Applications of Explicit Early-Z Culling

Jason L. Mitchell
ATI Research

Pedro V. Sander
ATI Research

Introduction

In past years, in the SIGGRAPH Real-Time Shading course, we have covered the details of real-time hardware shading on the R300 generation of products from ATI. Perhaps just as important as shading performance, however, is knowing when *not* to shade. The R300-based architectures from ATI have a variety of mechanisms for visibility-related culling, including asynchronous visibility queries, hierarchical-z culling and early-z culling. In these notes, we will outline a number of different applications which take specific advantage of early-z culling to achieve significant performance increases.

Early-Z

Prior to execution of the pixel shader, R300-based chips from ATI perform a check of the interpolated z value against the z value in the z buffer. This occurs for any pixels which passed the hierarchical z test and which are actually going to use the primitive's interpolated z (rather than compute z in the pixel shader itself). This additional check provides not only an added efficiency win when using long, costly pixel shaders, but also provides a form of pixel-level control flow in specific situations. In a number of applications such as volume rendering, skin shading and fluid simulation, the z buffer can be thought of as containing condition codes governing the execution of expensive pixel shaders. Inserting inexpensive rendering passes whose only job is to appropriately set the "condition codes" for subsequent expensive rendering passes can increase the performance of these three particular applications and certainly many more.

Volume Rendering

In the 2003 IEEE Visualization paper "Acceleration Techniques for GPU-based Volume Rendering," Krüger and Westermann take advantage of the early-z capabilities of the R300 to accelerate their ray-casting volume renderer [Krüger03]. Many volume data sets are sparse, opaque or both, as shown in Figure 1.

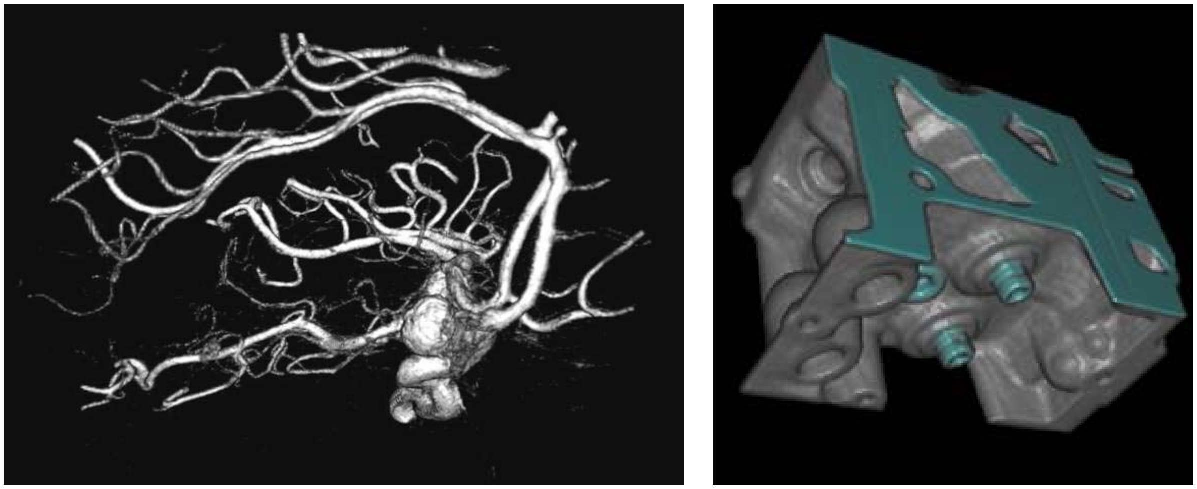


Figure 1 - Typical volume datasets. Only 2% of the fragments are actually visible.

Unfortunately, more traditional slice-based volume renderers are not able to take advantage of the sparse or opaque nature of these volume datasets and end up shading many more pixels than are necessary to compute the correct image. As shown in Figure 2, slice-based volume renderers draw a series of slices which are perpendicular to the viewing direction and clipped to fit inside of the volume dataset.

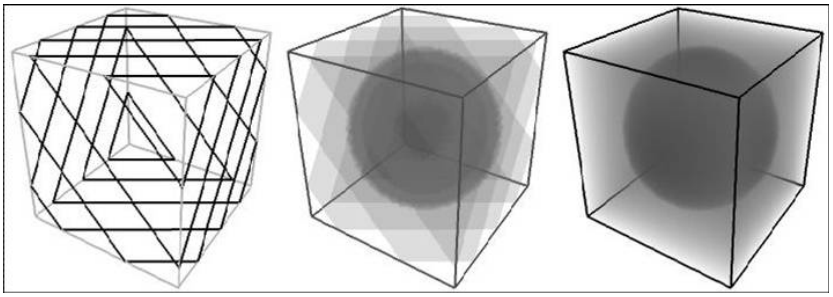


Figure 2 - Slice Based Volume Rendering

These planes are composited with the frame buffer in order to approximate the integrals of rays from the eye through each pixel. Computing these integrals is the essence of volume rendering. Unfortunately, on current graphics hardware, every pixel on every slice must execute the full pixel shader, even if the resulting pixel ends up contributing nothing to the final image due to the sample lying in an empty region of the dataset or due to it being occluded by an opaque region of the dataset. Krüger and

Westermann use early-z hardware to avoid computation of samples which are empty or occluded since the z buffer is not needed for hidden surface removal. This is true because the proxy geometry drawn (the bounding box of the volume data in Krüger and Westermann's raycaster) has a well-defined order. This is very important because it allows the z buffer to be used instead as an array of condition codes rather than an array of depths. Krüger and Westermann's volume renderer can use this "condition code" z buffer to halt and restart the computation along different rays through the volume dataset in order to stop rays which have hit opaque surfaces or are passing through large empty areas of the dataset.

The first step in their algorithm is to compute ray directions through the volume dataset for each pixel. This is done by performing a simple subtraction of the front and back faces of the bounding volume, which are shaded with their positions as shown in Figure 3.

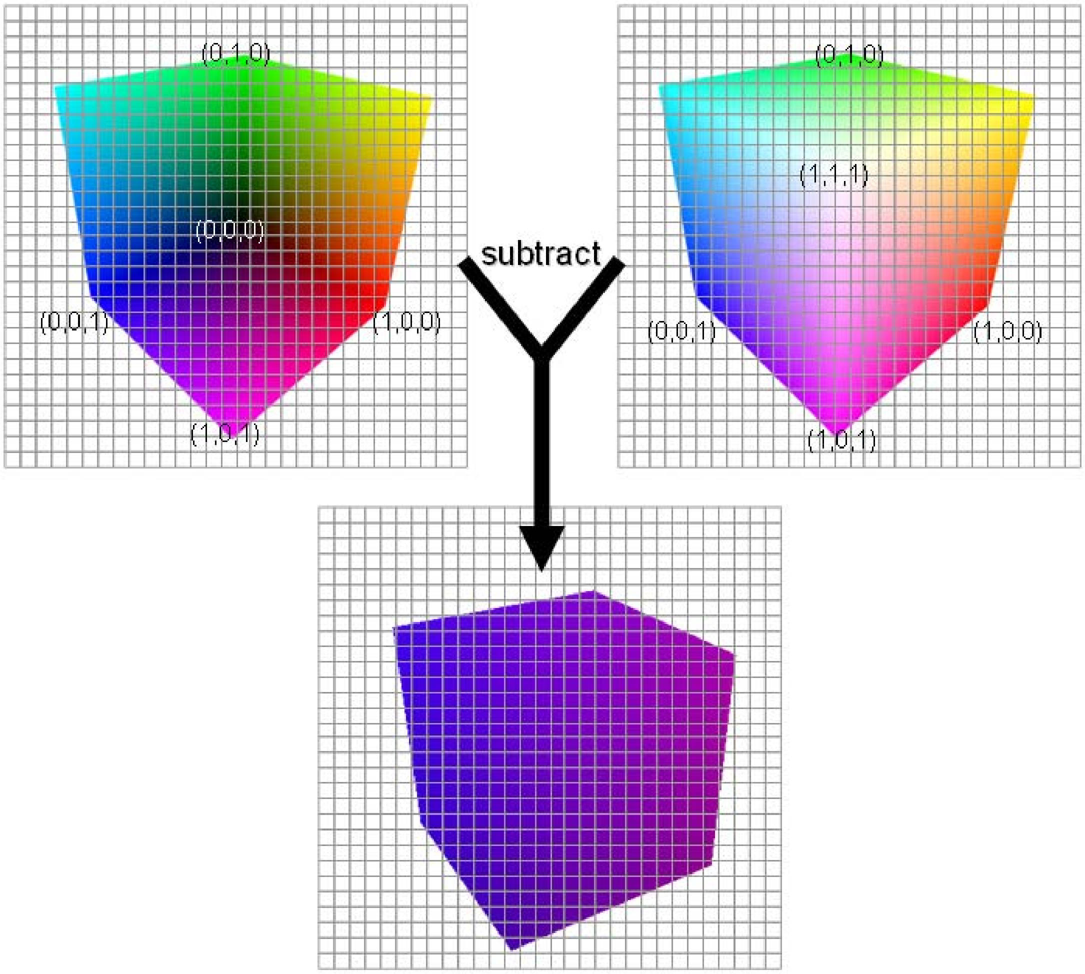


Figure 3 – Computing the rays through the volume dataset

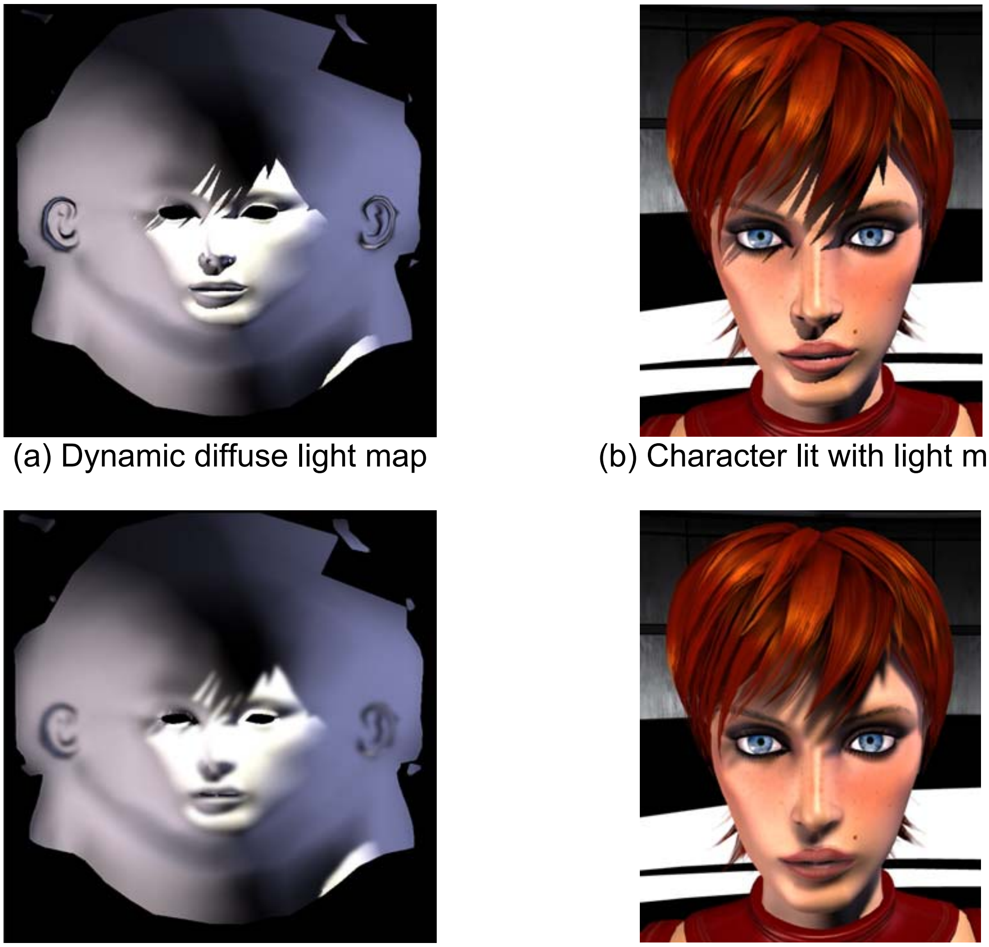
Subsequent steps in the raycasting algorithm march along these rays in parallel by repeatedly drawing the bounding box of the volume dataset as above. The pixel shader samples the volume dataset some number of times along the computed ray for the given pixel. Interleaved with these expensive passes that sample the volume multiple times and perform compositing are rendering passes that compute stopping criteria and set the z buffer accordingly. For example, if the intermediate "stopping criterion" pass discovers that a given pixel is already opaque, the z buffer will be set in such a way that early-z will keep any subsequent pixel shaders from being executed on that pixel. Rays which exit the volume are terminated similarly.

In addition to terminating rays that have hit opaque regions of the dataset or exited the volume, Krüger and Westermann's raycasting approach uses a hierarchical approach to skipping large areas of empty space. They store an additional volumetric texture which is 1/8th of the resolution of the original dataset on each axis. This texture contains the min and max opacities of the 512 original voxels which lie in each voxel of the low resolution version of the dataset. This low resolution texture can be accessed during the stopping criterion passes to skip over large empty areas of the dataset.

Skin Shading

For the animation *Ruby: The DoubleCross*, which appears in this year's SIGGRAPH animation festival, we adapted a skin rendering technique from last year's sketch on skin rendering in *The Matrix Reloaded* [Borshukov03]. In order to implement this technique in real-time, however, we made several modifications and optimizations, including the use of early-z culling to significantly speed up performance in cases where the character was not facing directly at the camera, was very far away from the camera, or was outside the view frustum. This is possible because the depth buffer is not needed for hidden surface removal during the texture-space image processing steps of the algorithm.

Briefly, the algorithm is based upon the observation that blurring a character's diffuse illumination in texture space gives a reasonably convincing approximation to subsurface scattering, a feature of real skin which is very important to its appearance. Prior to rendering the scene, we render a character's illumination to a light map as shown in Figure 4a.



(a) Dynamic diffuse light map

(b) Character lit with light map

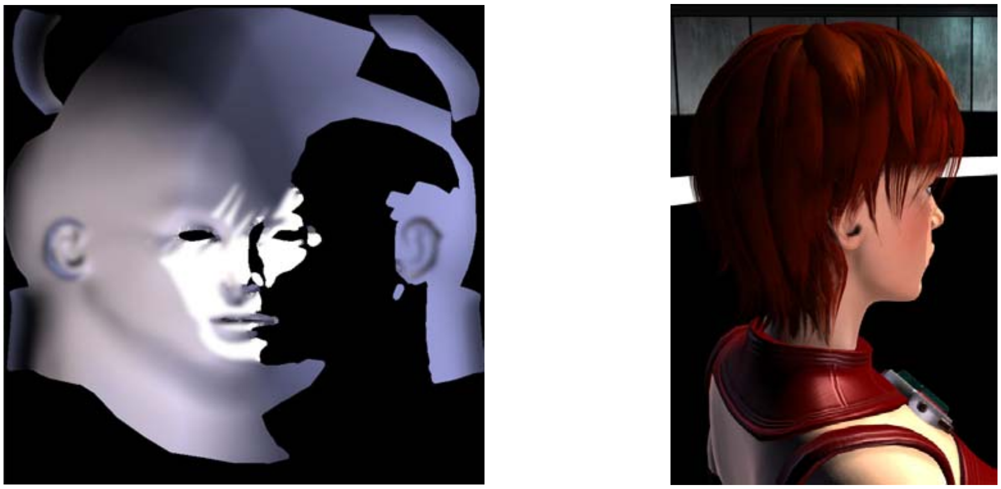
(c) Blurred light map

(d) Character lit with blurred light map

Figure 4 – Skin shading with texture space lighting

When we subsequently render the character to the back buffer, we use the light map as the source of diffuse illumination as shown in Figure 4b. If we blur the light map as shown in Figure 4c, we approximate the effect of subsurface scattering, which gives us more realistic shading on the skin as well as soft shadows as shown in Figure 4d.

The shaders used to process the light map in texture space can be fairly expensive, due to the large blur filter kernel and the fact that we have to effectively dilate the non-black areas of the map to account for edge cases. It would be nice to avoid executing these image processing pixel shaders where possible to speed up this step of the skin shading algorithm. Fortunately, since the z buffer is not necessary for hidden surface removal in the texture space steps of this algorithm, we can use the z buffer as a control-flow mechanism to skip processing for texels in the light map which are not visible. Figure 5a shows the light map used in the over-the-shoulder shot shown in Figure 5b.



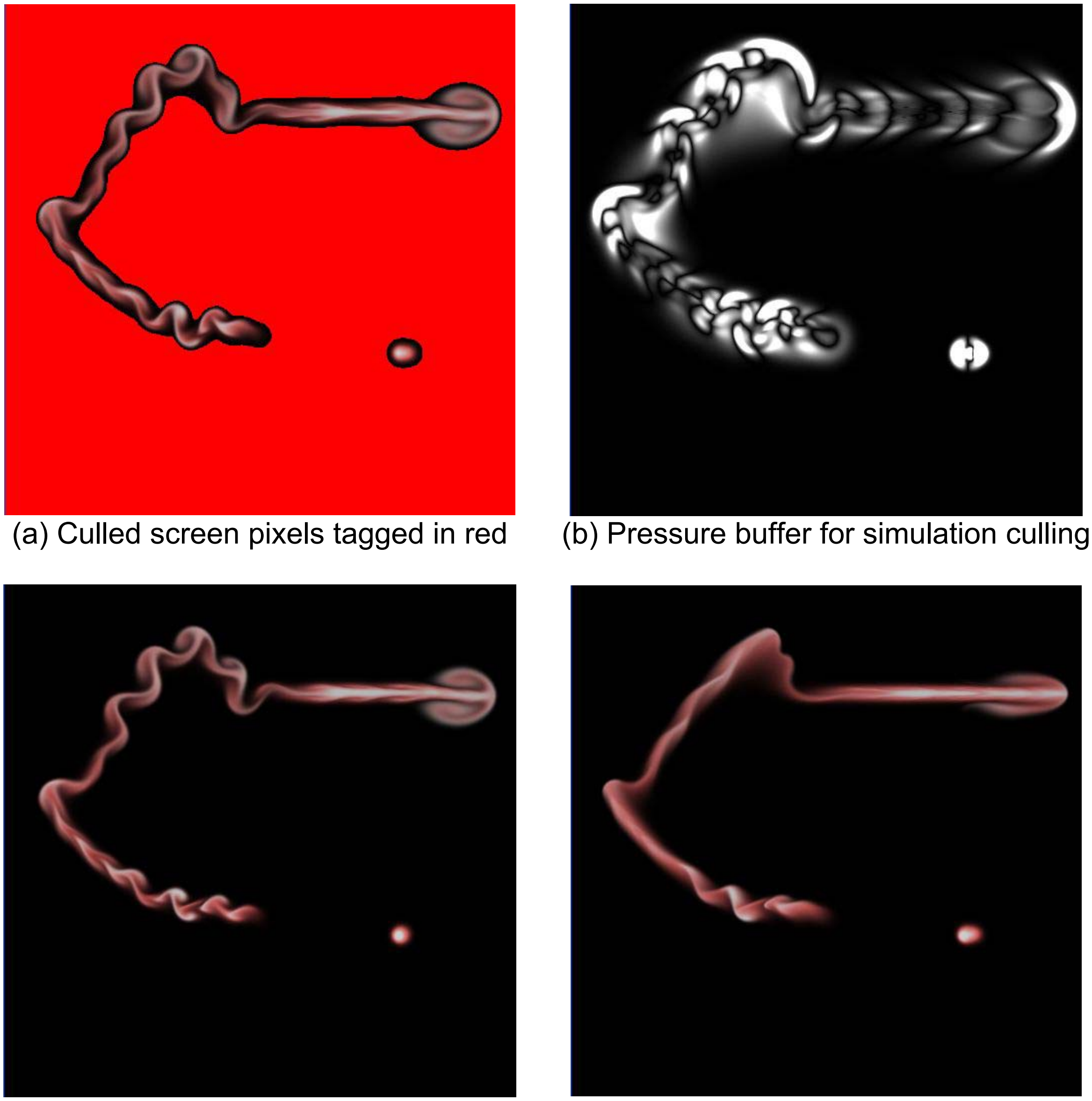
(a) Light map (b) Real-time character

Figure 5 – Texture space lighting for over-the-shoulder view

On the initial pass in which we draw the character's diffuse illumination into the light map, we check to see if a given polygon is front-facing with respect to the viewer. If it is, we set its depth to 0. Other polygons have their depth set to 1. We perform our image processing steps by simply drawing quadrilaterals. We set the depths of the quadrilaterals to 0 and the depth compare state to LEQUAL, thus causing the early-z hardware to cull pixels in the back-facing areas. Similarly, if the model is outside the view frustum, the geometry gets culled by our application, nothing is rendered to the z buffer, and therefore all pixels are culled during the blur pass.

Fluid Flow

As we have seen in the past few years at SIGGRAPH and other conferences, graphics processors are being applied to broader areas of computation, such as simulation. Due to their highly parallel nature and increasingly general computational models, GPUs are well matched with the demands of fluid simulation. We have implemented a Navier-Stokes fluid simulation on the GPU, including the use of early-z as a means of avoiding certain unnecessary computations, speeding up our simulations in some cases by a factor of three [Sander04].



(a) Culled screen pixels tagged in red

(b) Pressure buffer for simulation culling

(c) Early-Z

(d) Brute force

Figure 6 – Fluid flow Early-Z optimizations and comparison with Brute force

Our techniques are based on the observation that, in many applications, flow is often concentrated on certain regions of the simulation grid. The early-z optimizations that we employ significantly reduce the amount of computation on regions that have little or no flow, saving the computational resources for regions with higher flow concentration, or for other objects in the rendered scene. We present a culling technique for efficiently rendering fluid flow to the screen, and a culling technique for the projection step of the fluid flow simulation.

The first optimization we present is performed at rendering time. When rendering the 32-bit floating point density buffer to the screen, bilinear interpolation must be performed on the pixel shader. This computation can be expensive, especially if the dimensions of render target are significantly higher than those of the fluid simulation. In order to avoid applying the bilinear interpolation shader to regions of the screen that have very small density, prior to rendering the contents of the density buffer, we perform an inexpensive rendering pass that simply sets the z buffer value to the density of that particular pixel. When rendering, we set the depth compare state to LEQUAL, and set the Z value to 0.01 in the vertex shader. All pixels whose densities are smaller than 0.01 are culled. In Figure 6a, all pixels that were too dark to be visible were culled and are tagged in red for visualization purposes. (Figure 6c has the final rendering without the tagged pixels.)

The second optimization is performed during the most expensive step of the simulation: the projection step. This step is performed as a series of rendering passes to solve a linear system. We use a relaxation method for this step. The higher the number of iterations (rendering passes), the more accurate the result is. Instead of performing the same number of passes on all cells, we perform more passes on regions where the pressure is higher and less passes on regions with little or no density. This is accomplished by performing an additional inexpensive render pass that sets the z value of each cell in the simulation based on the maximum current value of that cell and its neighbors from the pressure buffer of the previous iteration of the simulation. Then, we set the depth compare state to LEQUAL and linearly increase the z value on each of the projection render passes. On the first pass, all cells are processed, and on the subsequent passes, the number of cells that are processed gradually decreases. Figure 6b shows the pressure buffer which is used to set the z buffer that cull the projection computation. Darker values indicate regions of lower pressure, where fewer iterations need to be performed.

Figures 6c and 6d compare a 512×512 fluid flow simulation with and without our early-z approach on both projection and rendering. Both simulation and rendering are done in approximately 50fps for both examples on a RADEON X800. Note that when early-z is used, the bulk of the projection computation can be concentrated on the region that has higher density, thus dramatically improving the result for the same rendering cost. The projection step on the early-z example has between 0 and 50 passes, while on the brute-force example it has exactly 10 passes for every cell.

These culling techniques are also suitable for fluid flow simulations with blockers. Since no computation needs to be performed on most cells that are blocked (e.g., white cells in Figure 7), these methods can further reduce computational costs. Note that blocked cells that have neighbors that are not blocked cannot be culled and need to be processed in order to yield the proper collision effects.

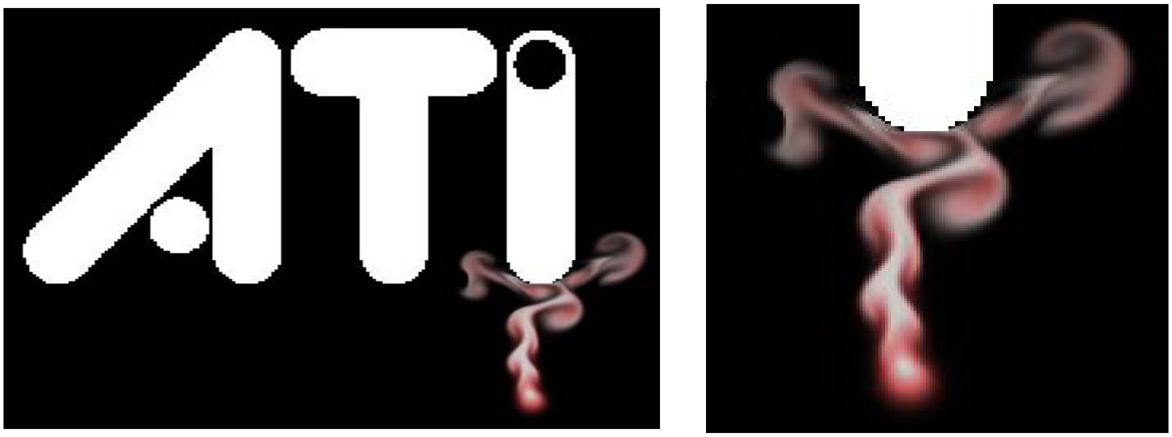


Figure 7 – Fluid flow with blockers

Conclusion

While raw shading performance is very important on modern graphics processors, perhaps just as important is knowing when *not* to shade. ATI graphics chips have a variety of mechanisms for visibility-related culling including early-z culling, which can be used in some non-obvious ways to achieve significant performance enhancements. We have outlined a number of different applications here which take specific advantage of early-z culling such as volume rendering, skin shading and fluid simulation.

References

[Borshukov03] George Borshukov and J. P. Lewis, *Realistic Human Face Rendering for "The Matrix Reloaded"* Technical Sketches, SIGGRAPH 2003.

[Krüger03] Jens Krüger and Rüdiger Westermann, "Acceleration Techniques for GPU-based Volume Rendering" IEEE Visualization 2003.

[Sander04] Pedro V. Sander, Natalya Tatarchuk and Jason L. Mitchell, "Explicit Early-Z Culling for Efficient Fluid Flow Simulation and Rendering," ATI Technical Report, August 2004.