Hidden Surface Removal and Hyper Z

OSKAR FORSSLUND
and JAN NORDBERG

KTH Computer Science
and Communication

Bachelor of Science Thesis
Stockholm, Sweden 2010

Hidden Surface Removal and Hyper Z

O S K A R F O R S S L U N D
a n d J A N N O R D B E R G

Bachelor's Thesis in Computer Science (15 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2010
Supervisor at CSC was Lars Kjelldahl
Examiner was Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/
forsslund_oskar_OCH_nordberg_jan_K10058.pdf

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Abstract

When rendering a scene to be shown on a computer screen, it is vital that the objects present in the scene are rendered so that objects at the front hide objects behind them. To make sure this is the case, hidden surface removal techniques are used. This paper gives an overview of techniques and optimizations in this area. It also describes an actual hardware implementation called Hyper Z, which is used by the graphics hardware manufacturer ATI.

Sammanfattning

När man renderar en scen för att visa på en datorskärm är det mycket viktigt att objekten som förekommer i scenen renderas så att de främre objekten döljer de bakre. För att garantera detta används tekniker för borttagning av skymda ytor. Denna uppsats ger en överblick över tekniker och optimeringar inom detta område och beskriver även en faktisk hårdvaruimplementation, Hyper Z, som används av grafikhårdvarutillverkaren ATI.

Contents

1 Introduction

Hidden surface removal techniques are used in computer graphics to speed up graphics rendering. The occlusion of objects that are hidden from the viewer, and the backsides of shown objects, lessens the strain on the computer. Over the years, a number of different approaches to the problem of doing this efficiently have been made, and in this paper we aim to make a comprehensive overview of the more commonly used techniques.

2 Background

Two major 1940s military projects can be seen as the launch of computer graphics development. These were the SAGE project in which the United States Military Forces funded and built a system for tracking incoming nuclear missiles, and the Whirlwind project, the building of the first flight simulator. Since then, the computers used to render graphics have gone from apartment-sized and military owned to pocket-sized and available to everyone. Nowadays, instead of being developed by the military, most state-of-the-art computer graphics are developed by the entertainment industry, be it computer game developers or movie makers.

Beginning with the most intuitive and obvious approach of rendering a multi-layered picture back to front, something that is known as the "painters algorithm", a number of graphics rendering improvements have been made. Even though the computers of today are immensely more powerful than the computers of the 1940s, the ever higher demands on computer graphics have necessitated the development of increasingly efficient graphics rendering algorithms.

The most widely used technique today is called Z-buffering. A buffer of all the x-y coordinate pairs of the display area with the distance (the z coordinate value) to the nearest object is maintained, and used to decide the color of each pixel. This combined with different culling techniques, i.e. the dismissal of non-visible objects, is the backbone of modern computer graphics rendering.

3 Classic Techniques

In March 1974 I.E. Sutherland, R.F. Sproull and R.A. Schumacker released the paper *A Characterization of Ten Hidden-Surface Algorithms*. A part of that paper was dedicated to describing available hidden surface algorithms used at that time, much like what we are hoping to accomplish with this paper. Therefore, we will include here a simplified summary of said techniques since many of the ideas of that time are still being used in various forms today.

3.1 Object-Space Algorithms

In their paper [8], Sutherland et al., divides the ten algorithms into three categories of which *object-space algorithms* is the first.
Object-space calculations are made to be as precise as possible and, more or less, compute the whole virtual reality of the picture as opposed to *image-space calculations* which instead focus on the display area and how to represent pictures on-screen.

Object-space algorithms are further divided into the algorithm of L.G. Roberts and the edge-intersection algorithms of Appel et al.

3.1.1 L G Roberts

Roberts' algorithm was the first known solution to the hidden-line problem, i.e. the hidden surface problem restricted to the edges of objects [8].
The algorithm tests which parts of a relevant edge, i.e an edge possibly visible to the viewer [8], are hidden by other objects. This is done by tracing a line from different points on the edge towards the observation point. If any point on the line lies within another object, the point from which the line originated is obviously hidden.
In this way all the relevant edges of all objects are tested against all other objects to see exactly which parts of them are hidden.

3.1.2 Edge-Intersection Algorithms

Sutherland et al. [8] describe four edge intersection algorithms, all of which are rather similar as they all focus on calculating the cumulative visibility scores of vertices, i.e. the number of faces hiding the vertices. Although managed differently in the four algorithms, this is generally done by first calculating the visibility score of an initial vertex of an edge. After this the visibility scores of all points lying on edges emanating from this vertex can easily be calculated.
The idea is that the visibility score along an edge will only change if, and when, the edge intersects a *contour edge*, i.e. an edge between one visible and one non-visible surface, in the two-dimensional projection of the model.

Further the score will only change by either +1 or -1 and only if the intersecting edge is determined to be closer to the point of view. Thus at each such intersection an edge can be divided into segments and all points lying on an edge segment will have the same visibility score.

3.2 List-Priority and Image-Space

List-priority and image-space algorithms differ from the object space algorithms by focusing on producing images and not a mathematical representation of a model. The difference between list-priority and image-space algorithms is that list-priority algorithms make computations in the object-space of the model and thus would be able to correctly display a picture of the model with arbitrarily high resolution, was there a suitable display. Image-space algorithms on the other hand make calculations from the image-space of the model and, although more efficient, would not be able to produce such high-quality results. Then again, that is not their purpose. Image-space algorithms are designed to produce an image with a known resolution.

3.2.1 List-Priority Algorithms

Both list-priority algorithms described by Sutherland et al. [8] are centered around a list of rendering priorities of the faces that make up the model. Once priorities are established the model is rendered so that faces with higher priority will hide those with lower priority. The difference between the two algorithms lies mainly in how the priorities are calculated for each frame that is to be displayed.

The first algorithm, devised by Schumacker et al. [8], uses two kinds of priorities: cluster and face-priority. The first is calculated by dividing the model into linearly separable clusters and numbering them. A cluster-priority can then be calculated for every cluster depending on the position of the viewpoint. After dividing the model into clusters, each face within each cluster is given a priority in such a way that if face A can hide face B from any viewpoint then face A should have a higher priority. This assigning of priorities is done ignoring back faces, i.e. faces facing away from the currently tested viewpoint, as these faces will never be visible from that viewpoint.
The cost of producing this kind of priority list is relatively high. The advantage though is that the face-priority only ever has do be calculated once and can then be reused as long as the clusters stay the same. This makes the algorithm well suited for static environments where only the viewpoint is changed.

The second algorithm, devised by Newell et al. [8], calculates only face-priority. This is done by first ordering the relevant faces by how far their furthest vertex is from the viewpoint depth-wise. The faces are then com-

pared back to front to see if the ordering was successful. A face is deemed to be ordered correctly if it cannot possibly hide any face with a higher priority. If this can not be done merely by re-ordering then at least one face has to be split and then the pieces are re-tested individually.

3.2.2 Depth-Priority Algorithms

These algorithms focus on calculating what shade a particular area of an image should have. The algorithms work by dividing the picture into homogeneous areas, or windows as they are called [8], and then calculating the shades of the separate windows. An area is tested to be a window and if either no faces intersect the area, or there is a *critical surrounder*, i.e. a face that covers the area and is closer to the viewpoint than any other face intersecting or covering the area, then the area is accepted as a window. If this is not the case the area is divided into smaller test areas and the process is repeated with those. Ultimately, if a critical surrounder can not be found in a certain number of subdivisions, the test will be given an average shade of the faces intersecting it.
One advantage with this method is that not all faces have to be tested against the smaller test areas. This is made possible by the possibility to store information on faces that either covered the original test area, or were completely disjoint from it. Naturally, these faces will have the same relation to the smaller test area as they had to the original.

3.2.3 Scan-Line Algorithms

These algorithms work by calculating what faces are visible along horizontal lines [8]. This is done by first ordering the faces in the picture in the xy-plane. Then the image is scanned top to bottom and for each horizontal scan-line a series of depth tests are made to see what face is showing for different parts of the scan line. The points on the scan-line where the depth-tests are performed are determined differently by different algorithms but one common way is to test every edge that intersects the scan-line. The nearest face at this point will be visible.

4 Techniques Used Today

A lot has happened in the world of computer graphics since Sutherland et al. published their paper in 1974. The biggest difference however, lies in the hardware used to render the graphics.

In the following section we will discuss some techniques that have been around for a while but are central in modern hidden surface removal. Z-buffering for instance, appeared in Edwin Catmull's Ph.D. thesis from 1974 [2] and can now be found implemented in nearly all graphics hardware.

4.1 Z-buffering

When drawing an object on screen, the graphics device stores the depth value (or z value) of a generated pixel in a buffer, aptly named the Z-buffer. For every new object drawn, the depth of each pixel is compared to the stored value of that pixel in the buffer, and the corresponding pixel of the object is either rejected for being further away than the pixel of the closest object so far, or drawn onto the screen and written to the Z-buffer. This image-based approach is currently implemented in the hardware of most graphics cards, and the computations involved can be carried out very rapidly. The precision of the buffer has to be sufficiently high to avoid a phenomenon called Z-fighting, where two objects that are very close to each other can appear to blink in and out of each other, due to the resolution of the buffer. Transparent objects are also a problem with Z-buffering techniques; if the object at the front is translucent, we would need to draw the object that is second from the front, so it can be seen through the front object. The easy fix to this is to draw the translucent objects after you are done with the opaque objects. This solves most of the problems, but if you try to render a translucent polygon behind another translucent one you will not see through both of them. Usually, there are so few translucent objects that one can ignore this anomaly.

4.2 Culling

According to the Oxford Dictionary of English, to *cull* is to pick out something and discard it as inferior. In computer graphics, culling is used to reduce the number of polygons being rendered, due to the simple fact that something you can not see does not have to be rendered. We will here cover three different types of culling.

4.2.1 View Frustum Culling

A viewing frustum is essentially six clipping planes bounding the part of the model that the viewer can currently see. Obviously, all parts of the model not inside the viewing frustum can be discarded as they are not in view [1].

If an object lies partly within the frustum it will be clipped and only the visible part will be rendered.

In order to speed up this process, one can organize all objects in a binary space partitioning tree (or BSP tree). This data structure recursively organizes objects within space by treating each polygon as a cutting plane, which is used to decide whether the other objects are in front or behind that plane. This structure can be quickly traversed when deciding which objects to cull.

4.2.2 Back-face Culling

Any surface whose normal is facing away from the viewer can be discarded, since these surfaces can not be seen. Such surfaces can easily be detected by calculating whether the dot product of the surface normal and the viewer vector is negative. Generally, almost half of the surfaces in a model can be discarded in this way, theoretically cutting the rendering time in half.

4.2.3 Occlusion Culling

Objects that can not be seen due to them being behind other objects can also be removed from the rendering pipeline. A number of different approaches to identifying these hidden objects exist, one of which is called Potentially Visible Sets (PVS). PVS is implemented by precomputing a candidate set of potentially visible polygons. These are then indexed at run-time to obtain an estimate of the visible geometry. Binary space partitioning schemes reduce the time needed for these calculations.

Another approach is Portal Rendering. In Portal Rendering, the object space is divided into different sectors, connected by shared polygons called portals. If at any moment a portal is not visible to the viewer, the whole sector connected to it can be discarded, otherwise the portal can be used as a viewing frustum to the connected sector. Portal rendering is very efficient in indoor environments, but an open landscape might not benefit much from this approach, since there are no obvious portals separating different parts of the landscape.

4.2.4 Contribution Culling

Contribution culling is the process of discarding objects that will be very small on-screen. This is usually determined by registering the size of the objects projection on the image plane and comparing it to a certain threshold value as these objects probably would not contribute very much to the finished frame but take up valuable rendering resources anyway. [4]

5 Hyper Z - A Hardware Implementation

There are several ways hidden surface removal can be implemented, both in hardware and in software, and we will not be able to give a full summary of all those techniques. What we aim for is to give an actual example of how hidden surface removal can be implemented.

Nowadays, much of the hidden surface removal can be managed by hardware. The development of powerful graphic processing units have somewhat shifted the focus of software designers from implementing smart rendering algorithms to synchronizing graphical applications with the rendering hardware. Naturally, there is still much to be gained from clever software solutions but taking advantage of the hardware is key in avoiding bottlenecks in the graphics rendering pipeline.

The ATI Hyper Z technique is a hardware implementation found in the ATI Radeon graphic cards. This, in turn, is comprised of four techniques: Hierarchical Z, Early Z, Z-buffer Compression and Fast Z Clear. [5]

5.1 Hierarchical Z

Hierarchical Z was originally developed by Ned Greene [3], and used several Z-buffers with different resolutions to determine an objects visibility. Greene implemented this with octrees[1], but ATI has chosen a somewhat simpler approach. They store a reference z value for every 8x8 pixel block of the buffer by finding the deepest value of all the pixels in the block, thus creating a low resolution version of a Z-buffer. It is then used to make a rough initial visibility estimate for all the objects of a scene. All the objects not guaranteed to be invisible are then passed on to the Early Z scan. [5]

5.2 Early Z

In a traditional graphics rendering pipeline, texturing of a pixel is done when writing a pixel to the color buffer. A depth test is performed by checking the Z-buffer to determine if the pixel is visible. If the z value of the pixel is greater, it is behind some previously rendered object and should be discarded. If the z value is less than the z value of the buffer, the pixel is visible, and should be written to the color buffer. This means that some pixels are rendered unnecessarily as they are overwritten moments after rendering by some other pixel deemed to be in front of it. Obviously this is a waste of time that could be hindered if the check for visibility was performed prior to the rendering of the pixel.

[1] An octree is a BSP data structure in which each node has up to eight children. It is mostly used to divide three-dimensional space by recursively subdividing it into eight octants

Early Z does exactly this, by performing an extra z compare before the texturing. Early Z operates per pixel, as opposed to the rougher Hierarchical Z test [5].

It is important to understand, that even if we have an Early Z check in the beginning of the graphics rendering pipeline, we still need to perform a Z check later in the pipeline as well. For instance, a pixel shader might change the z value of a pixel [7].

5.3 Z-buffer Compression

Z-buffer compression is used to cut down on the bandwidth used when using Z-buffers. This is managed in two ways of which the most obvious one is that compressed data is smaller than uncompressed data and therefore any compression made will save bandwidth. This compression, however, is subject to some restrictions [9].
The first restriction is that compression must be lossless since all changes in depth precision might affect the resulting picture. Second, the compression ratio must be fixed as the size of the blocks of data to be read from memory must be known.
The compression method used by ATI is called Differential Differential Pulse Code Modulation (DDPCM) and operates on 8x8 pixel blocks. In stead of storing actual depth values, the DDPCM stores a reference value and a set of differentials, capitalizing on the fact that all points belonging to a triangle will lie in the same plane [9]. An obvious problem with this method is that all the pixels in the block must belong to the same triangle for the compression to work. As this is not always the case memory must be allocated to handle uncompressed data resulting in the fact that the compression does not reduce the amount of memory allocated. However, the compression does reduce bandwidth load as a flag is maintained for each block signaling which state the block is in. This can either be compressed, uncompressed or cleared, letting the hardware know when to read compressed data to save bandwidth. This also allows another optimization; when a block is flagged as clear the data does not have to be read at all!

5.4 Fast Z Clear

After each picture frame is drawn, the Z-buffer needs to be cleared for the next frame. This can be done by writing some clear-value to every slot in the buffer but that takes a lot of time. Thanks to the Z-buffer Compression there is another, more efficient way of doing this.
As mentioned above, there is a flag maintained for every 8x8 pixel block. This flag is stored in a lookup table and can be easily accessed. This makes clearing the Z-buffer very fast as setting the compression flags to 'clear' effectively clears the whole Z-buffer. [6]

6 Summary

We have had a lot of fun researching the different topics of hidden surface removal. There is a wealth of information available on the Internet, as well as in different books and papers published by esteemed researchers both from different universities, as well as from the two major graphics hardware manufacturers, Nvidia and ATI. One is often lead astray down the winding paths of the marvellous land of computer graphics when confronted with topics such as shadow rendering, depth-of-field simulation or lens flare implementations to name but a few. Hopefully, readers of this paper will also be encouraged to learn more in this diverse field.

7 References

References

[1] Ulf Assarsson and Tomas Möller. "Optimized view frustum culling algorithms for bounding boxes". In: *J. Graph. Tools* 5.1 (2000), pp. 9–22. ISSN: 1086-7651.

[2] Edwin Earl Catmull. "A subdivision algorithm for computer display of curved surfaces". Ph.D. Thesis. The University of Utah, 1974.

[3] Ned Greene, Michael Kass, and Gavin Miller. "Hierarchical Z-buffer visibility". In: *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. Anaheim, CA: ACM, 1993, pp. 231–238. ISBN: 0-89791-601-8.

[4] Jong-Seung Park and Bum-Jong Lee. *Hierarchical Contribution Culling for Fast Rendering of Complex Scenes*. 2006. URL: http://www.mee.chu.edu.tw/labweb/psivt2006/papers/4319/43191324.pdf.

[5] Emil Persson. *Depth In-Depth*. 2007. URL: http://developer.amd.com/media/gpu_assets/Depth_in-depth.pdf.

[6] Guennadi Riguer. *Performance Optimization Techniques for ATI Graphics Hardware with DirectX® 9.0*. 2002. URL: http://ati.amd.com/developer/dx9/ATI-DX9_Optimization.pdf.

[7] Firing Squad. *FS Guide: Occlusion Culling*. 2002. URL: firingsquad.com/guides/occlusionculling.

[8] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. "A Characterization of Ten Hidden-Surface Algorithms". In: *ACM Comput. Surv.* 6.1 (1974), pp. 1–55. ISSN: 0360-0300.

[9] Per Wennersten. "Depth Buffer Compression". Master Thesis. Royal Institute of Technology, 2007. URL: http://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2007/rapporter07/wennersten_per_07121.pdf.