Point-visibility in computer games

Harald Riegler
Institute of Computer Graphics, Technical University Vienna

Abstract:

This paper gives an overview over current methods for visibility calculations from a point. We will evaluate the different methods to see how well they are suited for computer game applications, and we will describe current solutions that have been implemented in hardware. In the last section we will describe some practical examples that have actually been implemented in computer games.

1.) Introduction

Visibility computations have become a very important field of research in the last years, and without the advances in these fields modern computer games would not look as impressive as they do today. We will start our paper with a brief description of what visibility is, and make some definitions that are important in the description of the algorithms following this section.

When rendering large and complex models, the graphics hardware is mostly driven to the limits of the rendering capacity. Memory bandwidth and triangle rendering speed still can not cope with the amount of data that is fed through the rendering pipeline. Much of this effort is in vain though, because mostly we can only see parts of the actual model. The rest is either not in our field of view, is occluded behind other objects, or the vertices are on the rear side of an object and thus not visible. What visibility computation does is to determine which objects, and which vertices on such an object are visible and which are invisible. Once they have been identified as hidden, they can be removed from the rendering pipeline, and rendering speed is increased.

We can distinguish three different classes of visibility classifications.

- The *exact* visibility set:
  This is the set of all polygons that are partly or completely visible
- The *approximate* visibility set:
  This set includes most of the visible polygons, but also some hidden ones.
- The *conservative* visibility set:
  This set includes *all* visible polygons, and maybe some hidden ones too.

Determining the exact visibility is very intensive in terms of computational power. Furthermore, it is difficult to design a fast algorithm that is *exact* and still robust. The methods we will describe are mostly *conservative* visibility methods, which allows us to use simplifications in some areas. Some methods can also be modified to calculate the approximate visibility set, which can lead to enormous speedups. Especially in games we can sometimes accept that minor details, that should actually be visible, remain invisible.

There are two main different methods to approach visibility problems in computer games, depending mostly on the game setting, and thus on the type of geometrical data present.

Visibility from a region:

This is a kind of visibility calculation that is mostly used for indoor settings, making use of the limited viewing possibilities between rooms. The rooms are termed cells, and doors between these cells are called portals. In a precomputation step the data is analysed for the potentially visibile set (PVS) from all positions in a single cell. This is done for all cells in the data set. Current implementations in games use mostly precomputed level data in the final product, as precomputations are too time-consuming (up to several hours, and often needing manual interference) to be done after the loading of a level. Games like Quake and Unreal fall into this category. One of the benefits of this method is that it is very efficient, and that other things, like radiosity and shadows, can often

be calculated in the same procedure, once the process is started out of the level editor. The main drawback is the enormous memory consumption.

Visibility from a point:

This method is mostly implemented in real-time, but sometimes parts are precomputed too. Visibility is determined from the current viewpoint only. These methods are widely used in applications like racing games, sports games, beat-em-up's, platform games and sometimes in real-time-strategy games. We will focus on visibility from a point in this paper.

2.) Visibility from a point

There are 2 different categories of methods when it comes to point-visibility. Calculations in image-space and those in object space. Current methods have started to use the best of both worlds as we will see.

Furthermore we will judge the different methods in terms of their efficient use of coherency.

- Image-space coherency
- Object-space coherency
- Temporal coherency

Another quality criteria for these methods is if they support occluder fusion, which means that the method takes the effects of multiple occluders into account. An example for occluder fusion would be a hill that may be visible behind a skyscraper if we just examine this one building. But the hill may actually be hidden behind the complete block of buildings, which consists of multiple occluders. A method that does not support occluder fusion would mark the hill as visible.

We have selected four different methods that we will describe, two in image space, and two in object space. There are more algorithms, and we present a list of these, along with a brief description, for those further interested, below. Comparing all existing methods is beyond the scope of this document, a survey about other methods can be found in [Coh00]. We have selected those that are most important for computer games for a closer description.

2.1.) Methods overview

Methods using occluder synthesis:

- LOD-algorithms are used for occlusion synthesis in [And00].
- Directional discretized occluders (DDO). [Bern00]

Methods using geometric methods:

- Coorg and Teller present a method in object-space that explores the visibility relationship between two objects [Coorg97]. A related method is shadow frustum culling [Hud97] and shadow volume BSP tree culling [Bitt98].
- Hierarchical view frustum culling (HVFC) is a widely used object-space method. [Assa00]
- Hierarchical back-face computation (HBFC) is another object-space algorithm that can be effectively used. [Kumar99]
- Prioritized-layered projection (PLP) is method that calculates the approximate visibility set. [Klos00]

Methods using graphics hardware / occlusion maps:

- The hierarchical Z-buffer (HZB) is a method that computes OC in image and object space. [Greene93]
- Hierarchical occlusion maps (HOM) are also used in image and object space for fast OC. [Zhang97]
- OpenGL-assisted OC describes another image-space method. [Bartz98]
- Occluder shadows are targeted for effective rendering in urban environments [Wonka99]

Of these listed methods we have chosen the HZB and HOM as effective solutions that support hardware acceleration for the image-space category, and HVFC and HBF as the most widely used object-space algorithms.

2.2.) Visibility methods in image-space

Hierarchical Z-Buffer (HZB) [Greene93]

The hierarchical Z-buffer is an improvement to the classic Z-buffer. While the classic approach only made use of image-space coherence, this new method also makes use of object-space and temporal coherence. We will describe the three main sections below:

Object-space oct-tree:

The scene is arranged in a classic oct-tree to allow for object-space coherence. We start by inserting the whole scene into a cube, which is then subdivided in 8 cubes, and so on. Our entire scene is now enclosed in hierarchically sized cubes.
We can use this structure to use view frustum culling (VFC) to remove the oct-tree nodes from the pipeline that lie outside our viewing frustum.

Image-space Z-pyramid:

The hierarchical Z-buffer arranges the Zbuffer in a pyramid of Z-buffers, where the one with the finest resolution lies at the bottom, and the one with the coarsest at the top (See Figure 2 for an occlusion map pyramid, which uses a similar type of hierarchy). While constructing this pyramid four values of the underlying buffer are combined to calculate the new value as follows: The new value will be the one Z-value of these four which is furthest away. If an object lies behind our new coarse Z-value, it is guaranteed that it will be behind all four values which led to this new Z-value.

Now we begin to check our oct-tree bounding box nodes against the finest Z-buffer level where the size of one Z-value covers our bounding box in screen space. If we find that the scan-converted cube lies behind our Z-value, then we can remove the bounding box and all contained geometric data from the pipeline immediately. If it is found to be visible we have to recursively test the child bounding boxes to prove that the box is hidden behind the pixel of this Z-buffer. If we continue like this we will in the end either prove that the entire oct-tree node is hidden, or we will find a visible pixel.

There are some difficulties with this method. First if we imagine a polygon lying exactly in the centre of our largest bounding box, we will always have to proceed to the finest level because the polygon will always lie in more than one octree node. The solution to this is either to clip the polygon in question, or to associate it with multiple cells.
Another difficulty is that once the cells become increasingly small the examined polygon may be bigger than the cell, and thus we have to find the nearest Z-value of a part of the vertex, which can be very time consuming.
A solution and simplification of this algorithm is to test against the nearest Z-value of the entire current polygon, even if only a part of it is enclosed in the current cell. The consequence is that the test is no longer definitive, but if we revert to classic scan-conversion if we fail to prove the polygon is hidden, we can avoid errors.

Temporal coherence can be achieved with the following method. In the next frame we start rendering all oct-tree cells that were visible in the last frame, and initialise our Z-pyramid with these values. Because our viewpoint will not have changed much, the Z-buffer test will be much faster. After rendering of the new frame, we update the temporal coherence list.

What is important about the HZB algorithm is that it relies on a graphics hardware extension to query if a node or polygon is visible or not. Until recently graphics hardware did not provide any means to get information out of the Z-buffer, so it was not really possible to implement a full HZB in a game. The most recent proceedings on this topic are presented in section 3.

In section 4 we will show how parts of this method can be used in computer games nonetheless, even if not all required features of the original algorithm are at our disposal.

Hierarchical Occlusion Mapping (HOM) [Zhang97]

Similar to the HZB, HOMs rely on image-space and object-space coherence. Contrary to the HZB though, overlap and depth testing is done in two different buffers.

The occlusion test is performed in two steps: Construction of the occlusion maps, and occlusion culling using the constructed maps. The very first step is once again VFC using the oct-tree structure. The hierarchical occlusion maps are built in succession.

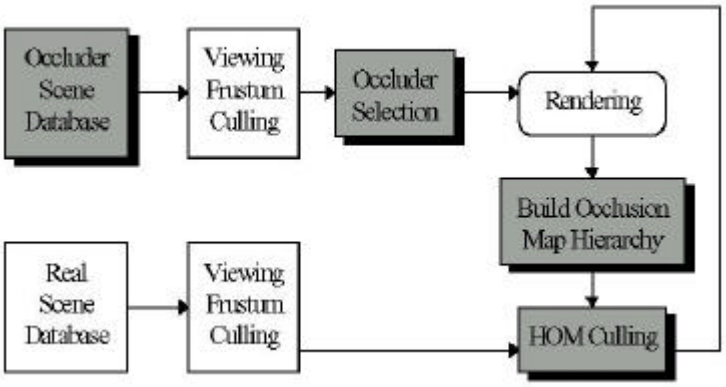


Figure 1: HOM graphics pipeline (From [Zhang97], p6)

HOMs use an occluder database to store potential occluders, which are selected in terms of size and position. These occluders are rendered in pure white on a black screen, and the rendered pixels represent the opacity of these occluders. In analogy to the HZB we construct an OM pyramid by scaling and filtering the occlusion maps. The results are levels of occlusion maps with coarser resolution, and greyscaled opacity values. (Light grey means that there is a lot of geometry in this screen-space area, dark grey that very little scene data exists). Furthermore the depth information of the occluders is stored in a separate depth estimation buffer, where the furthest Z-value of the corresponding bounding box is used along with its bounding screen-space rectangle.

What makes HOM interesting is that conventional texturemapping hardware can be used to filter and scale the initial occlusion map and thus to create the OM pyramid. We just have to use our OM as a texture, and set the texture-filtering parameters so that the hardware does correct bilinear filtering of the four contributing pixels.

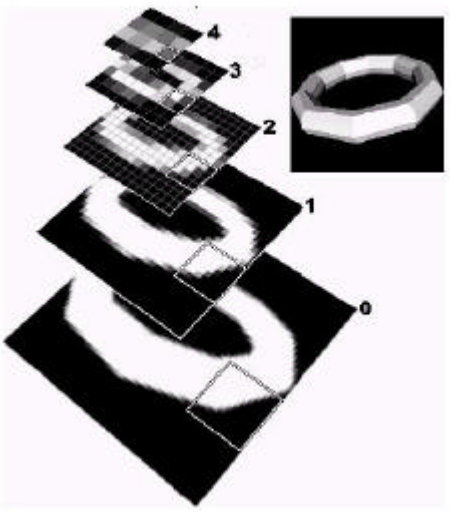


Figure 2: Occlusion maps hierarchy (From [Zhang97], p8)

A potential occludee is tested by projection of its bounding box onto the screen and testing against the HOM level where the size of one pixel roughly represents the same screen-space size. If the box is not behind a fully opaque region then we can not cull this box. Similar to the progression with the HZB we continue testing recursively through the hierarchy to detect occluded elements. What we can see here is that this method supports *approximate* visibility if we do not test against pure white, but a lower grey threshold - This will result in small errors in visibility, but for some applications like fast games this might be acceptable, and lead to significant speedups.

Once potential occlusion has been detected we have to find if the object is in front or behind our occluders, as the hierarchical occlusion maps do not store any depth information. We can use the information in our depth estimation buffer to determine this.

Selecting a suitable set of occluders from the occluder database (this has to be done each frame), can be very complicated, in fact, selecting the optimal occluders is the visibility problem itself. We can perform a simple selection by choosing our occluders according to their size, redundancy (some objects cause the same occlusions, like a clock on a wall, for example), complexity (very complex objects cause longer processing time) and z position.

Benefits of HOM are the use of existing hardware, and (as with the HZB) occluder fusion. Due to the large overhead HOM is not used much in games, especially if we have a look at recent hardware improvements (Section 3.) . The development of scenery in games, which is growing increasingly complex, could mean that HOM will become an interesting option in the future.

2.3.) Visibility methods in object-space

Hierarchical View Frustum Culling (HVFC) [Assa00]

View frustum culling on bounding boxes can be done in two different ways, either to perspective transform the bounding box to the perspective coordinate system, or to test the bounding box against the planes of the viewing frustum. HVFC chooses the latter approach.

As with the methods described above, HVFC uses multiple combined tests to speed up the classic VFC. The first step is to use a bounding box to test intersection into our viewing frustum. We can reduce the test to an examination of two points. These two points of the bounding box form the diagonal that is most closely aligned with the view frustums plane normal. (See Figure 3.).

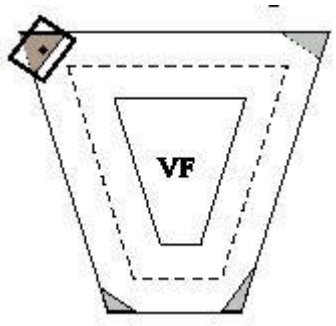


Figure 3: HVFC, intersection test.

If the outer point is inside the frustum, then the whole box is inside, if the inner point is outside, then the whole box is outside. For the rest we must assume intersection, and mark the volume as visible.

Additionally, we can make use of plane coherency: If an object was on the left side of the frustum, and we rotate our view to the right, we can conclude that the object will still be invisible the next frame.

View frustum testing can be accelerated even more if we divide our frustum in octants. Once we have determined in which octant our bounding box is located, we only have to test against the three outer planes of this octant. Furthermore masking can be used once we have determined that a bounding box is inside a plane:

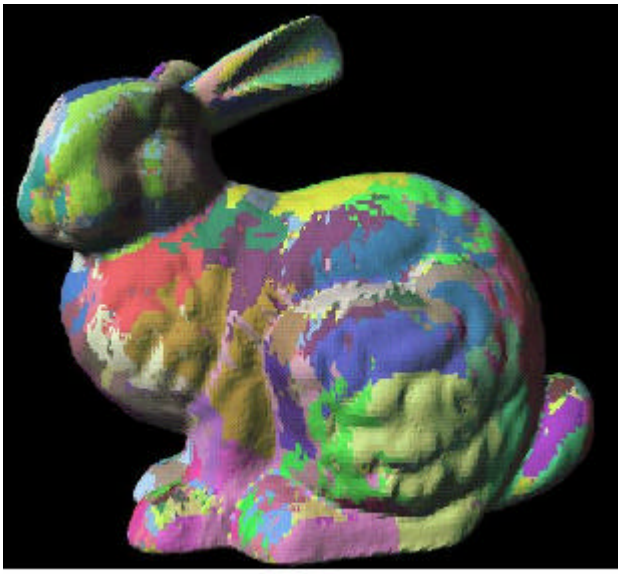
For the remaining tests of the BB children the tests for this plane can be masked and do not have to be performed.

Hierarchical Back-Face Culling (HBFC) [Kumar99]

HBFC groups polygons into clusters to accelerate the traditional back-face culling algorithm. One of the main drawbacks of the original approach was that the test had to be performed for each single polygon. If we can perform the test on groups of polygons, a significant acceleration is to be expected.

HBFC defines regions behind groups of polygons (Front region, back region and mixed region). If a cluster faces into the back region, then all vertices in this cluster are facing to the back region. If the cluster faces to the front region, all vertices face towards the front region. If the cluster is found to face towards the mixed region, some single vertices face to the front, and some to the back.

If we arrange the clusters in a hierarchical manner, BFC computational costs are reduced significantly, as only those clusters that face towards the mixed region need to be examined recursively.



(a) Bunny (372 clusters)

Figure 4: Clusters on a geometric model (represented by the same colour). (From [Kumar99], p7)

Polygons are assigned a cluster because of physical proximity, and proximity in the normal space.

3.) Occlusion Culling Hardware

We will give a feature overview in terms of visibility calculations on existing gaming platforms:

- PC/Mac/Unix

  Two big names dominate the PC hardware accelerator market: Nvidias GForce graphics chipsets and ATI's Radeon. Another architecture is a development by HP [Cunn00], but this has no importance in the gaming industry other than to function as a model for future development. At the moment current games can not use any of the methods that have been developed by HP.

  GeForce 2 occlusion features:

  - Z-buffer [RogZ], W-buffer (Linear Z-Buffer) [RogW].
  - Back-Face Culling

GeForce 3 occlusion features [Tom00]:

- Z-Buffer
- Lossless Z-Compression: 4:1 reduction of memory bandwidth
- Parts of a hierarchical Z-Buffer (called *Z Occlusion culling*): An absolutely new feature is that the application can request a Z-occlusion query from the hardware. This feature makes a client-side HZB a possible solution for a GeForce application.
- Back-Face Culling

ATI Radeon occlusion features [More00]:

- Z-Buffer
- Lossless Z-Compression: Similar to the GeForce 3 approach.
- Parts of a hierarchical Z-Buffer (called *Hyper-Z technology*): Contrary to the GeForce 3 ATI does not support occlusion queries.
- Fast Z-Clearing: Clearing the Z-Buffer is accelerated by a proprietary method.
- Back-Face Culling

- X-Box

  The X-Box uses Nvidias GeForce 3 graphic chip.

- Playstation 2

  - Z-Buffer

- Playstation 1

  - Back-Face Culling in 2D using the GTE (Geometry transfer engine).

- Dreamcast

  Segas Dreamcast uses the NEC Power-VR2 Chip, which uses a completely different approach (Infinitive Planes) to render primitives. Instead of rendering single vertices, polygons are rendered by using their bounding planes. This makes sure no pixel is rendered twice, thus removing the need for the Z-buffer. More details can be found in [SegW].

- Gamecube

  - Z-Buffer

4.) Examples and Conclusions

We will now evaluate the presented methods, especially in terms of support of existing hardware, and we will have a look at implementations in games.

Implemented methods:

On the PC platforms occlusion calculation aside from portals and PVS has only little acceptance. Games mostly rely on view frustum culling, and leave the rest to the 3d-hardware. Only in some very special cases does occlusion culling pay off. If we take a look why the industry shows slow acceptance it is due to four main reasons.

a.) Level of detail (LOD): While the original scene may contain thousands of polygons, the scene that is actually passed to the rendering pipeline is reduced significantly. Car models that consist of 2000 polygons are reduced to a simple box if they are just far away enough, as algorithms in games use a more aggressive scheme for LOD than standard rendering applications. Under these circumstances it is faster to render those few polygons which could be occluded instead of spending processing time with occlusion detection.

b.) Scenery: Often occlusion culling does not bring the desired speedups because the scenes are constructed modular or in some other specific way. Areas that would be visible are not dis played, because they have not been loaded yet. An example for this would be an area behind a hill that is loaded while passing through a valley and vice versa. While we are in one valley, the scenery in the other valley is simply not in memory. (This applies especially to older consoles with limited memory capacity). Another method that is used in some racing games: The course data is stored in a kind of hose, where we always know what will be coming next. Only the next few segments in this hose are rendered, so there is not much need for occlusion culling.

c.) Lack of large occluders: In many games the potential occluders are not large enough in comparison to the occludees. Occluders are often human figures, cars, ships, and so on. The scenery in the background is very large, and thus the occluder only occludes a few polygons on the background.

d.) Lack of time: Games are developed on a tight budget, and only the most biggest companies can afford programmers that delve into experiments that may work or may not. Most others have to stick to methods that are proven to work, and are fast and easy to implement.

On the other hand there are games around where occlusion culling can be a great benefit. Urban scenarios like cities are a good example. Buildings can be reduced to bounding boxes and can be tested against a coarse software Z-buffer for example. And we would also like to remark that all conclusions above do not apply to indoor games that can indeed gain a lot of speed through PVS calculations.

An interesting product in this context is Surrender 3D's dPVS, formerly known as Umbra, a visibility and occlusion culling library for PC, Dreamcast and PS2. The manual, which can be downloaded freely has a very interesting overview section about visibility algorithms [Aila01].

Most new hardware features for visibility calculations are implemented in a way that the programmer has no access or control over them. Methods like Z-buffer compression speed up the game without the need for additional programming. The only exception to this is the Z-buffer query supported by the GeForce 3, but this has proven to be too slow to allow a complete implementation of the HZB algorithm. The future will show if the GForce 4 or the Radeon successor will support this query fast enough for HZB to become a standard method.

Processing speed:

The game industry does only partly consist of the PC platform. Consoles like the Playstation 1, the N64 and others are still a big part, and they often do not have the processing power for complicated visibility calculations. Especially on these consoles view frustum culling and back-face culling are the most widely used methods. A method that can be used sometimes is a technique related to HBFC. The surface-normals of the vertices are stored into an 8-bit integer, and thus can be grouped if these coarsely quantised normals are equal. If one polygon fails the test, all others that have the same surface normal are discarded. This can lead to flickering, but in fast moving environments, it can be an acceptable drawback.

Related topics:

Other fields like radiosity, volumetric lights and shadow calculations will also benefit from work in the area of visibility computation, as they also relate to visibility from one point to another.

Conclusion:

Visibility computation will definitely become more important in the future of games, due to the growing complexity of current products. Nevertheless, due to the deadline pressure on development teams, programmers would be happy to have occlusion culling implemented in hardware that requires only little additional programming.

5.) Acknowledgements

We would like to thank Peter Wonka and Michael Wimmer of the Institute of Computer Graphics at the Technical University of Vienna for their helpful comments and insights.

6.) References

[Aila01] T.Aila, V. Miettinen, Surrender Umbra Reference Manual, Surrender3D, www.surrender3d.com/umbra/index.html

[And00] C. Andújar, C. Saona-Vázquez, I. Navazo. LOD visibility culling and occluder synthesis. Dept. LSI. Universitat Politècnica de Catalunya, Spain. July 2000

[Assa00] U. Assarsson and T. Möller. Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tool*s, 5(1), 2000.

[Bartz98] D. Bartz, M. Messner, and T. Httner. Extending graphics hardware for occlusion queries in openGL. In *Proc. Workshop on Graphics Hardware '9*8, pages 97– 104, 1998.

[Bern00] F. Bernardini, J. T. Klosowski, and J. El-Sana. Directional discretized occluders for accelerated occlusion culling. *Computer Graphics Foru*m, 19(3), 2000.

[Bitt98] J. Bittner, V. Havran, and P. Slavik. Hierarchical visibility culling with occlusion trees. In *Proceedings of Computer Graphics International '9*8, pages 207– 219, June 1998.

[Coh00] D. Cohen-Or, Y. Chrysanthou, C.T. Silva. A survey of visibility for walkthrough applications, School of Computer Science, Tel Aviv University, Submitted for publication, 2000

[Coorg97] S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. *1997 Symposium on Interactive 3D Graphic*s, pages 83– 90, April 1997.

[Cunn00] R. Cunniff. Visualize fx graphics scalable architecture. In *presentation at Hot3D Proceedings, part of Graphics Hardware Worksho*p, 2000.

[Greene93] N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. *Proceedings of SIGGRAPH 9*3, pages 231– 240, 1993.

[Hud97] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated occlusion culling using shadow frustra. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 1– 10, 1997.

[Klos00] J. T. Klosowski and C. T. Silva. The prioritized-layered projection algorithm for visible set estimation. *IEEE Transactions on Visualization and Computer Graphic*s, 6(2):108– 123, April - June 2000. ISSN 1077-2626.

[Kumar99] S. Kumar, D. Manocha, W. Garrett, and M. Lin. Hierarchical back-face computation. *Computers and Graphic*s, 23(5):681– 692, October 1999.

[More00] S. Morein. Ati radeon hyper-z technology. In *presentation at Hot3D Proceedings, part of Graphics Hardware Worksho*p, 2000.

[RogW] Doug Rogers. W-Buffering in Direct3D, NVIDIA Corporation, http://partners.nvidia.com/Marketing/Developer/DevRel.nsf/pages/628FAF6CE9234B0E8825685C000DD951

[RogZ] Doug Rogers. Z-buffering, Interpolation and More W-buffering, NVIDIA Corporation,

http://partners.nvidia.com/Marketing/Developer/DevRel.nsf/pages/990B69941E7C34E2882568770073BAE6

[SegW] Power VR2 revealed, http://www.infiniteplanes.net/segaworld/news/news120.htm

[Tom00] Thomas Pabst. High-Tech And Vertex Juggling - NVIDIA's New GeForce 3 GPU. Toms Hardware Guide, http://www7.tomshardware.com/graphic/01q1/010227/ , February 2001

[Wonka99] P. Wonka and D. Schmalstieg. Occluder shadows for fast wakthroughs of urban environments. In *Computer Graphics Forum*, volume 18, pages C51– C60. Eurographics Association and Blackwell Publishers Ltd 1999, 1999.

[Zhang97] H. Zhang, D. Manocha, T. Hudson, and K. Hoff. Visibility culling using hierarchical occlusion maps. In *Computer Graphics Proceedings of SIGGRAPH 97*, pages 77– 88, 1997.