Parallel Visibility Test and Occlusion Culling in Avango Virtual Environment Framework

Stanislav Klimenko[(1)], Lialia Nikitina[(2)], Igor Nikitin[(2)]

[(1)] *Institute of Computing for Physics and Technology, Protvino, Russia*
[(2)] *Fraunhofer Institute for Media Communication, Sankt Augustin, Germany*

Abstract

Real-time visibility test is an attractive feature for Virtual Environment (VE) applications where large datasets should be interactively explored. In such scenes most of the objects are usually occluded by other ones, and their omission from rendering can significantly accelerate the graphical performance. In this paper we present our novel approach for occlusion culling and its implementation in VE system Avango[1]. Visibility test module performs real-time update of the list of visible objects by means of color labeling and either hardware or software histogramming of corresponding color buffer. Exploiting the distribution capabilities of Avango, the main draw and visibility test processes are parallelized to different computers running Irix or Linux. Being applied to an architectural model containing 260,000 textured triangles, our method accelerates the graphical performance from 8 to 32 stereoimages/sec.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Bitmap and framebuffer operations, Display and viewing algorithms; I.3.7 [Computer Graphics]: Virtual reality.

1. Introduction

To create the sense of presence in virtual environments, high level of scene detailization is needed. Increasing size of datasets intensifies the graphical load, requiring new methods for graphics acceleration. Depth complexity, the number of layers of geometry underneath each pixel, is a characteristics, measuring the wasted effort of graphics spent on rendering of invisible objects. Common techniques, implemented on most of available graphical hardware, such as viewing frustum culling, backface culling and level-of-detail switching, do not reduce significantly the depth complexity. Occlusion culling is aimed to reduce the depth complexity ideally to one, and can considerably accelerate the rendering process. The occlusion culling algorithms are implemented only in certain graphical systems.

This paper is organized as follows. The next section briefly reviews the existing occlusion culling techniques. The following section introduces created at our laboratory software framework Avango, used for the development of interactive VE applications. Then we describe the implementation of occlusion culling in Avango, present its quantitative characteristics, and make comparison with other available methods.

2. Related work

A lot of efficient techniques have been developed in the area of visibility test and occlusion culling. Specially designed for indoor architectural models *cells and portals method*[2] subdivides the building to rooms or cells, only visible through small doorways or windows, also known as portals. The visibility data for each cell are pre-computed and then used on-line for occlusion culling. Other technique with intensive pre-processing stage, *prioritized-layered projection algorithm*[3] estimates the probability for each cell of the object space to be occluded by other objects, from a given viewpoint, and either discards unlikely visible cells or iteratively refines information on their visibility using on-line queries. Visibility queries are also used in *octree-based technique* [4]. The object space is subdivided by 2^3-cubes hierarchy, associating each primitive to the smallest cube containing it. Cubes are rendered recursively in front-to-back order, each time determining whether the faces of the cube are visible or hidden by already displayed primitives (this

is done using either stencil-buffer or HP-occlusion test flag based techniques, described below). If the cube is visible, associated primitives are drawn and the recursion is continued. The other technique, oftenly used in combination with the object-space octree, is *image-space Z-pyramid* [4]. The original Z-buffer is taken as the finest level in the pyramid and then four Z-values at each level are combined into one Z-value at the next coarser level by choosing the farthest Z from the observer (using e.g. texture maps[5]). Each polygon is projected to the screen, then one finds its 2D bounding box, smallest square of subdivision containing it and the level of Z-pyramid, where this square corresponds to one pixel. If the nearest Z-value of the polygon is farther away than Z-value at this pixel, the polygon is hidden. Otherwise the polygon is rendered and Z-pyramid is updated. In *HP-hardware supported algorithm*[6] the feedback loop is added to the graphical hardware, able to check, whether an attempt is made to write in Z-buffer during rendering of a given primitive, without actual change of Z and color buffers content. This feature allows to check whether the bounding box of the object is hidden, and in this case discard the object from further drawing. The method becomes effective only after front-back sorting of the objects, done e.g. in octree technique. Like all other bounding box based methods it is excessively conservative, because the hidden objects, whose bounding box is visible, are rendered. Similar to HP-algorithm, but implementable in other hardware by means of standard OpenGL, *visibility testing in stencil-buffer*[7] suggests to send the bounding box of the object to the stencil-buffer with Z-test but without write to Z-buffer. If the bounding box does not leave a footprint in the stencil-buffer, then it is hidden and the object is not drawn. A bottleneck of this method is a slow reading of the stencil-buffer content, which is resolved in[7] by the buffer sampling. *Histogramming techniques*[3] can be also used to resolve analogous problem in color buffer reading. Several other techniques are referred in the survey paper [8] describing much of the previous work done on visibility based graphical acceleration.

Some of the described techniques are hardware implemented in HP, ATI and Nvidia graphical systems and have support in OpenGL Optimizer[9], OpenSG Plus[10] and Jupiter[11] software. Most of widely used scene-graph programming software such as IRIS/OpenGL Performer do not support the occlusion culling. The major problem is that the existing techniques of the occlusion culling require to receive and analyze the feedback from graphics after rendering of *each primitive*. This can be easily implemented in OpenGL, but does not well suite to the concepts of Performer.

Occlusion culling techniques can benefit from implementation in server-client architectures and multi-processing systems. It was stressed in[12] that certain techniques of visibility test allow their parallel execution with draw process without frame-to-frame synchronization. Moreover, at appropriate scheduling the main draw can achieve higher framerates than the testing process, preserving visual quality of the output. In particular,[12] used parallel scheme with effective 2.5D visibility algorithm for urban walkthrough. Our preference is more general method applicable for arbitrary 3D scenes.

In this paper we describe the implementation of occlusion culling in Performer-based Avango VE framework. The method does not use special hardware features and is implemented by means of standard OpenGL on Irix and Linux platforms. Parallel execution of rendering and visibility testing processes is used to achieve greater speedups. The resulting graphical acceleration is sufficient for the real-time walkthroughs in large Virtual Environments.

3. Avango

is a programming framework [1] for building distributed, interactive VE applications. It uses the C++ programming language to define two categories of object classes. *Nodes* provide an object-oriented *scene graph* API which allows the representation and rendering of complex geometry. *Sensors* provide Avango with an interface to the real world and they are used to import external device data into the application. All Avango objects are *fieldcontainers*, representing object state information as a collection of *fields*. Avango uses connections between fields to build a *dataflow graph* which is conceptually orthogonal to the scene graph, and it is used to specify additional relationships between nodes, which cannot be expressed in terms of the standard scene graph. This facilitates the implementation of interactive behavior and the import of real world data into the scene graph.

Avango supports a generic *streaming* interface, which allows objects and their state information to be written to a stream, and the subsequent reconstruction of the object from that stream. This interface is one of the basic building blocks used for the implementation of object distribution. Distributed Avango objects are allocated from distributed segments of shared memory, which are kept synchronized via underlying network layer infrastructure. This model provides the required network transparency for the objects to be visible by all participating processes and to interact with the same efficiency as they are located in a single computer.

In addition to the C++ API, Avango features a complete language binding to the interpreted language Scheme [13]. All high level Avango objects can be created and manipulated from Scheme. The Avango is originally based on *IRIS Performer* to achieve the maximum possible performance for an application and addresses the special needs involved in application development of Virtual Environments. Whenever the underlying hardware allows, Performer utilizes multiple processors and multiple graphics pipelines. Currently we use Avango implementations on SGI computers and Linux-PCs.

Specially for the design of interactive VE applications Avango provides interfaces and interaction metaphors, used

for navigation in virtual spaces and manipulation of virtual objects. Particularly, walkthroughs in virtual scenes are supported by *av-mover*, which can be attached to various interaction devices (mouse, joystick, stylus) and configured for various navigation modes, including unconstraint flying and ground following.

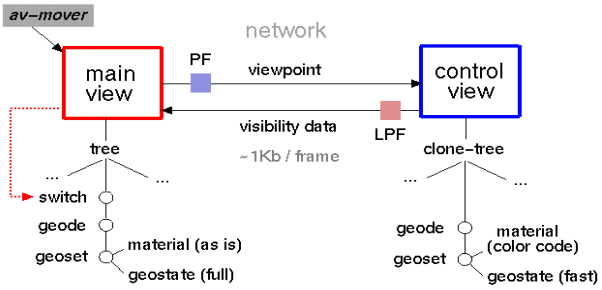

Fig.1. Distributed setup for occlusion culling.

4. Implementation of occlusion culling

Our approach is based on real-time visibility testing. To define, which objects in the scene are visible, we perform prerendering of all objects in a separate frame buffer, referred further as *control view*, using simplified graphical modes and labeling the objects by color. Then we perform histogramming of color buffer to determine which colors are visible, and draw the corresponding objects in *main view*, using necessary time-consuming draw modes.

To implement this approach, the following scheme is used (fig.1). Main and control applications are running on separate computers (members of Unix cluster) and are communicating via network. The scene graph for main view represents the original model, while the control view renders the copy of the scene graph with simplified geostates and materials used for color coding of the objects. The navigation is performed on the side of main view, and the viewpoint data (the matrix, describing camera position and orientation) are transmitted to the other side. The control view then produces visibility data for the given viewpoint and returns them to the main view. The latter one switches off invisible objects in its scene graph and renders only visible ones.

To accelerate the control view *draw* process, we disable unnecessary graphical modes, such as texturing, enlightening and antialiasing. Available view frustum and backface culling techniques are enabled. Additionally, in stereo-setups, used to support virtual environments, one control view can perform visibility testing for left and right viewpoints.

Histogramming operation, a counting of pixels of each color, present in the frame-buffer, is supported by most of graphical hardware and can be enabled by OpenGL call `glHistogram`. This operation fills four separate one-dimensional histograms, corresponding to RGBA channels, giving possibility to assign color codes to maximum $255 \times 4 = 1020$ objects (the background color is not used for coding). The hardware histogramming operation of full screen image is usually time-consuming, see Table 1.

Table 1: time of histogramming operation

	1024×768 screen		512×384 screen	
	hardware	software	hardware	software
Onyx2 195MHz InfiniteReality2	41ms	36ms	17ms	15ms
Athlon 1.3GHz GeForce Quadro2Pro	86ms	87ms	17ms	25ms

Other possibility is to copy the content of color buffer to main memory and perform its software histogramming. In this way higher-dimensional (2D...4D) histograms can be filled, giving maximum $256^4 - 1 \approx 4$billion color codes, enough for the applications. However, the software based histogramming of full screen is time-consuming as well.

Higher framerates can be achieved, if we reduce the resolution of the control window. Particularly, a quarter of the full screen is processed at acceptable rate (see Table 1). The reduction of the control window leads to the following problem. One pixel on the reduced control window corresponds to four pixels on main window. Small objects will have nonstable visibility test, resulting to flickering of their counterparts on main window. We suppress this effect applying to visibility data a low-pass filter (LPF-box on fig.1).

Low-pass filter averages the input data S_n^{in} (number of pixels of each color at n-th frame, found in histogramming operation) during a given number of frames N:

$$S_n^{\text{out}} = \frac{1}{N} \sum_{k=0}^{N-1} S_{n-k}^{\text{in}}.$$

As a result of averaging, high frequency flickering is removed. Besides, in the formula above we actually perform statistical improvement of input data found in N measurements from closely placed viewpoint positions. As a result, the output data S_n^{out} are equal to the areas occupied by the objects on the control view, known *at subpixel precision* (mean square error of $S_n^{\text{out}} \sim 1/\sqrt{N}$). This statistical mechanism is similar to antialiasing techniques, effectively enhancing the window resolution. The price for that is slightly increased conservatism of the method: some of already invisible objects will be rendered.

Parallelization of main draw and visibility test processes requires to take care on their synchronization. The major problem is that visibility data for a given viewpoint are returned to main process at the delayed time. To compensate

these delays, we pass the viewpoint data through a predicting filter (PF-box on fig.1).

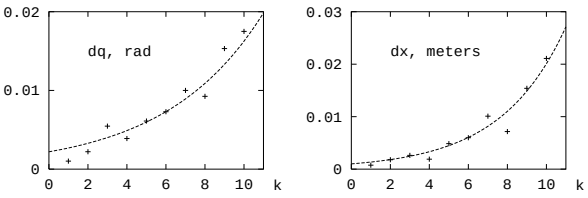

Fig.2. Precision of prediction as a function of prediction advance.

Predicting filters are extensively used in applications, particularly for motion tracking, autonomous or assisted navigation[14]. Predicting strategies are integrated in modern microprocessors design to speedup the instructions execution pipeline[15]. Similar approaches are also exploited for stabilization of various numerical methods in theory of dynamical systems[16, 17]. In our case, predicting filter is used to extrapolate the viewpoint matrix for k future frames. For this purpose we construct n-th order polynomial, approximating p last viewpoints by the least square method, and continue it to the nearest future. For sufficiently smooth motion, such as generated by *av-mover*, the method produces good prediction at $n = 2,\ p = 20$. The precision characteristics of the method are presented on fig.2. Here $dq = |q_{\text{measured}} - q_{\text{predicted}}|$ is precision of rotational component of the viewpoint matrix in quaternionic space, averaged by $N \sim 10^4$ frames during 5 min walkthrough in a virtual model, dx is similarly defined precision of translational component in world coordinates, k is the number of future frames (prediction advance). The measured values are well described by exponential dependence. For $k \leq 10$ the precision of the method is sufficient for our purposes.

Finally, the decision on a visibility of the object is taken using logical *or* operation on the data, coming from the low-pass filter:

$$(S^{\text{out}}_{\text{last}} > S_{\text{threshold}}) \parallel (S^{\text{out}}_{\text{next}} > S_{\text{threshold}}),$$

where $S^{\text{out}}_{\text{last}}$ are the nearest available visibility data in the past, and $S^{\text{out}}_{\text{next}}$ are the nearest available visibility data in the future, with respect to the present moment of time. As a result, on the camera path, connecting the points, where the visibility data are available, the object is decided to be visible, if it is visible at least in one of the end points. This solution also leads to small increase of the conservatism of the method, involving to draw process some of yet invisible objects. Besides, in the case, if the object becomes visible only for a short period of time between two consecutive control points, it will not be rendered at all. This situation, however, appears quite rarely and due to short interval between control points ($\sim 1/30$ sec) is not visually perceivable.

Note: As an additional method for control view draw acceleration one can use the level-of-detail algorithm to exclude faraway objects occupying too small area on the screen (e.g. $S < 0.01\text{px}^2$) from the control view, and consequently from the main view draw processes. Semi-transparent objects, if present in the scene, can be easily incorporated in our scheme by exclusion from the control view scene-graph and unconditional draw in the main view. Dynamical objects can be handled analogously. One can also introduce for the dynamical objects necessary advanced phase shifts and include them to control draw process, particularly if the objects move along pre-defined trajectories or their motion is sufficiently regular to be effectively predicted.

Depth complexity measurement, performed to verify the effectiveness of the method, is implemented as follows. All objects are drawn in white color, with alpha-value set to a given constant α. The scene is rendered in additive transparency mode. After that, the color buffer contains values αD, where D is the depth complexity.

5. The results

For test we perform walkthrough in a model of ancient city Colonia Ulpia Trajana[18], created by students of Civil Engineering Department at University of Dortmund. The model contains 260,000 textured triangles in 1,200 geosets, and without acceleration is rendered on Onyx2 at the frame rate 8 fps. The appearance of this model in main and control views is presented on fig.3. Fig.4 illustrates the results of occlusion culling, performed for the viewpoint located in the left bottom corner of the image. The scene is shown from a different viewpoint to display the occluded objects, omitted by culling, while the images from the original viewpoint are identical. The left scene on fig.4 is subjected only to view frustum culling and contains 86,000 triangles, while the right scene after the view frustum and occlusion culling contains 4,000 triangles. The depth complexity pattern is displayed on fig.5. Without the occlusion culling this pattern is concentrated in a narrow band along the horizon, where it reaches values 30...46. With the occlusion culling, the depth complexity becomes almost uniformly distributed over the screen, everywhere not exceeding value 9. Fig.6 presents the framerates for a given camera path, passing nearby a complex object. Without occlusion culling the framerate is considerably changed, revealing the graphics overload, while occlusion culling makes this dependence almost flat.

The acceleration factors, measured for this model in various hardware setups, are given in Table 2. Maximum value of acceleration strongly depends on the model composition and graphical hardware used. Particularly, one can hide very complex object by a single wall, and reduce the depth complexity by several orders of magnitude. The draw time in this case will be defined by the filling time of graphics card, i.e. the time, needed to draw one polygon, projected to the full screen (about 0.3ms for 1024x768 screen on InfiniteReality2). The scene with depth complexity one everywhere on the screen cannot be drawn faster.

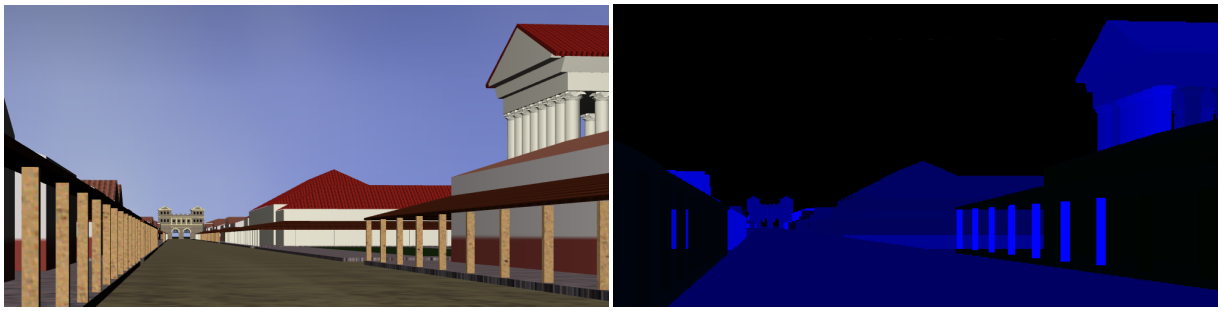

Fig.3. Architectural model, used for the test of the algorithm. On the left – the main view image (identical images are produced with and without occlusion culling). On the right – the control view image, encoding the objects by color.

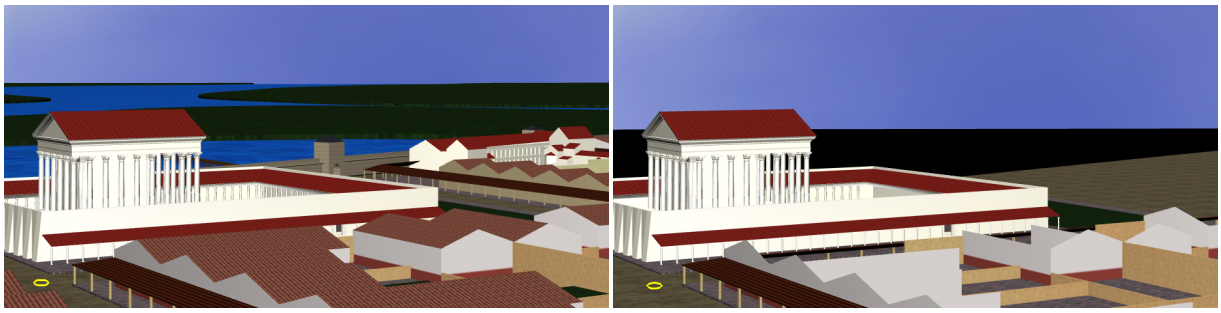

Fig.4. The same model, displayed from the other view point: on the left – without occlusion culling, on the right – with occlusion culling. The occlusion culling algorithm was applied for the viewpoint location, marked by a circle on the left bottom.

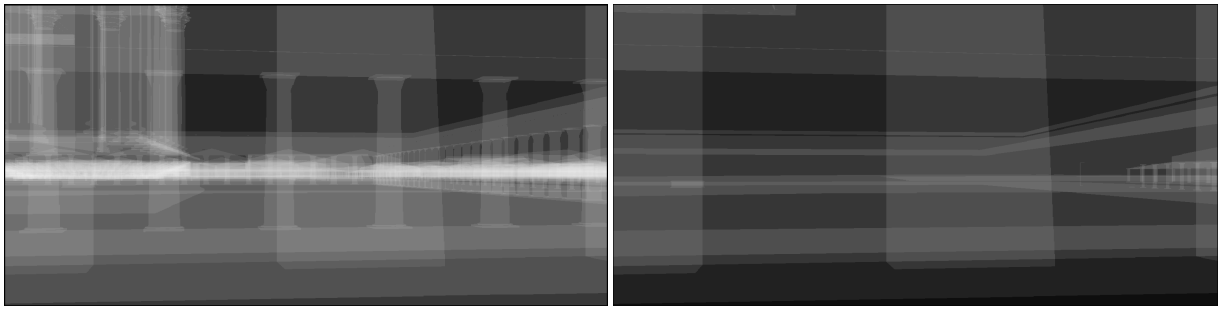

Fig.5. Depth complexity pattern: on the left – without occlusion culling, on the right – with occlusion culling.

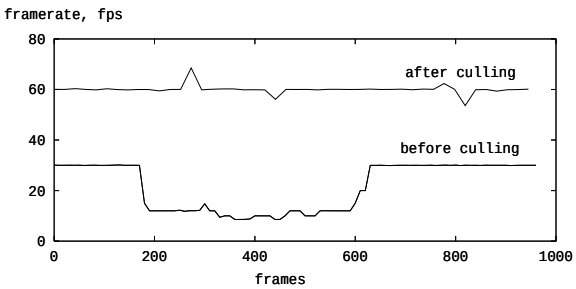

Fig.6. Framerate for the camera path.

Table 2: acceleration factors

main window 2x1024x768	control window 512x384	acceleration factor min	avg	max
Onyx2 300MHz InfiniteReality2 active stereo	Onyx2 195MHz InfiniteReality2	1.7	4	52*
	Athlon 1.3GHz GeForce Quadro2Pro	1.7	4	52*
HP 2GHz Fire GL4 passive stereo	Athlon 1.3GHz GeForce Quadro2Pro	1.3	2.5	10*

* limited by filling time of the graphics card.

Comparison with other techniques. At first, we emphasize that the acceleration factor for a given technique of occlusion culling strongly depends on the model used for testing. More dense environments, highly populated by occluders, demonstrate better characteristics. The following figures give broad estimate of capabilities for several available techniques, obtained on *different* models. The paper[10] gives the estimation of acceleration factors for three techniques, applied for a 750,000 polygonal model at 6fps: stencil test (1.12), Z-buffer test (1.148), HP-occlusion test (1.338). In the paper[11] the acceleration factors 1.5...4 are obtained using HP-occlusion test for models with 600,000...11mln triangles at 1..12fps. The paper[7] presents acceleration factors 2.6...7 for stencil test technique obtained for a set of models with 450,000...5mln triangles at 1...2fps framerate. In the paper[5] Z-pyramid test produced acceleration factor 5 for 300,000...1mln polygonal models at 8..12fps. The paper[3] reports acceleration factor 12 for 1mln triangles model at 12fps obtained with prioritized-layered projection algorithm using direct scan of 512×512 color buffer, and twice lower characteristics for its hardware histogramming. The paper[12] demonstrates the acceleration factor 28 at 60fps for 2mln polygons urban environment, obtained in distributed setup with 2.5D visibility testing technique.

6. Conclusion

In this paper we described our implementation of visibility test and occlusion culling in Avango VE framework. The approach uses determination of visibility in arbitrary 3D scenes by means of pre-rendering, color coding of the objects and histogramming of the color buffer, statistical improvement of visibility data to subpixel precision, distributed setup for parallelization of main draw and visibility test processes, and prediction of the viewpoint position for compensation of hardware and software delays. The method has been tested for large architectural model, accelerating the graphical performance of walkthrough VE application to real-time speeds.

7. Acknowledgments.

This work has been partially supported by RFBR grants 01-07-90327 and 02-01-01139.

References

1. H. Tramberend, Avocado: A Distributed Virtual Reality Framework, Proc. of the IEEE Virtual Reality, 1999.
2. D. Luebke, C. Georges, Portals and mirrors: Simple, fast evaluation of potentially visible sets. 1995 Symposium on Interactive 3D Graphics, ACM SIGGRAPH, April 1995, pp.105-106.
3. J. Klosowski, C. Silva, Efficient Conservative Visibility Culling using the Prioritized-Layered Projection Algorithm, IEEE Transactions on Visualization and Computer Graphics, Vol. 7, No. 4, 2001, pp. 365-379.
4. N. Greene, M. Kass, G. Miller, Hierarchical Z-buffer visibility, SIGGRAPH'93, pp.231-236.
5. H. Zhang, D. Manocha, T. Hudson, K.E. Hoff III. Visibility Culling Using Hierarchical Occlusion Maps. Computer Graphics (Proceedings of SIGGRAPH'97) pp.77-88, 1997.
6. N. Scott, D. Olsen, E. Gannet, An Overview of the visualize fx graphics accelerator hardware. The Hewlett-Packard Journal, 28-34, May 1998.
7. T. Hüttner, M. Meissner, 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.
8. D. Cohen-Or, Y. Chrysanthou, C.T. Silva. A survey of visibility for walkthrough applications. Proc. of EUROGRAPHICS'00, course notes, 2000.
9. Silicon Graphics Inc. OpenGL Optimizer Programmer's Guide. Technical Report, 1998.
10. D.S. Staneker, A first step towards occlusion culling in OpenSG PLUS, in Proc. of the 1st OpenSG Symposium, 2002.
11. D. Bartz et al, Jupiter: A Toolkit for Interactive Large Model Visualization, Proc. of Symposium on Parallel and Large Data Visualization and Graphics, pp.129-134, 2001.
12. P. Wonka, M. Wimmer, and F. Sillion. Instant Visibility. in Proc. Eurographics 2001.
13. R. Kent Dybvig. The Scheme programming language: ANSI Scheme. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, Edition 2, 1996.
14. G. Welch, G. Bishop. An introduction to the Kalman filter, ACM SIGGRAPH'2001, Course 8.
15. M. Evers et al, Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work, Proceedings of the 25th International Symposium on Computer Architecture, Barcelona, June 1998, pp.52-61.
16. S.V. Klimenko, I.N. Nikitin, W.F. Urazmetov. Methods of numerical analysis of 1-dimensional 2-body problem in Wheeler-Feynman electrodynamics, Computer Physics Communications, Vol.126 (2000) pp. 82-87.
17. G. Hirota, S. Fisher, M. Lin, Simulation of Non-penetrating Elastic Bodies Using Distance Fields. UNC Technical Report TR00-018, 2000.
18. http://viswiz.imk.fraunhofer.de/~lia/xanten