the area of interest into a multiresolution non-occlusion hit representation, e.g. a quadtree-like representation of occlusion in a given scene (see Fig 1).

As another application of multiple occlusion tiles, visibility of portals in a PVS approach can be determined [12].

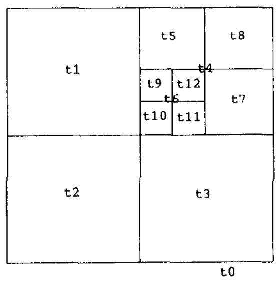


Figure 1: Quadtree of occlusion tiles $t_0..t_{12}$ are used.

3.1 An OpenGL API extension for Occlusion Queries

In order to exploit hardware extensions as proposed in Subsection 3.2 within OpenGL, we need to extend the OpenGL API. Basically, this extension takes place in three different ways:

Dual-use of already existing OpenGL calls

```
void glScissor(  GLint x, Glint y,
                 GLint width, Glint height)
```

To specify the occlusion tile, which limits the viewport for the occlusion test, the `glScissor` call is used. Within the viewport, only the tile, starting at x, y with width $width$ and height $height$ is considered for the occlusion test. This command is used to limit a test to the neighborhood of a certain area. By default, the whole viewport is used as occlusion test tile.

Adding new OpenGL calls

```
void glScissors(  GLint numTiles,
                  GLint *tiles)
```

In contrast to `glScissor`, `glScissors` specifies multiple tiles as occlusion test tiles. Depending on the occlusion hardware below, the various occlusion test tiles are distributed to different Occlusion Engines (see Subsection 3.2).

The parameter $numTiles$ and $tiles$ specify the number of tiles and a pointer to an array of $numTiles$ tile specifications. Each of these array entries contains x, y, $width$, $height$ of one tile.

```
void glOcclusionBuffers(  GLsizei *sizes,
                          GLuint **buffers)
```

Similar to the `glSelectBuffer` call of OpenGL, buffers for non-occlusion hits are specified occlusion tile-wise. All non-occlusion hits are stored into the occlusion buffers $buffers$ of the sizes specified in $sizes$. Minimum size of each occlusion buffer is eight, due to the minimal requirements of the `GL_BRIEF_OCCLUSION` mode, which is introduced in the next paragraph.

Adding new parameters to existing OpenGL calls

`void glGet(...)`

`GL_MAX_OCCLUSION_TILES` returns the maximal number of occlusion tiles. This information is important in case multiple occlusion tiles are used.

`GLint glRenderMode(GLenum mode)`

- `GL_BRIEF_OCCLUSION` is used to specify a fast occlusion mode. In this mode, the number of non-occluded hits and the number of projection hits are returned. Furthermore, to provide information on position and size of the various not occluded pixels, X_{min}, X_{max}, Y_{min}, and Y_{max} of the screen bounding box, and Z_{min}, and Z_{max} as minimal and maximal depth values of the non-occlusion hits are returned.
- `GL_VERBOSE_OCCLUSION`. In addition to the features of the `GL_BRIEF_OCCLUSION` mode, a list of the actual not occluded pixels of the occlusion tiles is returned, up to the maximum size of the occlusion buffer, specified with `glOcclusionBuffer()`.

If `glRenderMode(GL_RENDER)` is called, the respective occlusion information is returned into the buffers specified with `glOcclusionBuffers`. The syntax depends on the previous occlusion mode and enumerates the information tile-wise. If we encountered buffer overflows, the number of the respective tile buffers is returned. However, the buffers are still set with non-occlusion hits up to its maximum size - which is specified by `glOcclusionBuffers` - and terminates with a -1 entry. Consequently, some occlusion measure up to a user controllable limit is returned, without completely computing the potential costly occlusion information.

Note, similar to the `GL_SELECT` mode, all occlusion render modes do not change the content of the framebuffer,

3.2 Hardware-assisted Occlusion Culling

The implementation of the proposed extension to the OpenGL API does require a few modification within the OpenGL pipeline. To delineate our modifications, we will first give a brief overview of the OpenGL pipeline.

OpenGL rendering pipeline

OpenGL processes graphic data using a pipeline of several distinct stages [16]. In Figure 2, an abstract, high-level block diagram of this pipeline is given. Commands enter from the left and proceed through what can be thought of as functional units for the specific operations. Some commands specify the geometry of objects, while others control how the objects are processed during the various processing stages.

OpenGL operates in two modes. In **immediate mode**, all commands are executed directly when they are stated. Alternatively, a **Display List** can be used, where commands are compiled and stored for later execution.

In contrast to objects specified by vertices, parametric curves and surfaces are approximated by the **Evaluator** unit. Polynomial commands are evaluated to generate a vertex based description of the objects.

During the next stage, **Per Vertex Operations** and **Primitive Assembly**, OpenGL processes geometric primitives. These are points, line segments, and polygons, all of which are described by

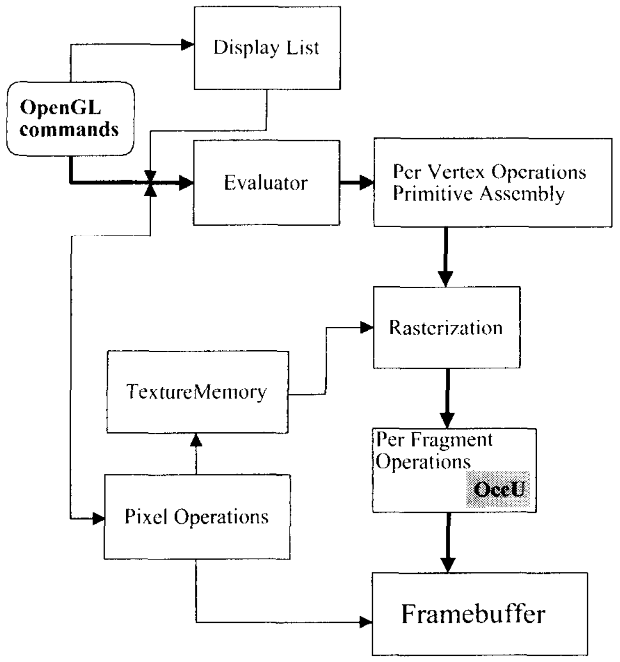


Figure 2: Schematic of the OpenGL rendering pipeline. OccU denotes where to fit in the proposed Occlusion Unit.

vertices. The vertices of the primitives are transformed and illuminated. Furthermore, the primitives are clipped to the viewport in preparation to the next stage.

The **Rasterization** unit produces framebuffer addresses for the rasterizing of the primitives. It interpolates associated values using two-dimensional descriptions of points, line segments, or polygons. The resulting fragments are then fed into the last stage, the **Per Fragment Operations**.

This stage performs the final operations on the data before the fragments are stored as pixels in the framebuffer. Since the framebuffer update depends on some conditions, some tests which evaluate arriving and previously stored z-values (for z-buffering) have to be carried out. Also, blending of incoming pixel colors with stored colors, as well as masking and other logical operations on pixel values are done in this stage of the pipeline.

Input can be in the form of pixels rather than vertices to describe two dimensional image data. This data skips the first stage of processing described above. Instead, it processes data as pixels in the **Pixel Operations** stage. The resulting pixels of this stage are either stored in **Texture Memory**, for use in the **Rasterization** stage, or merged directly into the **Framebuffer** just as if they were generated from geometric data.

The New Occlusion Unit

Many per fragment operations exist in the current OpenGL rendering pipeline. Some of the most important are scissoring, alpha test, stencil test, and depth buffer test, as shown in Figure 3.

Testing for occlusion is a "per fragment" operation since every pixel has to be tested. Therefore, our Occlusion Unit is part of the functional **Per Fragment Operations** block as illustrated in Figure 2.

We differentiate between the **Occlusion Unit (OccU)**, which is logically responsible for the overall occlusion, and the **Occlusion Engine**, which is the actual implementation of the Occlusion Unit.

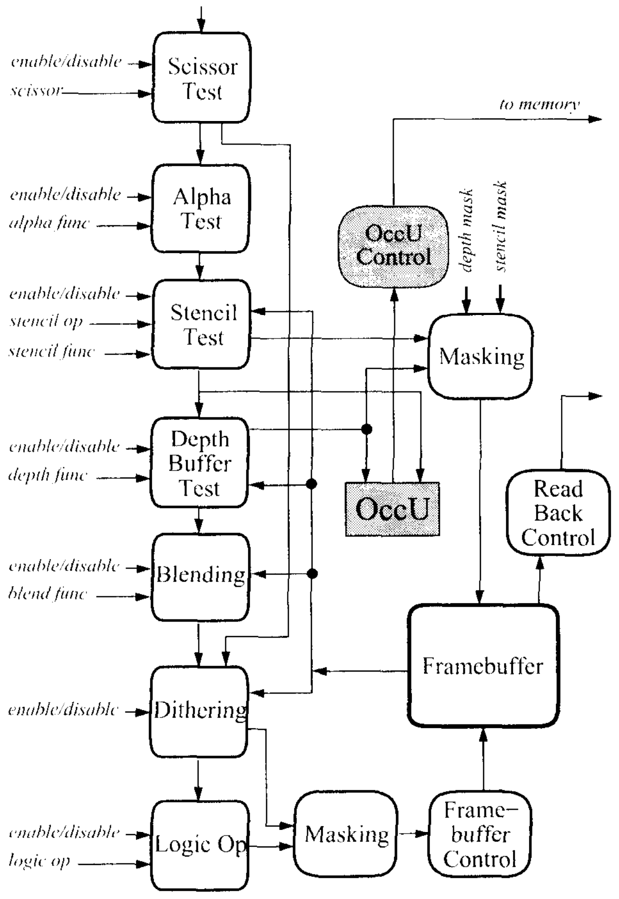


Figure 3: Per Fragment Operations and Framebuffer.

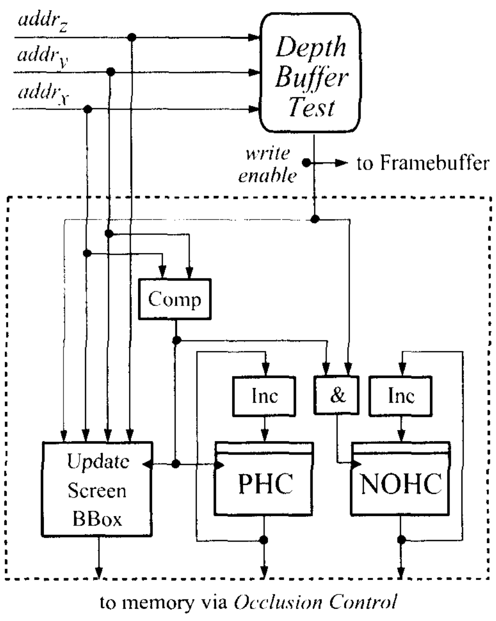


Figure 4: Schematical description of one Occlusion Engine.

In order to accelerate the processing of multiple tiles, the Occlusion Engine can be replicated within the Occlusion Unit. All Occlusion Engines of the Occlusion Unit are synchronized at the Occlusion Control (**OccU Control**).

To enable our proposed Occlusion Unit, we need to provide the unit with the x, y screen space address of the fragment, its depth value z, and the write enable signal of the depth buffer test, which is used to write and update the framebuffer with the fragment which is closer than the so far stored fragment. Therefore, we placed the Occlusion Unit behind the **Depth Buffer Test** unit, as it is demonstrated in Figure 3.

The Occlusion Unit tests the x, y screen space address of the fragment against the user defined occlusion tile. If the fragment resides within the tile, the projection hit counter (PHC) is incremented. Further, the non-occlusion hit counter (NOHC) is increased, if the depth buffer test was successful, which signifies the fragment contributes to the framebuffer. To trigger the increment of the non-occlusion hit counter, we use an AND operation. Besides increasing hit counters, we test whether the screen bounding box defined by the already found non-occlusion hits is increased due to the newly found hit. So far, the list of hits has yet not been updated. As long as the number of hits is smaller than the provided entries of the list, the x, y coordinates of the fragments are stored in the occlusion buffers which resides in main memory. To send data from the Occlusion Unit to the main memory, the OccU Control is introduced. This unit operates similar to **Selection Control** of the OpenGL selection mode. Its purpose is to synchronize memory access of the Occlusion Unit in case that multiple Occlusion Engines detect non-occlusion hits.

A schematic overview of an Occlusion Engine is given in Figure 4. Note, the Occlusion Engine shown in this Figure illustrates the schematic structure necessary to test for a user defined occlusion tile. Since the user can instantiate multiple tiles, e.g. a tile hierarchy, the Occlusion Engine has to be capable of updating all by the user instantiated tiles. This can be accelerated by assigning the tiles to multiple Occlusion Engines, using a round robin strategy.

4 Implementing the Occlusion Unit on two different Architectures

In this Section, we investigate the integration of our proposal in two existing architectures. We use for this two well known and described architectures of Silicon Graphics (see [13, 11]).

The SGI O_2 is an example for a medium performance graphics pipeline, which is comparable to many current PC graphics accelerators. It has a single rasterizing unit and a monolithic framebuffer. In Figure 5, we show details of the **Memory and Rendering Engine** of the SGI O_2.

This unit is connected to previous pipeline stages and to the main memory of the system. Its main part is the **Pixel Pipeline**, which performs all OpenGL rasterization, texturing, and per-fragment operations. Since no dedicated framebuffer is used, this implementation of the OpenGL pipeline uses an extensive pre-fetching algorithm to hide memory latency. The framebuffer itself is located in the main memory of the system.

To integrate our proposed extension, we need to place the Occlusion Unit into the Pixel Pipeline, where all the information necessary for the occlusion test is present. Address information is provided by the **Rasterizer**, while the write enable signal of the depth buffer test is provided by the **Depth/Stencil Pipeline**. Each Occlusion Engine which processes multiple tiles introduces additional cycles for each further tile. Since this is the case for OpenGL light sources too, we do not consider this as a serious drawback for mid-range graphics hardware.

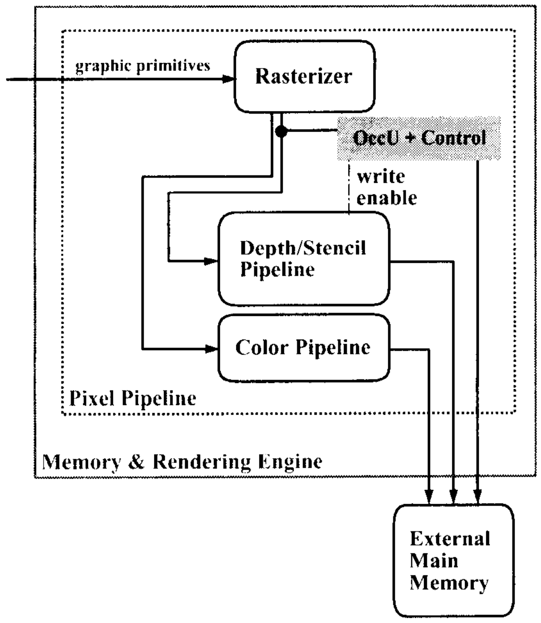


Figure 5: Memory and Rendering Engine of a SGI O_2 including the Occlusion Unit.

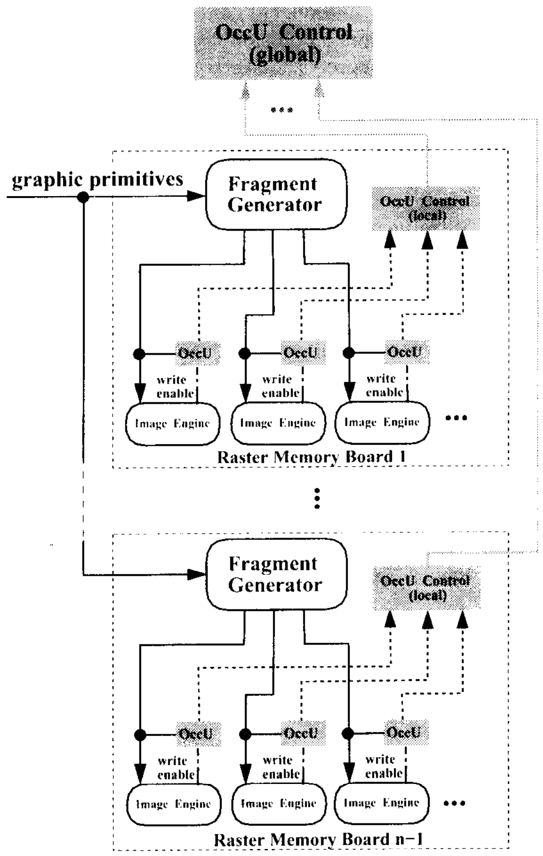


Figure 6: Schematic for implementing the Occlusion Unit on an InfiniteReality system.

In contrast to the SGI O_2, the InfiniteReality system is used as exponent for a high end graphics system. Its pipeline has a highly parallel architecture, containing multiple rasterizing units and an interleaved and distributed framebuffer [13], as it is illustrated in Figure 6.

The pixel operating part of the system is composed of the so called **Raster Memory Boards**. Each board has one rasterizer, called **Fragment Generator**, and an interleaved framebuffer which is accessible via special interfaces, the **Image Engines**.

Since our extension computes occlusion on a pixel basis, our Occlusion Units need to be integrated at the Image Engine level. In all Occlusion Units, Occlusion Engines are configured in the same way as their respective occlusion tile of the viewport. Consequently, each Occlusion Engine handles only hits of the part of the framebuffer to which its Image Engines belong to. In order to optimize occlusion performance, it is desirable to have an Occlusion Engine for each occlusion tile.

The evaluation process for an occlusion test has to respect the distributed nature of the system. Therefore, we propose a two stage synchronization process. First, the hits of one Raster Memory Board are synchronized locally. Thereafter, the result of the different boards are merged to form one occlusion report. During the latter synchronization process, detected non-occlusion hits which belong to the occlusion tile which is partioned between different Raster Memory Boards or different Image Engines needs to be merged to form a single occlusion report for this tile. This process can be either implemented in hardware or software.

The integration of our Occlusion Unit has been shown on two different graphic architectures. For a rather simple system as the SGI O_2, the Occlusion Unit can easily be integrated into the Pixel Pipeline. Although the integration into a InfiniteReality system is much more complicated, it is still feasible and not more difficult than the organization of the Image Engines themselves. Nevertheless, some latency will be introduced, due to necessary synchronization.

5 Further Applications

Adaptive Occlusion Culling

Adaptive occlusion culling was first proposed by Zhang in [17]. The basic idea is that objects which only have a small number of not occluded pixels have a small visual contribution to the final image. Therefore, if those objects are skipped, the visual impression of the rendered scene will not be jeopardized.

To quantify the contribution of the not occluded pixels of an object to the final image, we need to know their portion of the complete scan-converted object. This data is provided by the non-occlusion hits and the projection hits (pixels of the complete scan-converted object). Example: If only five pixels of 1000 scan-converted pixels of an object are not occluded, they might not have a significant contribution to the final image. Therefore, they can be skipped and valuable render time will be reduced (see Figure 10).

However, this feature needs to be used carefully in case the scene is very sparse. Furthermore, few pixels can have a strong impact on dynamic scenes. Skipping those pixels might result in flickering. On the other hand, missing interactivity usually has a stronger visual impact than flickering.

Support for Collision Detection

In virtual environments, it is a non trivial task to detect collisions of the user with objects. The process is highly demanding, since it requires a collision test with all objects. Currently, collision can be detected in VRML by introducing collision nodes. Each time the user changes its position, a collision test is applied to all collision nodes. Due to the time consuming collision test, the frame rate drops and people tend to switch the collision detection mode off.

Our proposed extension for OpenGL can be used for this purpose to a certain degree. For a given position of the user, an image of the scene is generated. A mostly valid assumption is that the user changes its position and direction incrementally. Hence, in case the user heads in viewing direction - e.g. straight, straight-left - a subdivision of screen space can be determined which represents the possible collision areas within the viewport. This is illustrated in Figure 7.

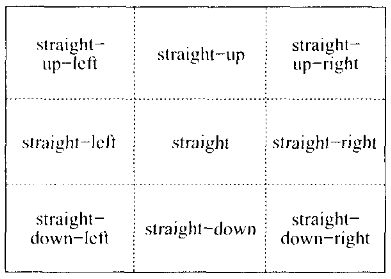


Figure 7: Subdivision of screen space into areas which correspond to the heading direction of the user.

To check whether a certain step will cause a collision, a customized view-frustum covering the check area of the screen is rendered in GL_BRIEF_OCCLUSION mode. The far plane of this view-frustum depends on the step size of the user, near plane is identical to the view plane. Information whether a step can be taken without causing a collision is indicated by the non-occlusion hits and the projection hits. If the number of non-occlusion hits is different from the number of projection hits, some pixels of the customized view-frustum are occluded, which means that a collision can be expected. In contrast to our occlusion test, backface culling must be disabled, in order to detect intersections with the backfacing polygons of the view-frustum. To get more detailed information, screen space can be subdivided further.

So far, collision detection can only be indicated and depends on the given viewport. Unfortunately, testing backward stepping does require two-pass rendering since no image of the scene behind the user is available in the framebuffer.

Support for Ray Casting

One of the results of the GL_BRIEF_OCCLUSION and GL_VERBOSE_OCCLUSION modes is a list of not occluded pixels of the tested subdivision entity. Besides usual statements on occlusion or non-occlusion of that entity, this information can be used to accelerate ray casting in a mixed volume graphics and polygon model. Considering a hierarchical ray casting approach, we render the subdivision entities in an occlusion mode. All computed non-occlusion hits for this entity mark pixels which are not occluded. Consequently, these pixels are image plane parts to cast rays through, because the content represented by the subdivision entity may be still visible. In other words, all pixels without an non-occlusion hit are not visible and therefore, rays casted through those pixels of the image plane have no contribution.

6 Conclusion and Future Work

In this paper, we proposed a hardware extension for occlusion queries. Although, we focused only on OpenGL, this extension can be adapted to other graphic API's as well. Based on a hierarchical occlusion strategy, first subdivision entities are rendered in a special occlusion mode to determine occlusion of a 3D scene. Thereafter, the actual scene primitives are rendered with respect to the occlusion information of the first step.

Aspects of the OpenGL API and hardware were discussed to specify an Occlusion Unit as rather small extension to the OpenGL pipeline. This Occlusion Unit is located in the "Per Fragment Operations" stage of the pipeline to catch write enable signals of the depth buffer test.

Depending on the actual number of Occlusion Engines in the rendering pipeline, multiple occlusion tiles, which subdivide the viewport, can be processed in parallel.

Future work will focus on an implementation of this unit in hardware, to determine the real-time occlusion query performance. Furthermore, advanced features such as occlusion support for ray casting in a mixed polygon/volume model will be examined.

Acknowledgments

This work has been supported by the MedWis program of the German Federal Ministry for Education, Science, Research and Technology, by DFG project SFB 382, and by the state of Baden-Württemberg.

We like to thank Urs Kanus and Anders Kugler for useful discussions, Michael Doggett for proof-reading, and Ralf Sondershaus for the tree models of the forest scene.

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

[5] 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.

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

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

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

[9] 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.

[10] T. Hüttner, M. Meißner, and D. Bartz. Opengl-assisted visibility queries of large polygonal models. Technical Report WSI-98-6, ISSN 0946-3852, Dept. of Computer Science (WSI), University of Tübingen, 1998.

[11] M. Kilgard. Realizing opengl: Two implementations of one architecture. In *1997 EG/SIGGRAPH Workshop on Graphics Hardware*, 1997.

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

[13] J. Montrym, D. Baum, D. Dignam, and C. Migdal. Infinite-reality: A real-time graphics system. In *Proc. of ACM SIGGRAPH*, pages 293–302, 1997.

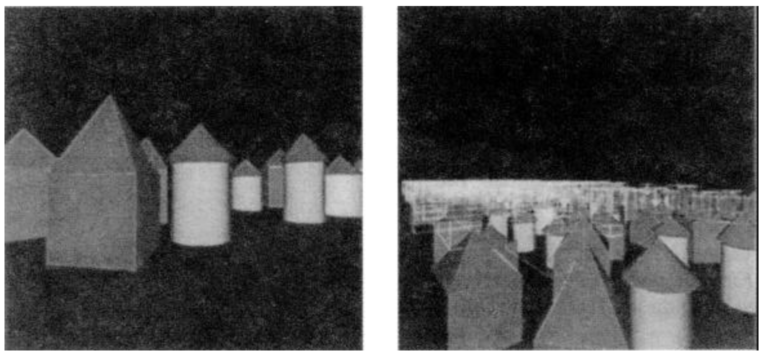
[14] W. Straßer. *Schnelle Kurven- und Flächendarstellung auf graphischen Sichtgeräten*. PhD thesis, Technische Universität Berlin, 1974.

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

[16] M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide*. Addison-Wesley, 2nd edition, 1997.

[17] 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.

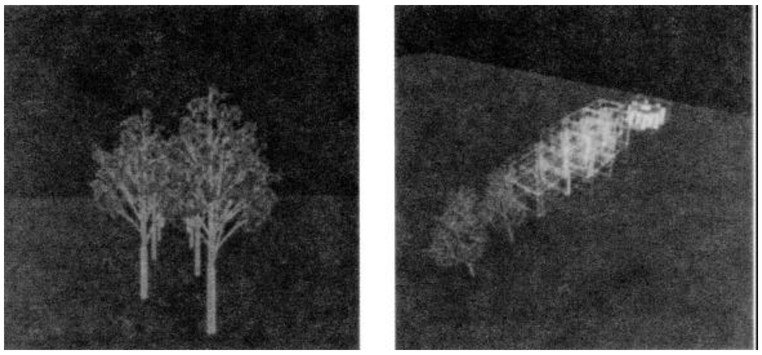
City Model



(a) (b)

Figure 1: City model is rendered using a hierarchical occlusion strategy: Bounding volumes are rendered in an occlusion mode to determine occlusion. All yellow bounding volumes are found occluded; only 0.2% of the geometry is actually rendered. (a) Visitor's perspective. (b) Bird's perspective of visitor's view

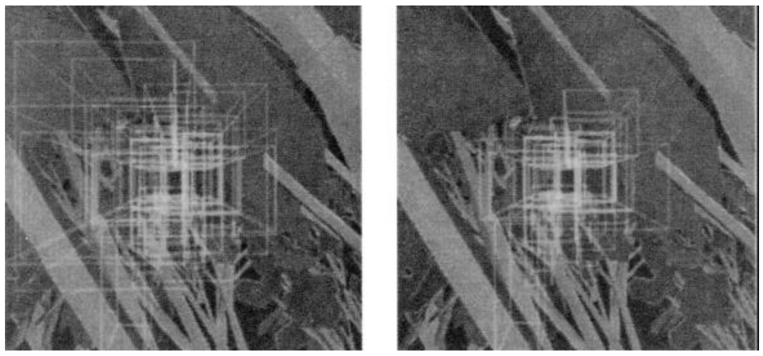
Forest Scene



(a) (b)

Figure 2: Forest Scene - (a) Front view. (b) Overview - all culled bounding volumes are marked yellow.

Forst Scene: Up Close and Personal



(a) (b)

Figure 3: For exploiting ratios of projections and occlusion hits, adaptive occlusion culling can be used. The forest scene is rendered using adaptive occlusion culling, where blocks are considered occluded if only a small number of occlusion hits is found (with respects to the number of projection hits). Alley of trees - bounding volumes of culled objects are marked yellow: (a) Adaptive Occlusion culling. (b) Occlusion culling.