Delay Streams for Graphics Hardware

Timo Aila*
Helsinki University of Technology and
Hybrid Graphics, Ltd.

Ville Miettinen†
Hybrid Graphics, Ltd. and
University of Helsinki

Petri Nordlund‡
Bitboys Oy

Abstract

In causal processes decisions do not depend on future data. Many well-known problems, such as occlusion culling, order-independent transparency and edge antialiasing cannot be properly solved using the traditional causal rendering architectures, because future data may change the interpretation of current events.

We propose adding a *delay stream* between the vertex and pixel processing units. While a triangle resides in the delay stream, subsequent triangles generate occlusion information. As a result, the triangle may be culled by primitives that were submitted *after* it. We show two- to fourfold efficiency improvements in pixel processing and video memory bandwidth usage in common benchmark scenes. We also demonstrate how the memory requirements of order-independent transparency can be substantially reduced by using delay streams. Finally, we describe how discontinuity edges can be detected in hardware. Previously used heuristics for collapsing samples in adaptive supersampling are thus replaced by connectivity information.

CR Categories: I.3.1 [COMPUTER GRAPHICS]: Hardware Architecture—Graphics Processors; I.3.3 [COMPUTER GRAPHICS]: Picture/Image Generation—Display algorithms; I.3.7 [COMPUTER GRAPHICS]: Three-Dimensional Graphics and Realism—Hidden line/surface removal

Keywords: 3D graphics hardware, occlusion culling, order-independent transparency, antialiasing, stream processing

1 Introduction

Modern consumer-level graphics cards have video memory bandwidths of almost 20GB/s. Still the bottleneck in real-time rendering applications, such as state-of-the-art computer games, is the available fill rate rather than the geometry processing power. Both the number of pixels and the cost of rendering individual pixels have risen dramatically. Screen resolutions of 1600x1200 pixels are not uncommon and frame buffers have high dynamic ranges. Interactive 3D applications are beginning to compute the illumination using complex pixel shader programs. Shadows improve the visual realism but increase the fill rate requirements even further. Realistic scenes may have millions of triangles and a high depth complexity,

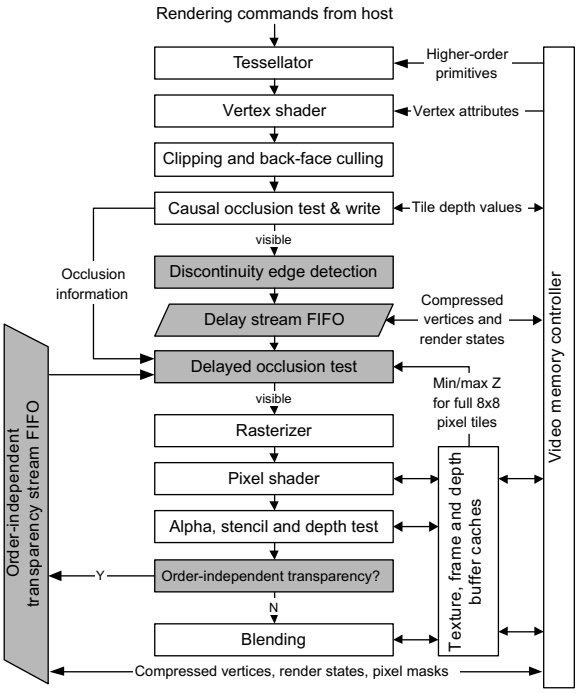


Figure 1: Architecture of a modern, DirectX9-level graphics hardware. We introduce several relatively minor modifications (marked in gray) into the hardware design for reducing the bandwidth requirements and improving the overall performance. The most important addition is the delay stream where triangles are placed after geometry processing.

i.e., the average number of times each pixel is drawn. Detailed geometry consumes more memory and is prone to aliasing. Demands for antialiasing in turn increase the number of samples used for each pixel.

A number of approaches are used for reducing the memory bandwidth and pixel processing work. Triangle data is stored in the video memory or generated on-the-fly from displacement maps and higher-order primitives. Textures are stored in compressed formats. Color and depth values of pixels are cached on-chip and compressed prior to transmission to video memory. Antialiasing can be limited to the edges of triangles. Separate occlusion culling units have been introduced for avoiding rasterization of hidden primitives and pixels. These units also supply visibility information back to the application so that objects that are entirely hidden do not have to be sent to the hardware.

Established rendering semantics dictate that triangles must be rasterized in the order they are submitted. As the hardware has no global information about the scene, the effectiveness of certain important algorithms is limited. Transparent triangles must be sorted in order to render them correctly. However, hardware-generated triangles cannot be sorted by the application. Also, occlusion culling would have a greater impact if triangles that *are going to be* in front could be used to occlude triangles submitted before them.

*timo@tml.hut.fi
†wili@hybrid.fi
‡petri.nordlund@bitboys.com

Contributions This paper concentrates on optimizing the pixel processing performance and bandwidth requirements of certain key areas in consumer graphics hardware. Our main contribution is using the concept of delay streams for increasing rendering performance (see Figure 1). By introducing a delay between the geometry processing and rasterization stages of a graphics pipeline, we enhance load-balancing and gather knowledge about the global scene structure. This information is used for improving the performance of several algorithms:

1. We introduce an additional occlusion culling test after the delay to further reduce depth complexity. Compared to existing hardware implementations [Morein 2000] our approach renders 1.8–4 times fewer pixels in our test scenes. In fill rate bounded applications this translates almost directly to the frame rate (Section 3).
2. We improve existing work [Wittenbrink 2001] for rendering order-independent transparency. In our test scenes the storage requirements are an order of magnitude smaller than in the previous method (Section 4).
3. We demonstrate a new hardware algorithm that can identify silhouette edges of objects using geometric hashing. This information is used to reduce the number of pixels that are antialiased. For our test models this approach supersamples only 0.3–2.1% of the screen pixels (Section 5).

We have implemented the delay stream and occlusion culling units in VHDL. The other components have been simulated using behavioral models. We present results for all three algorithms along with bandwidth measurements and compare them to existing approaches in several publicly available test scenes and industry-standard benchmark applications.

2 Related Work

Occlusion culling Greene and Kass [1993] avoid most per-pixel depth comparisons by organizing the depth buffer into a hierarchy. The scene is represented as an octree that provides an approximate front-to-back traversal. For each octree node the *occlusion query* 'Would this node contribute to the final image if rendered?' is asked. If not, all triangles inside the node and its children can be skipped. Occlusion queries were first supported in Denali GB graphics hardware [Greene and Kass 1993]. Extending OpenGL to handle occlusion queries is discussed by Bartz *et al.* [1998]. The queries are now implemented in consumer graphics hardware, e.g., NVIDIA GeForce3. Meißner *et al.* [2001] extend the queries by storing visibility masks of bounding volumes for subsequent culling of individual triangles and blocks of pixels. The use of hardware occlusion queries has been recently examined by Klosowski and Silva [2001] and Hillesland *et al.* [2002].

Maintaining a full depth buffer hierarchy is a challenging problem and several approaches have been proposed. Xie and Shantz [1999] update the hierarchy only a few times per frame according to a heuristic. ATI's HyperZ incorporates a two-level depth pyramid that provides fast depth buffer clears and adds an *occlusion test* stage into the traditional rendering pipeline [Morein 2000]. Their early depth rejection works best when the input primitives arrive in approximate front-to-back order. Pixel shading work can be reduced by rendering the scene twice: the first pass constructs the depth buffer and the second applies shading for the visible pixels. Our approach performs implicitly such deferred shading [Deering et al. 1988].

PowerVR [2000] captures the geometry of the entire frame and uses tile-based rendering with internal buffers. Occlusion culling is implemented by using on-chip sorting and thus the culling efficiency is not affected by the order of the input primitives. The majority of bandwidth problems are avoided but the limited amount of video memory makes capturing large scenes impractical.

Several experimental architectures have been proposed. SaarCOR [Schmittler et al. 2002] uses ray casting [Appel 1968] for determining the visible surfaces and performs occlusion culling implicitly. A few architectures consist of multiple rasterization nodes and create the final image by using composition [Fuchs et al. 1989; Molnar et al. 1992; Torborg and Kajiya 1996; Deering and Naegle 2002]. The sort-last nature of these architectures makes the issue of occlusion culling difficult to address efficiently.

Zhang *et al.* [1997] note that an occlusion query can be split into two sub-tests: one for coverage and one for depth. The result is conservatively correct, as long as the coverage test is performed using full accuracy. Their lower resolution depth test uses a *depth estimation buffer* (DEB) which conserves memory by storing a single conservative depth value for a block of pixels. We utilize DEB in our occlusion culling unit.

Durand [1999] provides a comprehensive survey on visibility determination. Akenine-Möller and Haines [2002] cover modern hardware occlusion culling algorithms as well as application-level culling techniques.

Order-independent transparency In order to correctly handle all OpenGL [1999] and DirectX [2002] blending modes, the visible transparent surfaces have to be blended in a back-to-front order. Techniques such as volume splatting [Westover 1990] and surface splatting [Zwicker et al. 2001] use transparent surfaces extensively.

The Z-buffer algorithm cannot handle transparent surfaces correctly [Catmull 1974]. To overcome this limitation the A-buffer algorithm stores a linked list of fragments for each pixel [Carpenter 1984]. The A-buffer also incorporates subpixel coverage masks for antialiasing of triangle edges. The final color of a pixel is computed by first sorting the fragments according to increasing depth and then recursively blending the fragments from back to front using accumulated coverage masks. Molnar *et al.* [1992] discuss difficulties of implementing an A-buffer in hardware. Nevertheless, a few implementations exist [Chauvin 1994; Lee and Kim 2000]. The handling of interpenetrating surfaces is improved by Schilling *et al.* [1993] and Jouppi and Chang [1999]. The latter describe an architecture that maintains as few as three active fragments per pixel and creates convincing images with a reasonable storage cost. However, combinations of different blending modes are not considered.

Mammen [1989] and more recently Everitt [2001] describe a multi-pass algorithm that *peels* transparent layers one at a time and blends them to the frame buffer. Two depth buffers are used concurrently for determining the farthest unprocessed transparent surface for each pixel. The process is repeated until all transparent surfaces have been blended. Order-independent transparency is computed correctly since the per-pixel sorting is performed implicitly using selection sort. Transparent surfaces need to be buffered by the application. Also the rendering time is bounded by the depth complexity of the transparent surfaces regardless of their visibility. Winner *et al.* [1997] describe a peeling variant of A-buffer using custom hardware.

Wittenbrink [2001] proposes performing the peeling operation fully in hardware. For each pixel the closest opaque fragment is placed into the frame buffer and all transparent fragments that could not be culled are stored into a separate *recirculating fragment buffer* (R-buffer). The peeling operation resembles that of Mammen. Wittenbrink draws the scene from front to back and optionally supports A-buffer antialiasing. Different blending modes are not considered. Unlike Mammen's algorithm, only a single geometry pass is required and the peeling operation is faster since only the unprocessed pixels are retained in the R-buffer. Mark and Proudfoot [2001] propose a fragment stream buffer similar to R-buffer for

practical multi-pass rendering. Our approach builds on the work of Wittenbrink but stores a geometric representation instead of shaded fragments.

Antialiasing The four major sources that contribute to aliasing in rendered images are discontinuity edges, shader undersampling, intersecting surfaces and objects falling between sample points [Crow 1977; Sander et al. 2001]. Although all aliasing artifacts can be reduced by sampling more densely, supersampling every pixel of an image can be uneconomical.

The sampling quality can be improved by using jittered sample positions [Cook et al. 1984] or by sparse supersampling [Akeley 1993]. An alternative solution that requires less memory but has an increased geometry processing cost is to render the scene multiple times with slight jittering and to accumulate the color buffer contents of each pass into a separate accumulation buffer [Haeberli and Akeley 1990]. Deering and Naegle [2002] describe a scalable high-end graphics architecture producing high-quality antialiasing. In commodity hardware, performance improvements can be achieved by adaptive supersampling, i.e., concentrating most efforts to pixels that are likely to alias [Crow 1977; Carpenter 1984; Lau 1995]. This approach has been used in Matrox Parhelia [2002] and 3Dlabs Wildcat 4110 [2002] graphics cards as well as in our method.

Discontinuity edges are classified into silhouette edges at the boundaries and creases of objects, and sharp edges that separate connected triangles with different materials or vertex attributes. Detecting discontinuity edges was proposed by Crow [1977]. His method requires a full scene capture and storing additional edge connectivity information. Sauer *et al.* [1999] perform antialiasing as a post-process by blending adjacent pixels at discontinuity edges. Sander *et al.* [2001] find discontinuity edges by using a hierarchical search structure and render them in a back-to-front order with antialiased lines. The approach is challenging to implement fully in hardware and cannot efficiently support deformable models. Our implementation detects discontinuity edges in hardware and performs supersampling only for pixels intersected by them.

The effects of undersampling in textures are usually reduced by filtering methods such as mip-mapping [Williams 1983] and anisotropic filtering. Detecting intersecting surfaces is difficult and requires that additional information, such as triangle plane equations, is stored in the depth buffer [Carpenter 1984; Jouppi and Chang 1999]. For static models a preprocessing pass is commonly used for clipping the intersecting triangles. The problem of subpixel size objects can be somewhat reduced by level-of-detail mechanisms or by explicitly detecting them by using bounding volume information and increasing the sampling rate locally [Whitted 1980].

3 Delayed Occlusion Culling

In this section we explain how a delay stream can be used for improving the efficiency of occlusion culling hardware.

3.1 Causality and Delay

The white blocks in Figure 1 outline a common architecture employed in current consumer graphics hardware. Occlusion information is generated and occlusion tests are performed for the input primitives *in the order they arrive* from the geometry-processing unit. In these causal architectures optimal performance is achieved when the input triangles are sorted by the application (Figure 2). Our finding, based on discussions with the middleware and games development community, as well as traces made from commercial applications, is that spatial sorting is generally extremely coarse if done at all. Most applications order primitives and objects to minimize the amount of state changes of shaders and textures. This goal partially conflicts with the requirements for spatial sorting. The ideal approach would be to let the application perform coarse spatial sorting and optimize the input data based on state changes. The hardware then resolves the local ordering issues.

Figure 2: A simple scene consisting of four opaque primitives (ABCD). The optimal solution is to render only A. Without occlusion culling all four primitives are rendered. In *causal* occlusion culling architectures, no matter how accurate, only the primitives that have already been processed can hide the current primitive. With the input order ABCD only A is rendered whereas with the order DCBA all four have to be rendered. With our architecture the input order affects the efficiency only weakly. In this example only A is rasterized regardless of the input order.

In our delayed occlusion culling a triangle that is visible after the first occlusion test is not immediately rendered. Instead, it is used for generating occlusion information and appended into a delay stream FIFO. While the primitive resides in the delay stream, more primitives generate occlusion information. When the triangle emerges from the delay stream, the occlusion test is performed again. Since substantially more occlusion information about the scene has been collected during the delay, there is a much greater chance for a hidden primitive to be culled. The rendering order of the triangles is not altered by this method.

In practice the per-pixel depth tests are often performed before the pixel shader to reduce the amount of generated texture bandwidth and pixel shader work. If the primitives of Figure 2 are submitted in back-to-front order, the Z-buffer algorithm will have to execute the pixel shader four times for each pixel even if causal occlusion tests are used. Delayed occlusion culling on the other hand is able to reject pixels that will be hidden in the future and would thus execute the pixel shader only for the closest primitive.

The delay stream does not capture the entire scene. Our tests suggest that a sliding window of 50K–150K triangles is sufficient to reduce the depth complexity close to the theoretical optimum. The culling efficiency degrades gracefully when only a part of the geometry fits into the delay stream. In this case the ordering of the input affects weakly the resulting depth complexity.

The use of a delay may slightly increase the latency of rendering a frame. The pixel units start processing the stream after it is 40% full or a flush command is received. The remaining 60% acts as an efficient load-balancing mechanism between the geometry and the pixel processing units in fill rate bounded scenes. The performance of both units is improved, as congestion in one of them does not immediately reflect to the other. The amount of buffering provided by a delay stream is several orders of magnitude larger than that of typical on-chip FIFOs. The delay stream does not require double-buffering as an end-of-frame marker is injected into the stream. However, to allow continuous processing between two frames, any occlusion information has to be double-buffered.

3.2 Delay Stream Implementation

Our delay stream contains compressed vertices and render state changes, i.e., everything that is needed for rasterizing the geometry correctly. The size of the stream is configurable and the physical implementation is a circular buffer (FIFO) in video memory.

Before a vertex is inserted into the delay stream, it is compressed by using a simple algorithm based on history data. Four distinct previous values are maintained for each vertex attribute, e.g., screen-space position, in vertex-wide registers. If an attribute of the current

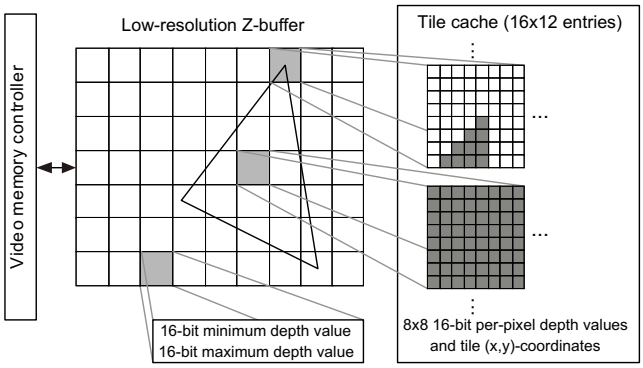


Figure 3: Our occlusion culling unit subdivides the screen into 8x8 pixel tiles. The closest and farthest depth value of each tile are stored into a low-resolution Z-buffer. A separate tile cache contains per-pixel depth information for 192 recently accessed tiles. All occlusion writes are performed into the tile cache. When a cached tile needs to be replaced, its minimum and maximum depth values are scanned and the LRZ-buffer is updated.

vertex is found from the history buffer, a two-bit index is stored instead of the raw data. Also, if the entire vertex is found from a certain slot, only a single index is stored. Usually only a small fraction of the vertex attributes are active at a time and need to be stored into the delay stream. Most of the time two 2D texture coordinate sets are sufficient as the same sets are reused to access multiple textures. When per-pixel lighting is used, vertex normals are stored instead of the diffuse and specular vertex colors. Also, widely adopted techniques such as normal maps move data from vertices into textures [Peercy et al. 1997].

In our test scenes the compressed triangles consume 25–65 bytes each. Ordering the input data coherently improves the compression rate [Hoppe 1999]. Using a longer history buffer or sophisticated compression schemes would further reduce the amount of data at the expense of more complex hardware.

3.3 Occlusion Tests

Occlusion tests are commonly used for removing entirely hidden triangles or hidden parts of them prior to the per-pixel depth tests. This test is often implemented by using a *low-resolution Z-buffer* (LRZ-buffer) that subdivides the screen into tiles of fixed size. Each tile corresponds to a region in the depth buffer and contains the farthest depth value of that region. Conservative occlusion tests can be made by processing triangles in tile-sized chunks and comparing the closest depth value of the chunk against the depth value stored in the corresponding tile. If the triangle chunk is hidden, none of its pixels need to be processed [Akenine-Möller and Haines 2002].

The set of tiles is either fully or partially stored in on-chip memory. As large parts of the scene can be conservatively culled using this information, fewer accesses need to be made to the depth buffer stored in video memory. In most implementations the depth information for the tiles is generated as a side-product of actual rasterization. When a pixel in the depth buffer is modified, the corresponding tile is updated.

Occlusion information cannot be generated for primitives that use per-pixel rejection tests, e.g., alpha or stencil tests, until these tests have been executed. Furthermore, occlusion tests cannot be applied for primitives that modify their depth values in the pixel shader or update the stencil buffer when the depth test fails.

3.4 Occlusion Unit Implementation Details

Most consumer graphics hardware include an occlusion culling unit but no exact details have been published. Therefore we briefly describe the implementation used in our tests to make the tests reproducible. Our implementation (Figure 3) differs somewhat from existing ones, as delayed culling requires that occlusion information is generated before the delay. However, the concept of improving occlusion culling by using a delay stream is not limited to any specific hardware implementation.

Low-resolution Z-buffer Our LRZ-buffer uses 8x8 pixel tiles. Each tile requires 32 bits of video memory as the minimum and maximum depth values are stored as 16-bit[1] floating point numbers. Both of the values are needed to allow changing the depth comparison mode during the frame. The LRZ-buffer is stored in video memory and accessed through a single-ported 32KB on-chip cache, split into eight banks to facilitate multiple simultaneous accesses. Paging to video memory is performed with a granularity of 256 bytes. Each depth buffer has its own separate LRZ-buffer.

Tile cache We also maintain an on-chip 16-way set associative *tile cache* that contains per-pixel depth values for 192 recently accessed tiles. All occlusion writes are performed into these tiles. The cached tiles are not paged into video memory. Instead, when a tile needs to be replaced, its minimum and maximum depth values are scanned and applied to the LRZ-buffer. Although collapsing the tiles causes some occlusion loss, our tests indicate that in over 95% of the cases a tile to be replaced is fully covered, i.e., the far plane is not visible. Replacing these entries causes a negligible increase in conservativity. If the cache set has only partially covered tiles, the spatially (XY) most distant one of the 16 tiles is replaced.

Occlusion tests The causal occlusion test after the vertex shader is executed in two parts. If the input block is visible according to the LRZ-buffer, a more accurate per-pixel test is made by using the tile cache while the occlusion write is performed. The delayed occlusion test consults only the LRZ-buffer.

Optimizations The culling efficiency of the LRZ-buffer is further improved by augmenting its depth information by the per-pixel values of the actual depth buffer. This provides additional coverage from triangles that could not be initially written into the LRZ-buffer such as alpha matte objects. This feature is used in the processing of order-independent transparency (Section 4). When supersampling is used, we employ a depth estimation buffer [Zhang et al. 1997] for storing the sub-pixel depth values compactly. In this case tiles are allocated from the tile cache three at a time: one for storing the per-pixel minimum depth values and one for the maximum values. The third tile is used for storing a 16-bit coverage mask for each pixel. The depth values are considered valid only if the corresponding coverage mask is full.

3.5 Results

We have implemented cycle-accurate VHDL models of the delay stream compression and decompression units and the occlusion test units. In total they consume 56KB of SRAM and 300K gates, which results roughly in a 4% size increase in modern graphics chips.

We have tested the performance of delayed occlusion culling using several industry-standard benchmark applications and test scenes (Table 1). The delay stream size was set to 2MB, which was able to hold 33K–80K triangles. Transparent surfaces were

[1] Our Z_{16} satisfies $Z_{16} \leq Z_{32} < (Z_{16} + 1\ unit\ in\ the\ last\ place)$.

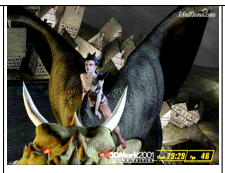
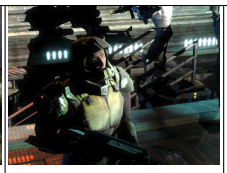
										
	Car Chase		**Dragothic**		**Alpha Squadron**		**VillageMark**		**PowerPlant**	
Culling method	*causal*	*delayed*	*causal*	*delayed*	*causal*	*delayed*	*causal*	*delayed*	*causal*	*delayed*
Triangles in view frustum	82K		38K		36K		1K		420K	
Depth complexity (DC)	2.9		3.5		3.1		6.0		14.2	
DC after occlusion cull	2.21	1.26	2.62	1.25	2.38	1.34	4.20	1.18	5.24	1.31
Ratio in pixel processing	1.8:1		2.1:1		1.8:1		3.6:1		4.0:1	
Z-buffer bandwidth	20.3	11.8	24.8	11.6	22.2	12.4	41.5	11.5	47.0	12.2
Frame buffer bandwidth	9.1	5.4	11.6	5.4	9.6	5.7	20.3	5.6	21.1	5.6
Texture bandwidth	18.0	10.0	21.2	10.0	19.0	10.9	42.2	11.8	0	0
Delay stream bandwidth	-	3.1	-	1.2	-	0.42	-	0.05	-	1.0
Compressed triangle size	-	63B	-	51B	-	63B	-	39B	-	25B
Total bandwidth	47.3	30.3	57.5	28.2	51.5	29.4	104.0	23.4	68.1	18.8
Ratio in bandwidth	1.6:1		2.0:1		1.8:1		4.4:1		3.6:1	

Table 1: The resulting average depth complexity after occlusion culling with causal and delayed architectures and the measured bandwidth (MB/frame) requirements at 1280x1024x32bpp without supersampling and with 32bpp depth. Texture bandwidth was estimated according to an 80% texture cache hit rate. The initial Z-buffer clear was excluded from the measurements. *Car Chase* and *Dragothic* from 3DMark2001 SE and *Alpha Squadron* from 3DMark2003, courtesy of FutureMark Corporation. *VillageMark* courtesy of PowerVR, a division of Imagination Technologies. *PowerPlant* courtesy of the Walkthrough Group at the University of North Carolina at Chapel Hill (www.cs.unc.edu/~walk).

excluded from all scenes in order to make the optimal depth complexity exactly 1.0. The 3DMark scenes have a relatively low depth complexity compared to games currently in development. To find out how the algorithm scales in presence of more overdraw, we included two scenes with a higher depth complexity. To validate that the average size of triangles does not affect the culling rate, we also tested the same environments with tessellation applied to all triangles. Even when 16 times more triangles were rendered per frame, the resulting depth complexity remained unchanged.

Compared to causal occlusion culling, the number of pixels that require execution of a pixel shader was reduced on average by a factor of three. The delayed occlusion culling overestimated the number of visible pixels only by 18–34% in all of the test scenes. Only the compressed output of the causal occlusion culling was written into the delay stream, consuming 0.1–3.1MB of video memory bandwidth per frame. The total video memory bandwidth was reduced two to four times. Assuming the overall performance was not bounded by geometry processing, savings in pixel processing and memory bandwidth should almost directly result in a proportionally increased frame rate. Consequently, the *Car Chase* and *Dragothic* benchmarks from 3DMark2001 SE should render almost twice as fast and the *VillageMark* test almost four times faster.

4 Order-Independent Transparency

The fundamental nature of order-independent transparency is that all visible data must be collected before the processing can begin. This is exactly what a sufficiently long delay stream does. In this section we show how the hardware units used for delay stream management can be used for reducing the memory and bandwidth requirements of order-independent transparency (OIT) in many cases.

Not all transparent surfaces need special treatment. For example, head-up displays and alpha matte objects can easily be rendered correctly by using the standard Z-buffer. Therefore the use of an OIT solver should be controllable by render states.

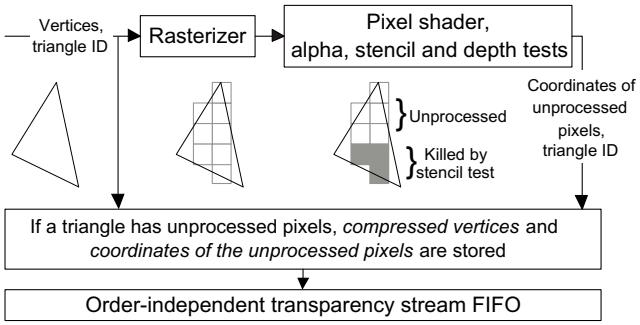


Figure 4: Storing information to the order-independent transparency stream. Only the triangles having unprocessed pixels after the pixel shader and per-pixel tests are stored into the stream.

4.1 Transparency Stream Construction

After the second occlusion test the triangles requiring OIT are collected into the OIT stream FIFO (Figure 1). The main difference compared to the R-buffer [Wittenbrink 2001] is that R-buffer stores shaded fragments (ARGBZ, blend mode, coordinates), whereas we store all render state changes as well as compressed vertices and coordinates of unprocessed pixels (Figure 4). We exploit image-space coherence and store coordinates only for 8x8 pixel chunks. In addition to the chunk coordinates, a bit mask indicating the coordinates of the remaining pixels inside the chunk is stored.

Color, depth and other attributes are computed from the vertices and render states during peeling by using the existing hardware units that would otherwise be idle. This effectively implements deferred shading for visible transparent surfaces. Few hidden primitives are stored due to the efficient occlusion culling.

In order to preserve the established rendering semantics, all per-pixel operations that modify or depend on the stencil buffer contents must be executed in the correct order. We propose duplicating the pixel shader programs in the device driver. The first program is

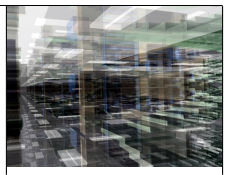
	 Venus	 Naked Empire		
Triangles in frustum	90K		123K	
Depth complexity	6.4		10.2	
Z-buffer bandwidth	50		331	
Color bandwidth	9.6		90	
OIT storage type	*Fragments*	*Geometry*	*Fragments*	*Geometry*
OIT bandwidth	127	36	1800	172
OIT size	14.9	2.1	137	8.4

Table 2: The required storage (MB) and bandwidth (MB/frame) of the processing of order-independent transparency. The results are shown for both the R-buffer fragment stream and our geometry stream. The resolution was 1280x1024x32bpp with 32bpp depth without supersampling. Our variant used roughly an order of magnitude less memory. *Venus de Milo* courtesy of Viewpoint Digital was rendered using surface splatting. All primitives are transparent in surface splatting. *Naked Empire* courtesy of Ned Greene and Gavin Miller, Apple Computer.

executed before the OIT stream and the second one computes depth and color during depth peeling. Dead code elimination [Muchnick 1997] is performed so that the first shader has only the instructions that can affect the stencil and the second one has all that cannot.

The potentially unbounded number of visible transparent surfaces is a fundamental issue in order-independent transparency. The size of the OIT stream is configurable but an overflow is possible. In that case we follow the approach used in R-buffer and page the excess surfaces into system memory.

4.2 Transparency Stream Peeling

The depth peeling procedure using one frame buffer and one depth buffer proceeds as follows:

1. Scan the OIT stream and mark all hidden pixels as processed. Remove fully processed triangles from the stream.
2. At this point, the remaining surfaces are visible. Peel (a-b) until the OIT stream is empty:
   (a) Clear the depth buffer, scan the OIT stream and store the most distant depth value for each pixel.
   (b) Scan the OIT stream again and for each pixel blend the surface with a depth value equaling the stored depth value. Mark processed pixels and remove fully processed triangles.

The number of passes over the OIT stream can be reduced by using multiple depth buffers simultaneously [Wittenbrink 2001].

The OIT stream contains render state changes that may have become obsolete due to the peeling process, e.g., setting the alpha blend mode twice without triangles in between. Due to the compact size of render state changes it may not be necessary to remove the obsolete ones. If needed, these occurences could be marked to another stream, which has one validity bit for each currently stored render state change. Obsolete changes would thus be removed from the OIT stream during the next peeling pass.

4.3 Results

We have measured the memory consumption and required bandwidth by using both our approach and a variant that stores fragments. Both of the approaches create the same final image. Algorithms that use fragment storage include A-buffer [Carpenter 1984] and R-buffer [Wittenbrink 2001]. Depth peeling was performed using one frame buffer and one depth buffer.

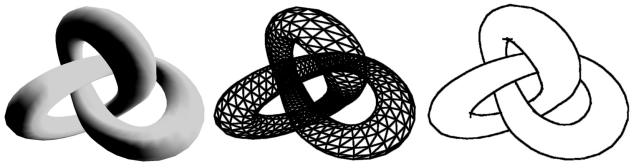


Figure 5: The picture on the left shows a rendering of the torusknot model. The center image is a back-face culled wireframe view of the model and the one on the right shows the discontinuity edges after hidden surface removal. Antialiasing pixels intersected by discontinuity edges is significantly less expensive than antialiasing all edge pixels. For highly tessellated models the difference is even more pronounced.

Because the Z-buffer cannot handle order-independent transparency, no industry-standard test scenes exist. We tested the approaches with two very different models (Table 2). A transparent version of a part of the Naked Empire consists of relatively large triangles and has an average depth complexity exceeding ten. The Venus model was rendered with the surface splatting algorithm of Zwicker *et al.* [2001] using 45K splats. The average depth complexity of drawn pixels is 6.4, a figure typical for surface splatting. The Z-buffer bandwidth is smaller than one might expect due to hierarchical rejection provided by the occlusion test unit.

In our test scenes the geometry stream uses considerably less memory than the fragment stream. With the Naked Empire, the ratio is 16.3:1 and with the Venus model 7.0:1. Storing geometry instead of fragments is appealing as higher screen resolutions and antialiasing further widen the gap in storage requirements. Optimally, the two approaches could be combined by converting the triangles into shaded fragments when they have only a few unprocessed pixels left. Further reductions could be achieved by compressing also the fragment data.

5 Discontinuity Edges

Concentrating supersampling efforts only to the discontinuity edges of objects has several benefits. First of all, the number of silhouette edges in many models is only $O(\sqrt{N})$ of the total edge count [Sander et al. 2001]. The number of pixels intersected by silhouette edges stays almost constant when an object is tessellated and the relative frequency of discontinuity edge pixels decreases when screen resolutions grow (Figure 5).

Supersampling only the discontinuity edges produces results with a good image quality at a reasonably low cost. The approach has also limitations: shader undersampling, geometric aliasing and interpenetrating surfaces are not handled properly. All kinds of edge supersampling may emphasize the edges under certain circumstances such as low-frequency lighting. However, the use of texture-mapping often masks these artifacts.

5.1 Detecting Discontinuity Edges

In this section we describe an algorithm for detecting discontinuity edges in hardware. The implementation uses only logical operations and on-chip memory. The detection is performed after the vertex shader and can thus support arbitrarily deformed meshes. Also, the working set is limited to the triangles surviving the clipping and culling stages. This detection could not be done without using a delay of some kind.

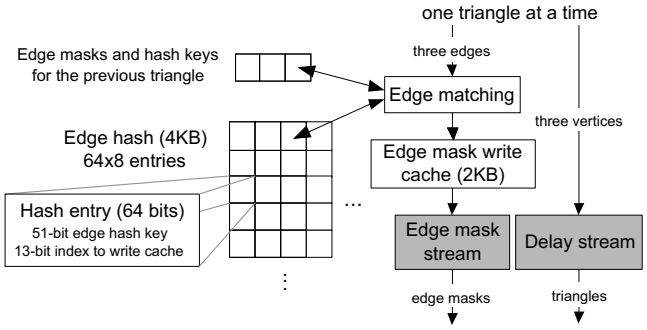


Figure 6: The edges of an input triangle are classified as shared or discontinuity edges using geometric hashing. If a matching edge is found from the edge hash table, the edge is marked to be shared by both of the triangles.

Our approach is to conservatively assume that all triangle edges are discontinuity edges and to use a simple geometric hashing algorithm for detecting shared non-sharp edges (Figure 6).

Each triangle in the delay stream has an associated six-bit edge mask that classifies its edges either as discontinuity or shared edges. For the latter case we further indicate whether this is the first or the second occurence of an edge; we call these *opening* and *closing* edges respectively. The edge masks of the previous 2730 triangles are stored in a circular 2KB on-chip *edge mask write cache*. Hence, the final classification of edges is delayed by almost 3K triangles, which is possible since the triangles are retained in the delay stream. For most models this delay is enough to detect all shared edges.

When a triangle is stored into the delay stream, its edges are tested by using an *edge hash* table. If a matching shared edge is found from the hash table, the cached edge masks of the two triangles are updated and the entry is removed from the hash. Otherwise, a new entry for the opening edge is placed into the hash.

The implementation of the edge hash is an eight-way set associative LRU cache with 64 sets. The hash keys are 64-bit values, where 13 bits are used for indexing the triangle that *opened* the edge. The edge hash keys use 57 bits but the lowest six bits are used for addressing the edge hash and are thus not stored explicitly. The edge hash key is formed from a bit indicating whether the triangle is front- or back-facing and two vertex hash keys constructed by combining the bits of all active vertex attributes produced by the vertex shader, e.g., positions, normals and texture coordinates.

To further reduce the conflict misses the three edge hash keys of the previous triangle are stored separately. If the input triangles are ordered coherently as triangle strips or fans, two consecutive triangles are likely to share an edge. By processing these edges prior to the hash lookup it is often possible to reduce the number of entries placed into the hash by 20–33%. Since a material boundary is always a discontinuity edge, the edge hash is cleared whenever the rendering state changes.

Due to the finite size of the caches and the use of hash values for matching the edges, the algorithm may occasionally misclassify an edge. A discontinuity edge is missed only if two unrelated edges generate the same hash key. This has not occurred in our tests as the probability [2] is negligible in practice. Classifying an edge incorrectly to be a discontinuity edge is more common and happens due to a conflict or capacity miss in the hash. In the test scenes this misclassification increases the number of detected discontinuity edges by less than 1%. The probability can be reduced by increasing the capacities of the edge hash and the edge mask write cache. Another option is to use additional connectivity information provided by the tessellator or the application, e.g., OpenGL edge flags.

[2] $P(same\ key) = 1 - (1 - 2^{-57})^{8*64} \approx 2.81 * 10^{-14}$ per edge.

	 **Beethoven**		 **Venus**	
Triangles in frustum	21016		5668	
Antialiasing type	*E*	*DE*	*E*	*DE*
Memory use increase	57.8%	10.0%	24.6%	2.7%
Bandwidth increase	251.2%	74.0%	156.2%	49.9%
Pixels antialiased	17.4%	2.1%	8.0%	0.3%
Pixel Ratio	8.3:1		25.9:1	

Table 3: A comparison between our antialiasing algorithm when supersampling is performed at the edges of triangles (E) and only at the discontinuity edges (DE). The models were rendered in 1024x1024 resolution and 16x16 sparse supersampling was used for the antialiased pixels. The 32-bit frame and depth buffers require 4MB each when no supersampling is used. Both models courtesy of Viewpoint Digital.

5.2 Application to Adaptive Supersampling

The edge classification produced by the discontinuity edge detection can be used by various antialiasing algorithms. For example, the visible pixels intersected by the discontinuity edges could be tagged and subsequently filtered in a post-processing pass. Our approach is to increase the sampling rate only for the pixels intersected by discontinuity edges, as shown in Figure 7. For all other pixels the color and depth are computed once per intersecting triangle. Having geometric information about shared edges replaces commonly used heuristics of coverage, color, depth and object identifiers with connectivity information.

To determine the pixels covered by connected triangles having shared edges, a 4-bit *pixel edge counter* is maintained for each pixel intersected by an edge. As explained in Section 5.1, shared edges have been classified into opening and closing edges. Opening edges intersecting a pixel increment the counter by one and closing edges decrement it. The counter is sticky, i.e., once it reaches its maximum value of 15, its value cannot be modified. Visible discontinuity edges intersecting a pixel always set the counter to the maximum. When the counter reaches zero it is *known* that the pixel is covered by a continuous surface. Then the pixel is collapsed and the subsample at the center of the pixel is written into the frame buffer. Depth and stencil buffers are updated similarly. Edge pixels are also collapsed if a non-edge pixel completely covers all of the samples.

Storing the edge counter and 16 samples for each pixel would require enormous amounts of memory and bandwidth. Therefore we use an *edge pixel cache* for storing the recently-accessed edge pixels. In higher screen resolutions almost all edge pixels are intersected only by two triangles. Such pixels can be represented compactly using lossless block truncation coding (BTC) [Delp and Mitchell 1979]: each pixel requires only two samples and a 16-bit mask for indicating which samples are covered by which triangle. Our 16KB edge pixel cache can hold 512 BTC-encoded pixels and 32 pixels for which all 16 samples are stored. When a cache entry needs to be replaced, fully covered and collapsed pixels are written directly to the frame buffer whereas pixels with non-zero edge counters are paged into video memory. At the end of the frame the contents of the cache and the supersampled pixels in the video memory are filtered and stored into the frame buffer.

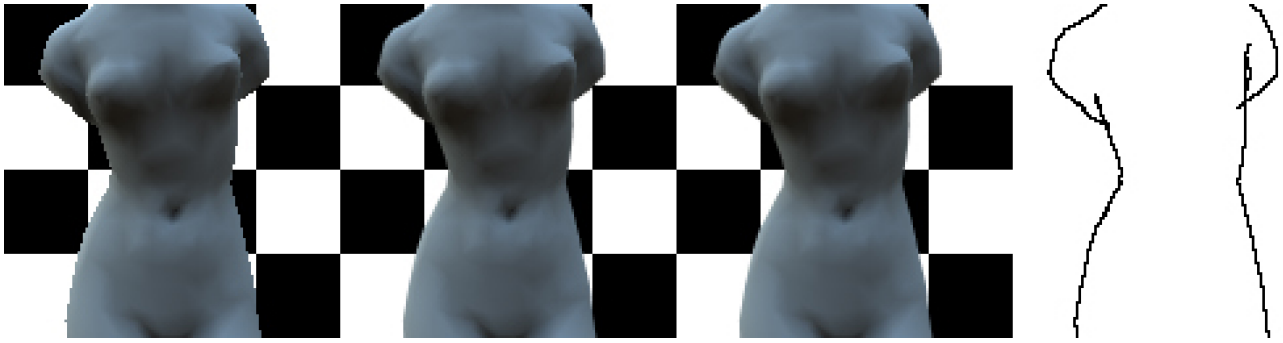


Figure 7: From left to right: a) Closeup of the Venus model rendered in 128x128 resolution with one sample per pixel b) the image rendered with 16 samples per pixel c) same image where 16 samples have been used only for the discontinuity edge pixels and d) pixels intersected by discontinuity edges.

5.3 Results

The relative performance of antialiasing all edges and only the discontinuity edges is illustrated in Table 3. In our test scenes the ratio between the number of all edges and discontinuity edges varied between 7.1:1 and 30.1:1. In 1024x1024 screen resolution the percentage of pixels intersected by discontinuity edges and thus antialiased was 0.3–2.1%. Although these pixels required 3–16 times more memory than the single-sampled ones, there were relatively few of them. As a consequence, the amount of video memory used was increased by less than 10% compared to no antialiasing. The memory bandwidth requirements were increased by 50–75%.

6 Conclusions and Future Work

We have shown that delay streams can be used for improving the performance of occlusion culling hardware and edge antialiasing. Load-balancing between geometry and pixel units has been enhanced as a side-product. We have also demonstrated how the memory and bandwidth requirements for order-independent transparency can be substantially reduced in many cases.

As the use of delay streams allows hardware implementation of semi-global algorithms commonly used in software rendering systems, we believe that they will have many more potential applications. An interesting possibility would be using the discontinuity edge detection algorithm for generating optimized shadow volumes in hardware. Moreover the silhouette edges could be made available to pixel shader programs for non-photorealistic rendering styles.

Performing occlusion culling based on the bounding volumes of input objects would reduce the workload of the tessellator and vertex shader units. Although OpenGL and DirectX support user-made occlusion queries, they do not currently provide a method for specifying bounding volumes for an automatic culling of objects. This has long been a feature of RenderMan [Upstill 1990].

Both the R-buffer and our approach for sorting transparent pixels place a heavy burden on the depth buffer bandwidth. The situation could be improved by using tiled rendering for the OIT passes [Fuchs et al. 1989; Upstill 1990; Molnar et al. 1992].

Acknowledgments We would like to express our gratitude for the support provided by the R&D teams of Hybrid Graphics and Bitboys. We would also like to thank the anonymous reviewers and Janne Kontkanen, Jaakko Lehtinen, Kari Pulli, Jussi Räsänen and Lauri Savioja for their valuable feedback.

References

3DLABS, 2002. Wildcat: SuperScene antialiasing white paper. http://www.-3dlabs.com/product/technology/superscene_antialiasing.htm.

AKELEY, K. 1993. RealityEngine graphics. In *Proceedings of ACM SIGGRAPH 93*, ACM Press, 109–116.

AKENINE-MÖLLER, T., AND HAINES, E. 2002. *Real-Time Rendering, 2nd edition*. A.K. Peters Ltd.

APPEL, A. 1968. Some techniques for shading machine renderings of solids. In *AFIPS Conference Proceedings*, vol. 32, 37–45.

BARTZ, D., MEISSNER, M., AND HÜTTNER, T. 1998. Extending graphics hardware for occlusion queries in OpenGL. In *Proceedings of the 1998 EUROGRAPHICS/SIGGRAPH workshop on Graphics hardware*, 97–104.

CARPENTER, L. 1984. The A-buffer, an antialiased hidden surface method. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, ACM, vol. 18, 103–108.

CATMULL, E. 1974. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah.

CHAUVIN, J. C. 1994. An advanced Z-buffer technology. In *Proceedings of the Image VII Conference*, 77–85.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, ACM, vol. 18, 137–145.

CROW, F. C. 1977. The aliasing problem in computer-generated shaded images. *Communications of the ACM 20*, 11, 799–805.

DEERING, M., AND NAEGLE, D. 2002. The SAGE graphics architecture. *ACM Transactions on Graphics 21*, 3, 683–692.

DEERING, M., WINNER, S., SCHEDIWY, B., DUFFY, C., AND HUNT, N. 1988. The triangle processor and normal vector shader: a VLSI system for high performance graphics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 88)*, ACM, vol. 22, 21–30.

DELP, E., AND MITCHELL, O. 1979. Image coding using block truncation coding. *IEEE Transactions on Communications 27*, 9 (September), 1335–1342.

DIRECTX, 2002. Microsoft DirectX SDK Documentation. http://www.microsoft.com/directx.

DURAND, F. 1999. *3D Visibility: Analytical Study and Applications*. PhD thesis, Université Grenoble I - Joseph Fourier Sciences et Géographie.

EVERITT, C. 2001. Interactive order-independent transparency. http://www.developer.nvidia.com.

FUCHS, H., POULTON, J., EYLES, J., GREER, T., GOLDFEATHER, J., ELLSWORTH, D., MOLNAR, S., TURK, G., TEBBS, B., AND ISRAEL, L. 1989. Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Computer Graphics (Proceedings of ACM SIGGRAPH 89)*, ACM, vol. 23, 79–88.

GREENE, N., AND KASS, M. 1993. Hierarchical Z-buffer visibility. In *Proceedings of ACM SIGGRAPH 93*, ACM Press, 231–240.

HAEBERLI, P., AND AKELEY, K. 1990. The accumulation buffer: hardware support for high-quality rendering. In *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, ACM, vol. 24, 309–318.

HILLESLAND, K., SALOMON, B., LASTRA, A., AND MANOCHA, D. 2002. Fast and simple occlusion culling using hardware-based depth queries. Tech. Rep. TR02-039, UNC Chapel Hill.

HOPPE, H. 1999. Optimization of mesh locality for transparent vertex caching. In *Proceedings of ACM SIGGRAPH*, ACM Press, 269–276.

JOUPPI, N., AND CHANG, C.-F. 1999. Z3: an economical hardware technique for high-quality antialiasing and transparency. In *Proceedings of the 1999 Eurographics/SIGGRAPH workshop on Graphics hardware*, ACM Press, 85–93.

KLOSOWSKI, J. T., AND SILVA, C. T. 2001. Efficient conservative visibility culling using the priorized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics 7*, 4 (October-December), 365–379.

LAU, R. W. H. 1995. An adaptive supersampling method. In *International Computer Science Conference (ICSC)*, 205–214.

LEE, J.-A., AND KIM, L.-S. 2000. Single-pass full-screen hardware accelerated antialiasing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, 67–75.

MAMMEN, A. 1989. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications 9*, 4 (July), 43–55.

MARK, W., AND PROUDFOOT, K. 2001. The F-buffer: a rasterization-order FIFO buffer for multi-pass rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on on Graphics hardware*, ACM Press, 57–64.

MATROX, 2002. 16x Fragment Antialiasing. http://www.matrox.com/mga/-products/ tech_info/pdfs/parhelia/faa_16x.pdf.

MEISSNER, M., BARTZ, D., GÜNTHER, R., AND STRASSER, W. 2001. Visibility driven rasterization. *Computer Graphics Forum 20*, 4, 283–294.

MOLNAR, S., EYLES, J., AND POULTON, J. 1992. PixelFlow: high-speed rendering using image composition. In *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, ACM, vol. 26, 231–240.

MOREIN, S., 2000. ATI Radeon - HyperZ Technology. ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, Hot3D session.

MUCHNICK, S. S. 1997. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc.

OPENGL ARCHITECTURE REVIEW BOARD, D. S. 1999. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2*. Addison-Wesley.

PEERCY, M., AIREY, J., AND CABRAL, B. 1997. Efficient bump mapping hardware. In *Proceedings of ACM SIGGRAPH 97*, ACM Press, 303–306.

POWERVR, 2000. PowerVR white paper: 3D graphical processing. http://www.powervr.com/pdf/TBR3D.pdf.

SANDER, P., HOPPE, H., SNYDER, J., AND GORTLER, S. 2001. Discontinuity edge overdraw. In *2001 ACM Symposium on Interactive 3D Graphics*, ACM Press, 167–174.

SAUER, F., MASCLEF, O., ROBERT, Y., AND DELTOUR, P. 1999. Outcast: Programming towards a design aesthetic. In *Proceedings of Game Developers Conference*, 811–827.

SCHILLING, A. G., AND STRASSER, W. 1993. EXACT: Algorithm and hardware architecture for an improved A-buffer. In *Proceedings of ACM SIGGRAPH 93*, ACM Press, 85–92.

SCHMITTLER, J., WALD, I., AND SLUSALLEK, P. 2002. SaarCOR: A hardware achitecture for ray tracing. In *Proceedings of the conference on Graphics hardware 2002*, ACM Press, 27–36.

TORBORG, J., AND KAJIYA, J. T. 1996. Talisman: commodity realtime 3D graphics for the PC. In *Proceedings of ACM SIGGRAPH 96*, ACM Press, 353–363.

UPSTILL, S. 1990. *The Renderman Companion*. Addison Wesley.

WESTOVER, L. 1990. Footprint evaluation for volume rendering. In *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, ACM, vol. 24, 367–376.

WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*, 6, 343–349.

WILLIAMS, L. 1983. Pyramidal parametrics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 83)*, ACM, vol. 17, 1–11.

WINNER, S., KELLEY, M., PEASE, B., RIVARD, B., AND YEN, A. 1997. Hardware accelerated rendering of antialiasing using a modified A-buffer algorithm. In *Proceedings of ACM SIGGRAPH 97*, ACM Press, 307–316.

WITTENBRINK, C. M. 2001. R-buffer: a pointerless A-buffer hardware architecture. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, ACM Press, 73–80.

XIE, F., AND SHANTZ, M. 1999. Adaptive hierarchical visibility in a tiled architecture. In *Proceedings of the 1999 Eurographics/SIGGRAPH workshop on Graphics hardware*, 75–84.

ZHANG, H., MANOCHA, D., HUDSON, T., AND HOFF, K. E. 1997. Visibility culling using hierarchical occlusion maps. In *Proceedings of ACM SIGGRAPH 97*, ACM Press, 77–88.

ZWICKER, M., PFISTER, H., VAN BAAR, J., AND GROSS, M. 2001. Surface splatting. In *Proceedings of ACM SIGGRAPH 2001*, ACM Press, 371–378.