Actuality
Systems

Public Information Disclosure

Actuality Ref:	ASI-0110
Title	TECHNICAL INTRODUCTION TO THE BASIC SPATIALGL RENDERING PIPELINE FOR A PERSPECTA 1.9/2.0
Publication Date	13 November 2005
Inventors	Won-Suk Chun (Cambridge, MA), Thomas J. Purtell II (Cambridge, MA), Oliver S. Cossairt (Cambridge, MA)
Point of Contact	Gregg Favalora, CTO favalora [at] actuality-systems.com +1 (781) 541-6016 x102
Notes	Patent Pending

Technical Introduction to the Basic SpatialGL Rendering Pipeline for Perspecta 1.9/2.0

Reference: See also “Volumetric Core Technologies: Display-Independent ‘Spatial Platform’, High-Speed Rendering Electronics, and Several New Displays,” Provisional application filed July 8, 2004. Atty ref: ACS-0006 which is incorporated herein by reference. See Attachment 2, page 4, of above document for useful context for this document.

Petition to Make Special: This invention has applications relating to countering terrorism and curing HIV / AIDS.

Government Rights: This invention was made with United States Government support under Contract No. xxxxxx-xx-x-xxxx awarded by the xxxxxx program. The United States Government has certain rights in the invention.

All documents referred to herein are expressly incorporated herein by reference.

This disclosure pertains to graphical software for display systems, including 3-D displays. An overview of 3-D displays follows:

There are many types of 3D displays, including stereoscopic displays, multiplanar volumetric displays (for example U.S. 6,554,430, “Volumetric three-dimensional display system”) holographic video systems (for example U.S. 5,172,251, “Three dimensional display system”), and multi-view 3D displays. Applications of 3D displays may include the depiction of medical images, such as a transparent CT image of a patient’s anatomy which includes vasculature and tumors; geophysical data for the petroleum industry, such as seismic data overlaid with drill paths; or 3D luggage scan data, for example the CT scan of luggage in which each 3D pixel (“voxel”) is color coded as a function of effective atomic number.

Many implementations are possible. For example, in some cases, a user will use a 3D display by providing input to a host PC which is in communication with the 3D display. In other cases, the user will interact directly with the 3D display, for example if a video game computer is embedded in the 3D display and joysticks are mounted on the display’s enclosure. In another case, the user will simply look at the 3D display, which generates imagery on its own or relays data provided by an outside source.

The display front-end can be any of:

- A volumetric display, as taught in

    - M. Hirsch, “Three dimensional display apparatus,” U.S. 2,967,905 (1961)
    - R.D. Ketchpel, “Three-Dimensional Cathode Ray Tube,” U.S. 3,140,415 (1964).
    - R.K. Dorval, et al, “Volumetric three-dimensional display system,” U.S. 6,554,430 (2003).
    - G.E. Favalora, R.K. Dorval, D.M. Hall, M.G. Giovinco, J. Napoli, “Volumetric three-dimensional display system with rasterization hardware,” in *Stereoscopic Displays and Virtual Reality Systems VIII,* Andrew J. Woods, Mark T. Bolas, John O. Merritt, Stephen A. Benton, Editors, Proceedings of SPIE Vol. 4297, pp. 227-235 (2001).
    - A. Sullivan, “Multi-planar volumetric display system and method of operation,” U.S. 6,100,862 (2000).
- A holographic display, as taught in
    - S.A. Benton and J.S. Kollin, “Three dimensional display system,” U.S. 5,172,251 (1992).
    - D. Payne and C.W. Slinger, “Holographic displays,” U.S. 6,753,990 (2004).
- A multi-view, lenticular, or panoramagram display as taught in:
    - F.E. Ives, “Parallax stereogram and method of making same,” U.S. 725,567 (1903).
    - R.B. Collender, “Three dimensional unaided viewing method and apparatus,” U.S. 3,178,720 (1965).
    - J.B. Eichenlaub, “Autostereoscopic display with illuminating lines, light valve, and mask,” U.S. 4,829,365 (1989).
- A stereo display, as any well-known goggle-based or other system
    - I.E. Sutherland, “A head-mounted three dimensional display,” *Joint Computer Conference [Proceedings of the AFIPS],* Vol. 33, pp. 757-764 (1968).
    - L. Lipton, et al, “Stereoscopic Television System,” U.S. Pat. 4,523,226 (1985)

Introduction

A software framework for developing 3-D applications for spatial displays is desirable. Once such framework may be Actuality Systems’ SpatialGL API. The analogous problem of developing 3-D software for 2-D displays is well studied and solutions are widely deployed. A straightforward but naïve application of such a framework, while likely to be marginally effective, is unlikely to fully exploit the intrinsic advantages afforded by a true spatial display. In fact, each of several classes of spatial display possesses such distinguishing characteristics. Ultimately, the goal of SpatialGL is to provide a universal framework for all spatial displays (including conventional 2-D displays), including the SpatialGL pipeline implementation on Actuality Systems’ Perspecta display.

Still, the current state-of-the-art in traditional 3-D graphics on 2-D displays provides two foundations on which to design and implement SpatialGL. Even in light of substantive

differences between rendering to 2-D and spatial displays, there remains a conspicuous analogy between the two. At least conceptually, segments of the entire process can be adapted to particular implementations. Also, the considerable mass-market penetration of powerful and affordable 3-D graphics processing units (GPUs) is impossible to ignore. Even if GPUs cannot directly implement SpatialGL, they can be reprogrammed to perform certain parts of SpatialGL. Tying the implementation of SpatialGL to the performance of commodity GPUs, which double in performance every 6 to 9 months (compared to 18 months for CPUs), will be a tremendous advantage in the future.

This being said, first consider the canonical 3-D graphics pipeline.

The Canonical 3-D Graphics Pipeline

The canonical 3-D graphics pipeline converts descriptions of 3-D objects and scenes into 2-D images. Several different approaches are available (e.g. ray-tracing, voxel, surfel, Reyes), but, particularly for real-time applications, the most common approach by far is the triangle rendering pipeline (Figure 1: Canonical 3-D Graphics Pipeline). The two most ubiquitous real-time 3-D graphics APIs, Direct3D and OpenGL, both expose such a pipeline.

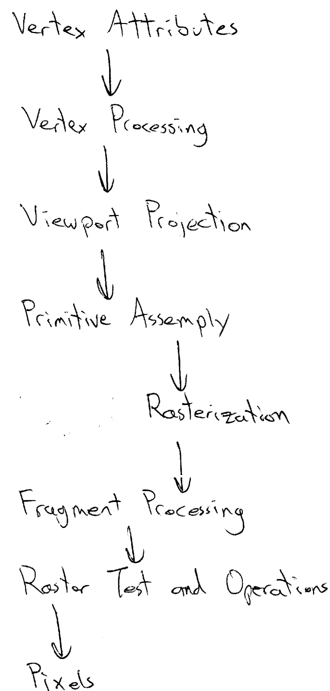


Figure 1: Canonical 3-D Graphics Pipeline

The triangle rendering pipeline has many advantages for real-time applications. Because it uses triangles as a rendering primitive, all major operations can be performed using simple linear equations. Other primitives, such as points and lines, can be rendered as triangles. Curved surfaces, can be approximated to an arbitrary degree of accuracy using

triangles. Each stage of the pipeline is itself highly parallel, encouraging extremely efficient hardware implementations in GPUs.

The goal of the triangle rendering pipeline is to synthesize 2-D digital images that a camera would capture in a virtual scene comprised of triangles. The basic operation is described as follows. An application passes geometry (described as vertices) to a GPU through an API (e.g. Direct3D or OpenGL). The GPU maps these vertices into a view volume, which is the region of space that can be seen by the virtual camera. The view volume is projected onto a 2-D surface that represents the view of the camera. This surface is sampled to synthesize the final 2-D digital image output.

Modern GPUs are now programmable. Instead of providing a small set of selectable rendering modes for each stage, certain stages expose an instruction set to allow client applications to request arbitrary computations. This provides the opportunity to tap a GPU as a general-purpose computation resource; essentially, the programmable stages are used as highly parallel floating-point processors. The inexpensive, mass-market nature of video cards, as well as their extremely steep performance growth, makes GPUs an attractive target for non-graphical computation (ref: www.gpgpu.org), or non-traditional graphics algorithms such as SpatialGL. Actuality Systems is developing SpatialGL.

SpatialGL Graphics Pipeline

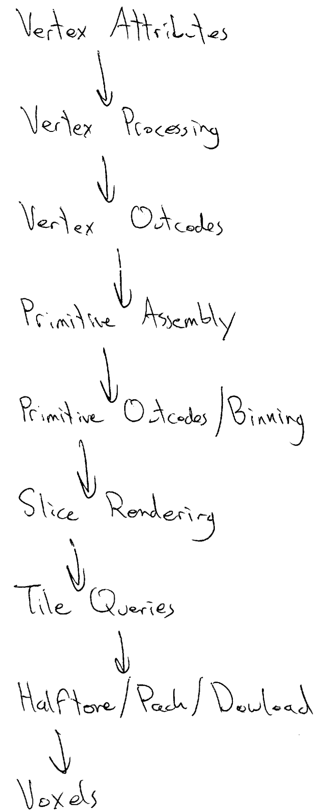
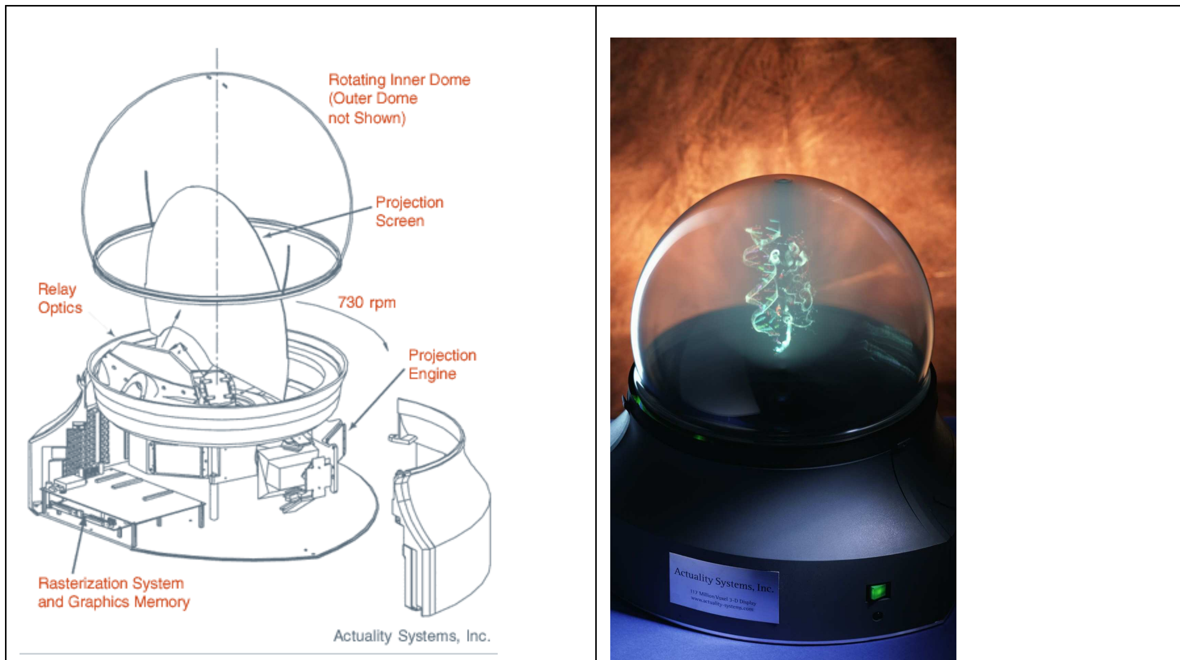


Figure 2: SpatialGL Graphics Pipeline

An exemplary embodiment of the SpatialGL graphics pipeline (Figure 2: SpatialGL Graphics Pipeline) is strongly modeled after the canonical 3-D graphics pipeline. Conceptually, they are structured the same way. However, unlike the canonical 3-D graphics pipeline, the SpatialGL graphics pipeline does not project the entire virtual scene onto a 2-D image surface. Instead, it divides the scene into a sequence of slices.

The Perspecta display contains a spinning screen synchronized to a high-speed digital projector. Each revolution, Perspecta strobes 396 different image slices so rapidly that visual persistence causes an observer to perceive the images as a single image volume. The goal of the SpatialGL graphics pipeline is to compute the contents of each of these image slices to present a synthetic image volume to observers. The Perspecta display is described in R.K. Dorval, et al, “Volumetric three-dimensional display system,” U.S. 6,554,430 (2003).



Slice View Volume

Computing the contents of these image slices may be similar to what a GPU does. However, instead of rendering a single view per 2-D image, the GPU renders 396 slices per 3-D image on Perspecta. Because the screen is constantly spinning, and the high-speed digital projector strobes images for a brief duration, each slice actually occupies a volume of space. To compute the image for a particular slice, the scene is rendered using the canonical rendering pipeline while setting the view volume to be the occupied by the slice. Usually, setting a projection matrix transformation or writing a vertex shader program accomplishes this task. Unfortunately, the shape of the slice volume is not simple; instead of being a convex volume, it crosses itself along the axis of screen rotation (Figure 4: Slice Volume). Such a view volume cannot be specified by setting a projection matrix transformation or by writing a vertex shader program.

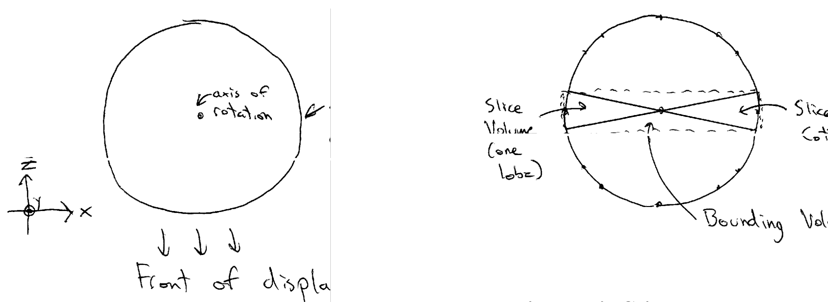


Figure 4: Slice Volume

Bounding Slice Volume

One potential approach is to make the best effort using only the standard projection matrix transformation. It may not be possible to specify the actual shape of the slice volume this way, but it is possible to specify a view volume that tightly bounds the slice volume. The bounding slice volume is simply a rectangular prism with near and far clip planes that are parallel to the slice plane, and whose left, right, top and bottom clipping planes frame the left, right, top and bottom boundaries of the slice image (Figure 4: Slice Volume) The result is a thin orthogonal view volume, which is easily expressed as a projection matrix transform.

The bounding view volume may include too much of the synthetic image, particularly near the axis of rotation. Thus, using bounding slice volume generates recognizable synthetic image volumes, but the parts of the image volume near the axis of rotation are blurry and may be too bright.

User Clip Planes

Most 3-D rendering APIs (including OpenGL and Direct3D) include clip planes. Clip planes constrain rendering to occur only in an arbitrary half-space. The clip plane partitions the view volume, and only rendering on one side of the clip plane is accepted. Multiple clip planes can be enabled, and only rendering accepted by all enabled clip planes is accepted. This means that multiple clip planes may only bound convex volumes.

However, the actual slice volume is concave. In order to use clip planes to bound this actual slice volume, the slice volume is decomposed into concave regions, and each concave region is rendered separately. Using this technique, slice rendering requires two passes, one for each concave “lobe” of the slice volume.

Shader Clip Planes

It is possible to program a GPU to perform clipping to the slice volume in a single pass by using vertex and fragment shader programs rather than user clip planes. Two intersecting planes bound the slice volume, but user clip planes do not combine correctly. Instead, a vertex shader can evaluate the clip plane equations, and a fragment shader can accept only the appropriate fragments.

A clip plane equation (a, b, c, d) partitions the view volume into a negative half-space and a positive half-space. A projective point (x, y, z, w) is in the negative half-space if $ax+by+cz+dw < 0$ and in the positive half-space otherwise. Because it is a linear expression, the plane equation can be evaluated per-vertex and interpolated during rasterization. The interpolated plane equation evaluations are passed to the fragment program.

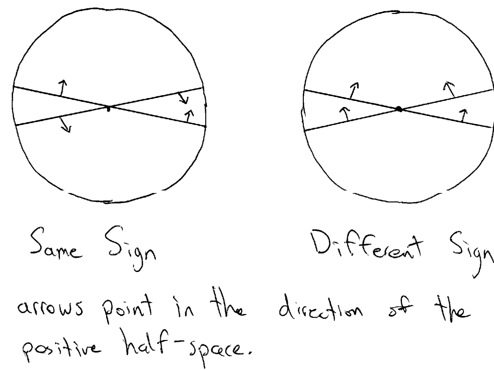


Figure 5: Slice Orientation

If the clip planes are oriented as shown on the left, then the fragment program should accept fragments where both clip plane signs are the same sign (their multiplicative product is positive). Alternatively, if the clip planes are oriented as shown on the right, then the fragment program should accept fragment where the clip plane signs are different (their multiplicative product is negative).

Rendering Dead Zone

The axis of rotation persists in creating problems whether user or shader clip planes are used. Because of numerical precision problems, clipping suppresses rendered near the axis of rotation. Because the distance between the clip planes is so short near the axis, it is unlikely that any fragments will be accepted in this region, creating a visible dead zone.

Plane Offset

One solution to eliminate the dead zone is to offset clip regions. Instead of setting the clip planes to tightly enclose the slice volume, bias the clip planes to enclose more space particularly near the axis of rotation. The clip planes can be tapered so that the clip region is tighter near the edge of the slice (Figure 6: Tapered Slice Volume), where the offset is not necessary.

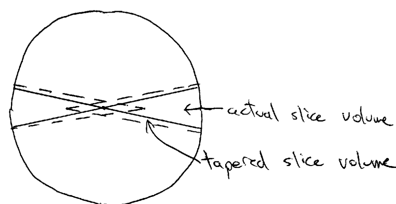


Figure 6: Tapered Slice Volume

If the clip planes are tapered, then each lobe of the slice volume requires its own clip planes. This means that clipping must be done in two passes.

Evaluation Bias

If the clip planes are parallel to the slice planes, then clipping can be done in a single pass. The plane equation evaluations for parallel planes differ only by a constant bias, so the plane equation evaluation can be reused in the fragment program with different biases. This allows clipping to occur in a single pass, even with plane offsets.

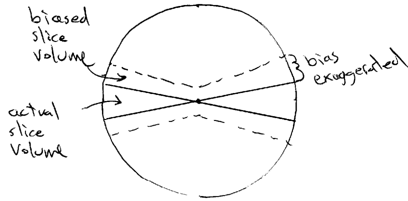


Figure 7: Biased Plane Equation Evaluation

Hyperbolic Clipping

There are several disadvantages to using clip planes parallel to slice planes. This technique uniformly dilates the clip volume, but the dilation is only necessary near the axis. Rendering to the slice either requires two passes or requires additional computation in the fragment program. One solution is to apply the bias after the plane equation evaluations are multiplied rather than before. The resulting clip region is a hyperbola (hyperbolic cylinder), where the clip planes are the asymptotes. The region near the axis is dilated, and bounds near the edges shrink asymptotically tighter.

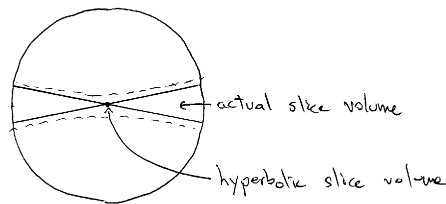


Figure 8: Hyperbolic Clipping Volume

Degenerate Sections

Some triangle orientations produce sections that are very thin. For example, any plane perpendicular to the axis of rotation would only intersect each slice in a line. However, because triangle rendering within each slice is ultimately performed as triangle rendering on the GPU, degenerate sections will not generate any fragments during GPU rasterization. The GPU will cull the lines as triangles with zero projected area during primitive assembly.

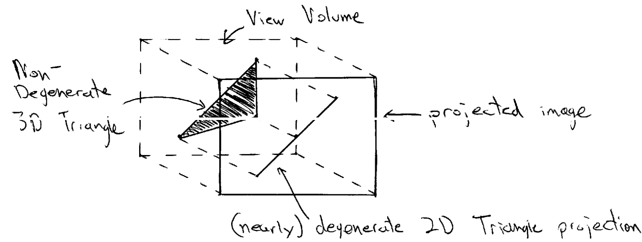


Figure 9: Sliver Triangle

The case does not improve substantially for nearly degenerate or sliver triangles (Figure 9: Sliver Triangle). The GPU rasterizer samples triangles with limited resolution, so nearly degenerate triangles cause aliasing that manifests as disconnected groups of fragments. The GPU implements rasterization rules that are robust enough to ensure that meshes of nearly degenerate triangles produce connected groups of fragments, but adjacent fragments may not necessarily be part of the adjacent triangles. This is an adequate guarantee for OpenGL, but it insufficient for SpatialGL.

In fact, it is extremely difficult to achieve analogous guarantees for triangle rasterization for a display whose geometry is as unusual as Perspecta. Ideally, SpatialGL prefers rasterization rules that guarantee continuous fragment generation even for degenerate primitives, but of course these rules are incompatible with the rasterization rules implemented on GPUs. An alternate is to reduce the likelihood of geometric aliasing artifacts during slice rendering.

One solution is to simply increase the sampling resolution of each slice to reduce the set of triangles that induce geometry aliasing. This can be done simply by increasing the render target resolution (supersampling) or slightly more efficiently by enabling multisample anti-aliasing. Unfortunately, this solution is very detrimental to performance; the GPU is already fillrate limited during SpatialGL rendering. More importantly, it can never properly handle all slivers or prevent the culling of truly degenerate sections.

Instead, another solution is to manipulate the projection of the primitives. The projected area of a triangle depends on the projection matrix transformation. The projection matrix should be modified to distribute degenerate triangle orientations so that no single orientation (e.g. perpendicular to the axis of rotation) reliably creates degenerate triangles, and that no individual triangle is degenerate over a wide range of slices.

It is simple to predict triangle degeneracy for a given projection transformation. Each triangle defines a plane; degenerate triangles define planes that contain the eye point (sliver triangles define planes that nearly contain the eye point). For an orthogonal projection, the eye point is infinitely distant, but in a particular direction; this is a well-defined point in projective geometry.

This observation indicates that an orthogonal projection is not a good choice for slice rendering; for a given slice, all triangles with a particular plane orientation will be

degenerate. A perspective projection is a better choice; in this case, triangle degeneracy depends on both plane orientation and position.

If each slice had a different projection transformation then it is unlikely that a triangle degenerate in one slice is also degenerate in adjacent slices. The implementation of SpatialGL does this by jittering the eye point location and randomly shearing the projective planes by subpixel distances. This technique also has the beneficial side effect of suppressing aliasing patterns that could arise from other sampling issues; such aliasing patterns will produce uncorrelated noise instead of Moire patterns. The jitter and shearing is randomized between slices, but each slice consistently uses the same projection matrix to maintain rendering invariance.

Image Reformatting

After the GPU computes a slice image, it must be downloaded from the GPU and uploaded to the digital projector. However, the most efficient image format for the GPU is 8-bits for four channels (red, green, blue and alpha for a total of 32-bits per pixel), but the digital projector may only support 1-bit per color channel (red, green and blue for a total of 3-bits per pixel). One solution is to simply transfer the 32-bit images directly to the projector. However, this amounts to over 90% waste in bandwidth because only 3-bits are necessary. This waste is unacceptable because bandwidth out of a GPU is typically too low.

A better solution is to halftone the image and pack as many 3-bit pixels into a single 32-bit pixel using the GPU. A fragment program binds the slice image as a texture and fetches 8 horizontally adjacent pixels from the slice image. The red, green, and blue color channels of each pixel are compared against a threshold from an ordered dither matrix, resulting in a 3-bit, halftoned color for each input pixel. Each 8-bit color channel of the output color contains the 8 halftoned bits for the corresponding color channel of each input pixel.

Theoretically, it is possible to pack 10 3-bit pixels in a single 32-bit pixel, but it is computationally more efficient to align the packing on color byte boundaries. The tradeoff is GPU computation for GPU bandwidth, and more complicated approaches simply shift the bottleneck to computation rather than communication.

SpatialGL Optimizations

GPUs are substantially more efficient than CPUs at 3-D rendering, but each SpatialGL frame still requires hundreds of slices to be rendered and hundreds of megabytes of image data to be processed and transmitted. Binning and tiling are two important techniques that may be utilized to improve the performance of the SpatialGL implementation on Perspecta.

Binning

Overall SpatialGL rendering performance can substantially improve if a small amount of CPU work can coarsely simplify the scene for each slice. This CPU operation is called binning. During binning, the CPU determines which primitives must be rendered in each slice. Binning is a conservative estimate; for each slice, the rendered set of primitives must contain all the primitives that would be actually visible in that slice. Essentially, binning is a scene acceleration algorithm (such as frustum culling) adapted for Perspecta display geometry.

Tuning a scene acceleration algorithm to find the sweet spot between CPU and GPU load is a difficult problem exacerbated by the disproportionate performance scaling between CPUs and GPUs. If an algorithm is tuned to a particular generation of CPU/GPU technology, that same algorithm will almost certainly become CPU bound for the next generation technologies. This phenomenon happens regularly and predictably in computer games. The situation is so severe that developers must undertake extreme measures to avoid overloading the CPU even for contemporary technologies; function call overhead to the rendering API alone can become a significant if not dominant performance factor (more so for Direct3D than OpenGL) in a 3-D application.

Thus, it is extremely important to be sensitive about the issue of CPU load, particularly because the CPU is almost always performing other tasks besides issuing function calls to the rendering API. In the case of Perspecta, the CPU is responsible for the general management of the entire embedded system, including running a full operating system. Binning is necessarily simple, and likely to become even simpler in future generations.

Binning occurs in two stages; the first stage sorts each vertex into binning regions, and the second stage sorts each primitive into binning regions. Like slice planes, bin regions are bounded by planes coincident with the axis of rotation. Bin regions contain the slice volumes bound with the bin planes. Each slice in a bin must only render the primitives contained in the bin.

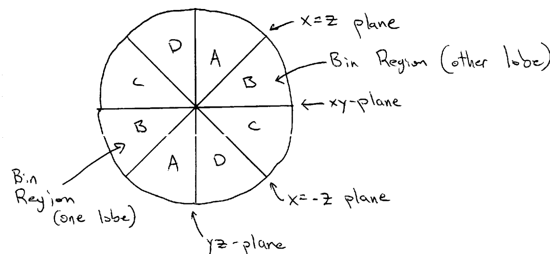


Figure 10: Bin Planes and Bin Regions

Vertex Outcodes

Binning computes the post-transform positions for each vertex. This process is different from executing the complete vertex program because optional attributes are not computed. From the post-transform position, each vertex is labeled with binning

outcodes. A binning outcode is the whether the vertex lies on the positive or negative half-space of a bin plane.

Bin planes are selected so that the half-space computation is as simple as possible. The planes are coincident with the axis of rotation (the y-axis), so the y and w coefficients must be 0. The coefficients of bin planes should only be 0, 1 and -1, so evaluating the plane equation only requires addition. The bin planes are the xy plane (0, 0, 1, 0), the yz plane (1, 0, 0, 0), the x=z plane (1, 0, -1, 0) and the x=-z plane (1, 0, 1, 0).

Primitive Outcodes

The vertex binning outcodes are used to compute primitive binning outcodes. The vertices in a primitive computes the primitive outcodes by combining the vertex outcodes with Boolean operations. The “and” of all the vertex outcodes in a primitive computes the bin planes for which the primitive is entirely on the positive side. Correspondingly, the “or” of all the vertex outcodes in a primitive computes the bin planes for which the primitive is entirely on the negative side (Figure 11: Outcode Example).

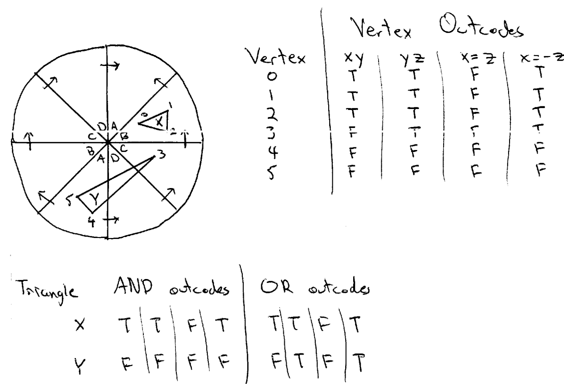


Figure 11: Outcode Example

Two bin planes bound each binning region. Primitive outcodes effectively determine the binning regions that the primitive intersects (primitives may extend into several binning regions) in a manner similar to frustum culling. Primitives entirely on the opposite side of a bin plane from the binning region are rejected.

Tiling

Perspecta slice images tend to be very sparsely populated; most of the content of a slice is empty space. Halftoning, packing, and downloading this empty space wastes GPU computation and bandwidth. Tiling provides a way to skip empty regions and improve halftoning, packing and downloading performance.

Tile Queries

GPUs provide a stencil buffer that may be used to mark regions that have been rendered to. They also provide occlusion queries that can be used to query regions of the stencil

buffer for marked regions. Before each slice is rendered, the stencil buffer is cleared to 0 and stencil writes are enabled. Every rendered fragment marks the stencil buffer as 1. When the slice rendering is complete, the GPU disables all framebuffer writes and sets the stencil test to check where the stencil buffer equals 1. Then, occlusion queries are issued by rendering tiles to the slice. Because framebuffer writes are disabled, this has no effect on the image itself. Because the stencil test is enabled, the occlusion query for each tile will return the number of pixels where then stencil buffer is 1.

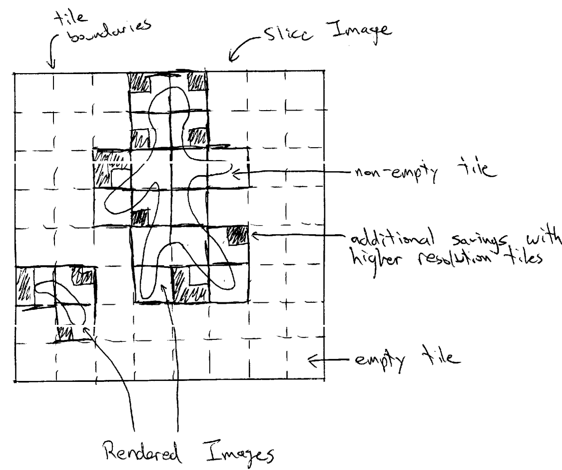


Figure 12: Tile Query Example

Now, it is known which tiles contain non-empty regions. Only these regions are halftoned, packed and downloaded. Smaller tiles are more effective at eliminating empty space, but smaller tiles may not be rendered as efficiently by the GPU, leading to a tradeoff. The current implementation of SpatialGL uses tiles that are 32 pixels square. For most scenes, only 10% of these tiles are non-empty, yielding large performance benefits.

Tile Batching

Performing stencil clears and issuing occlusion queries adds some overhead to slice rendering. Performing tiling on groups of slices significantly reduces this overhead; effectively, tiles extend into adjacent slices, making 3-D tiling regions. The current implementation of SpatialGL performs stencil clears and occlusion queries only every 4 slices.