Using GPU Shaders for Visualization

Mike Bailey, Oregon State University

Introduction

Most of the uses of GPU shaders seem to be for gaming and other forms of entertainment and simulation. And, why not? The effects that can be created are stunning, and definitely enhance the gaming experience. But, there are *visualization* uses for GPU shaders as well – for the same reasons: appearance and performance. In the drive to understand large, complex data sets, no method should be overlooked. This column looks at the use of GPU shaders and the GLSL shading language in two very common visualization applications: point clouds and contour cutting planes.

Previous Work

Using shaders for visualization started with experiments using RenderMan, e.g., [Corrie1993]. Interactive (i.e., graphics hardware-based) GPU Shaders appeared in the early 2000's [Mark2003]. Since then, researchers have pushed them into a variety of applications. Many of these have involved scientific and data visualization involving volume rendering [Stegmaier2005], level of detail management [Petrovic2007], volume segmentation [Sherbondy2003], and level sets [Lefohn2003]. Other work has used GPU programming to combine and filter visualization data to show particular features of interest [McCormick2004].

Reading 3D scalar data into a shader

Shaders were designed to accept relatively small sets of scene-describing graphics attributes such as colors, coordinates, vectors, and matrices, as input data. Passing general-purpose large amounts of data into them, such as through uniform variables, is inefficient. It is better to find some way that looks more consistent with the graphics intent of shaders. In this case, an excellent approach is to hide the data in a 3D texture.

Textures, were designed to store RGBA color values. The most common format for textures, still, is probably the unsigned byte format, specifically created to hold 8-bit color components. However, today's graphics cards can also use 16- and 32-bit floating point formats to store texture components. Thus, textures can hold any scale of numbers , within the limits of those floating-point formats, that we want. This makes them ideal as a way to hold 3D data for visualization.

Point Clouds

However, a 3D texture is just data and data, by itself, cannot be displayed. It needs some sort of geometry to hang itself on, or more accurately, it needs a geometry to map itself to. A good start is to map it to a *3D point cloud*, a uniform mesh of 3D points. When you map the temperature distribution dataset above to a point cloud, you get the image in Figure 1:

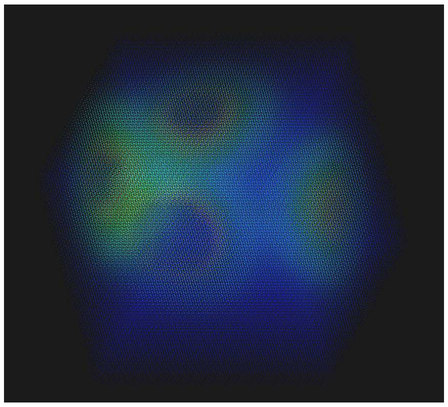


Figure 1. Point cloud in orthographic projection

One of the interesting aspects of this approach is that the resolution of the point cloud does not have to exactly match the resolution of the dataset. Because this example uses texture mapping to access the data, the OpenGL display process will trilinearly interpolate the data values in the texture to the cloud's 3D point locations. Making the resolution of the point cloud much *less* than that of the data is usually a bad idea, since some of the data values will be completely skipped over in the display. A little less is not desirable, but not bad – the trilinear interpolation fills in the values nicely. You can also give the point cloud a *higher* resolution than the data and get a really nice-looking display.

Using a higher point cloud resolution assumes, of course, that interpolation makes sense for the particular data you have. It doesn't always. For example, suppose the data values represent integer-only data, such as the number of children per family. Even though a point cloud dot could exist midway between two data values, it makes no sense to combine half of one with half of the other to produce a data point that represents a fraction of a child. In this case, the resolution of the point cloud should be the *same* as the resolution of the data.

The vertex shader in this case is simple. It just needs to record model coordinates and do the proper transformations:

```
varying vec3 MCposition;

void
main( void )
{
   MCposition = gl_Vertex.xyz;
   gl_Position =
gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

The fragment shader uses those model coordinates to determine where each fragment is in texture coordinate space, and thus what its data value is there. I personally like thinking of the data as living in a cube that ranges from -1 to 1 in all directions. It is easy to position geometry in this space and easy to view and transform it. This means that *any* 3D object in that space, not just a point cloud, can map itself to the 3D texture data space. So, if we want the *s* texture coordinates to go from 0 to 1, then the linear mapping from the physical *x* coordinate to the texture *s* coordinate is $s = \frac{x+1}{2}$. The same mapping applies to *y* and *z* to create the *t* and *p* texture coordinates. Once we have the *s-t-p* texture coordinates, we can look up the data value at that location, which is then used to set the color for this fragment:

```
varying vec3 MCposition;

void
main( void )
{
   vec3 stp = ( MCposition + 1. ) / 2.; //
maps [-1.,1.] to [0.,1.]

   if( any( lessThan( stp, vec3(0.,0.,0.) )
) )
      discard;

   if( any( greaterThan( stp,
vec3(1.,1.,1.) ) ) )
      discard;

   float scalar = texture3D( TexUnit, stp
).r;

   if( scalar < Min )
      discard;
   if( scalar > Max )
      discard;

   float t = ( scalar - SMIN ) / ( SMAX -
SMIN );
   vec3 rgb = Rainbow( t );

   gl_FragColor = vec4( rgb, 1. );
}
```

Note that the SIMD parallelism inherent in the GPU is being taken advantage of by (1) computing the *s-t-p* mapping from *x-y-z* in a single statement, and (2) checking for a fragment living beyond the bounds of the data in single if-tests.

Also notice how the `discard` operator is used in this case -- it allows us to eliminate any points that lie outside our data areas of interest. It doesn't have to end there, though. We could also have this shader cull data values based on lots of criteria, such as physical location, or even based on some derived properties such as data gradient or data curvature. Another variation could allow us to use the vertex shader code to set the point size based on some physical or data criterion.

Point clouds are notorious for their artifacts, especially the row-of-corn problem in orthographic projection and Moiré patterns in perspective. A common way to alleviate these artifacts is to use a different type of point cloud, known as a *jitter cloud*. In a jitter cloud, the dots are randomly shifted by small amounts in *x*, *y*, and *z*, and the data values are reinterpolated to those new points. Because the *s-t-p* coordinates are computed automatically from the *x-y-z* model coordinates, the point data display is still correct. Results from using a jitter cloud in orthographic and perspective are shown in Figure 2.

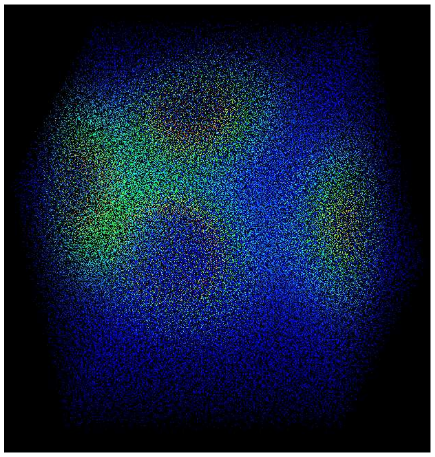
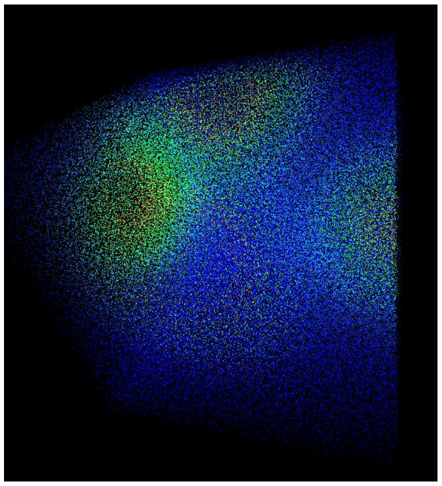



Figure 2. Jittered point cloud in orthographic and perspective projections

Cutting Planes

There are two general kinds of cutting planes. In one, you interpolate data values (and thus colors) at each pixel in the plane, and in the other, you create contour lines in a reduced set of pixels. As before, the color interpolation approach requires some sort of geometry to hang the data on. In this case, we use a quadrilateral as the geometric primitive.

The interesting part is that the code for the vertex and fragment shaders is nearly the same as the code for the point cloud shaders above. Figure 3 below shows how this looks:

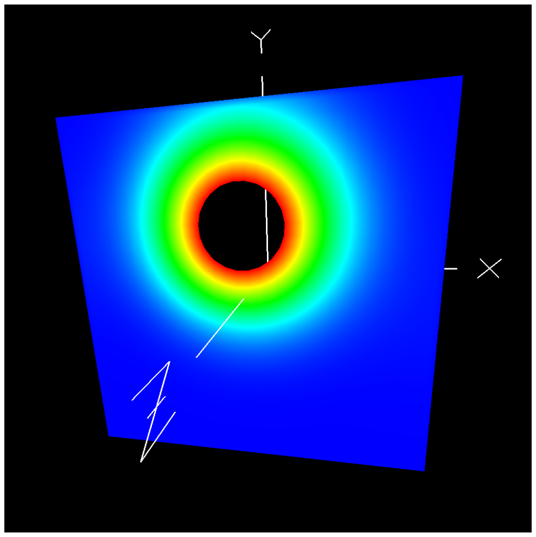


Figure 3. Interpolated Color Cutting Plane

Now, let's change the fragment shader to create contour lines. There are geometric ways to create contour lines with real OpenGL line segments, but for this example, we will use almost the same fragment shader code as we did above. Let's say that we want contour lines at each 10 degrees of temperature. Then the main change to the shader will be that we need to find how close each fragment's interpolated scalar data value is to an even multiple of 10. To do this, we add this code to the fragment shader:

```
    float scalar10 = float( 10*int(
(scalar+5.)/10. ) );
    if( abs( scalar - scalar10 ) > Tol )
        discard;
```

Notice that this uses a uniform variable called `Tol`, which is read from a slider and has a range of 0. to 5. `Tol` is used to determine how close to an even multiple of 10 degrees we will accept, and thus how thick we want the contours to be. Various values for *Tol* produce the individual images in Figure 4:

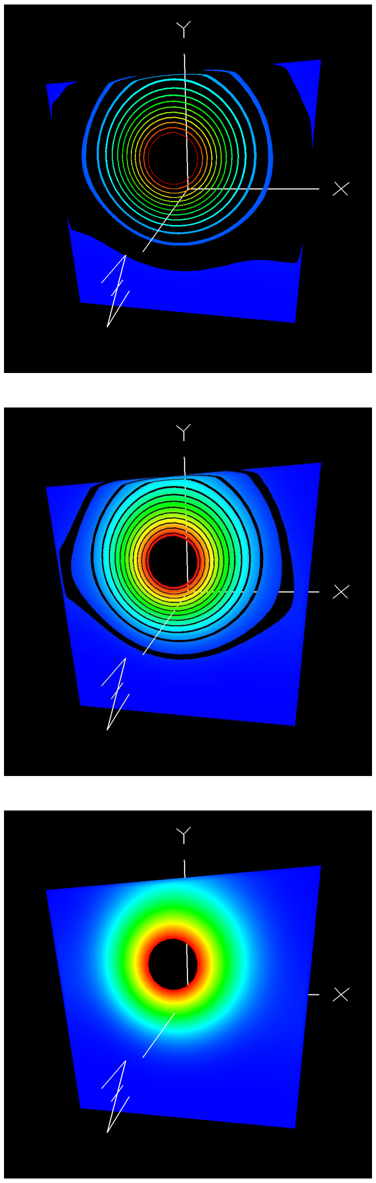
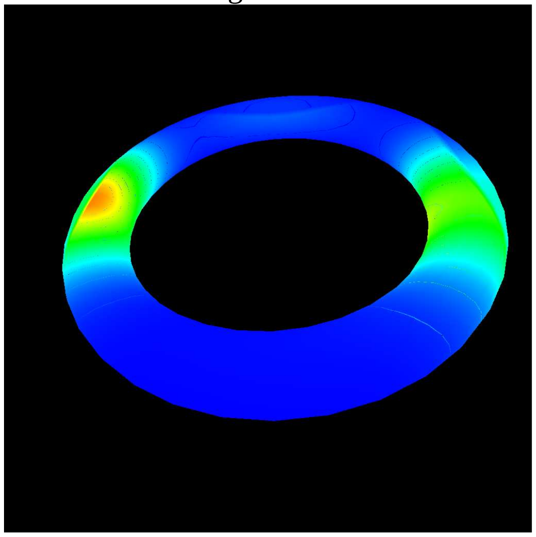


Figure 4. Contour lines using `Tol` values of 1, 4, and 5.

Take a close look at what this fragment-based approach to contours gets you compared with a line-based approach. Notice that the contours have different thicknesses. This is an indication of how much area was within `Tol` of a 10-degree value. Standard contour lines show the gradient, how fast the data is changing, by how closely spaced they are. This new method shows the gradient in a different way – it also lets us see how fast the data is changing based on the thickness of the contour. Thus, we can tell that the data is changing slower at the blue areas than at the red areas.

Also, notice that when `Tol=5.,` the `Tol` if-statement always fails, and we end up with the same display as we had with the interpolated colors. Thus, we wouldn't actually need a separate *cutting plane* shader at all. Shaders that can do double duty are always appreciated!

It is important to notice that the shaders maintain the mapping from the coordinates of the cutting planes to the texture coordinates that hold the data. This means that the cutting planes do not need to be oriented parallel to principal axes, but can be rotated into any orientation. It also means that the cutting geometry does not even need to be a plane at all. It can be any shape for which you can produce the coordinates-to-texture mapping. We saw this before in the point and jitter cloud examples. It is also seen below, where a torus is being used as a "cutting plane" (although I would more likely call this a *data probe)*. And, like before, we have also played the "contouring trick" using the `Tol` uniform variable. This, again, shows that the data is changing slower in the blue regions and faster in the red regions.



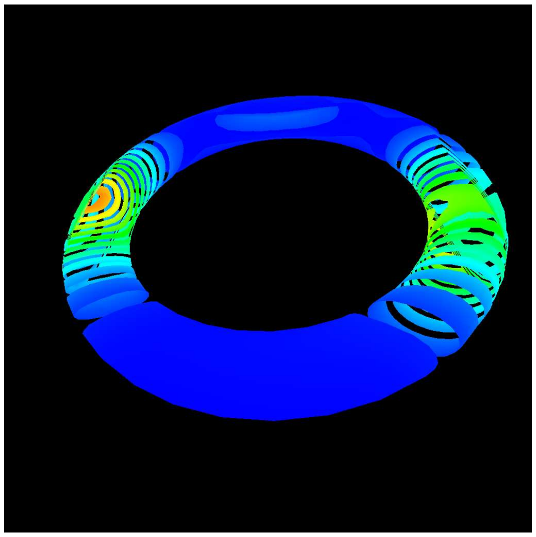
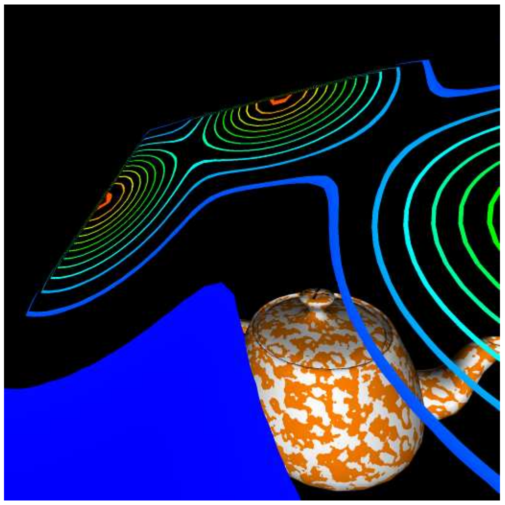
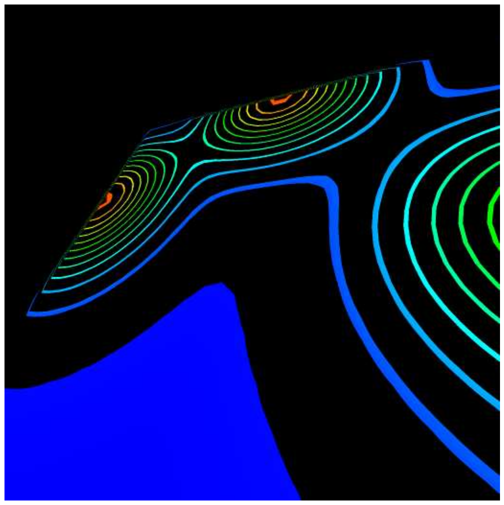


Figure 5. Torus 3D data probe, without and with contour tolerances

Discard versus setting alpha

In these examples, we used the fragment shader `discard` operator to eliminate fragments. Another way to do this might have been to set the opacity, *alpha*, to 0. But, this would not work. Can you figure out why? The images below show a 3D object (our favorite teapot) sitting behind a cutting plane. When `discard` is used (top), you see through it just fine. When *alpha*=0.is used (bottom), somehow you can't





The answer is in how OpenGL performs its alpha blending. Even though you and I both know that *alpha*=0. means not to display the fragment, OpenGL does not know this. It just knows to perform a blending using *alpha*=0. and then put the resulting fragment back in the framebuffer. The problem is that putting this pixel back in the framebuffer sets its Z value into the z-buffer as well. So, even though the pixel looks like it is not there, it really is, and its z-buffer value blocks the display of items behind it.

Conclusions

We usually think of data-mapping visualization techniques such as point clouds, jitter clouds, cutting planes, contour planes, and data probes as *different* techniques, but in fact they have more in common than they have differences – they are all part of a family of techniques that map data display to arbitrary geometry. This can especially be seen in that the shader code to implement them is largely the same – it is mostly the underlying geometry that changes.

This gives a lot of freedom to the person doing the visualization programming. The choice of underlying geometry can be made based on what matches the inherent characteristics of the visualization situation rather than what is simply available. Hopefully, this idea will be used to uncover new geometric shapes to map the data too, and in doing so, will reveal new insights into the nature of the data itself.

References

Corrie1993
Brian Corrie, Paul Mackerras. “Data Shaders,” *Proceedings of IEEE Visualization 1993*, pp. 275-282.

Lefohn2003
Aaron Lefohn, Joe Kniss, Charles Hansen, Ross Whitaker, “Interactive Deformation and Visualization of Level Set Surfaces Using Graphics Hardware”, *Proceedings of IEEE Visualization 2003*, pp. 75-82

Mark2003
William Mark, Steven Glanville, Kurt Akeley, and Mark Kilgard, “Cg: a system for programming graphics hardware in a C-like language,” *Computer Graphics* (Proceedings of SIGGRAPH 2003), pp. 896-907.

McCormick2004
Patrick McCormick, Jess Inman, James Ahrens, Charles Hansen, Greg Roth, “Scout: A Hardware-Accelerated System for Quantitatively Driven Visualization and Analysis, *Proceedings of IEEE Visualization 2004*, pp. 171-179. *International Workshop on Volume Graphics*,, 2005, pp. 187-241.

Petrovic2007
V. Petrovic, J. Fallon, F. Kuester, “Visualizing Whole-Brain DTI Tractography with GPU-based Tuboids and Level of Detail Management”, *IEEE Transactions on Visualization and Computer Graphics*, Volume 13, Number 6, Nov.-Dec. 2007, pp. 1488 – 1495.

Scharsach 2005
H. Scharsach, “Advanced GPU raycasting”, *Central European Seminar on Computer Graphics* 2005, pp. 69-76.

Sherbondy2003

Anthony Sherbondy, Mike Houston, Sandy Napel, "Fast Volume Segmentation with Simultaneous Visualization using Programmable Graphics Hardware", *Proceedings of IEEE Visualization2003*, pp. 171-176..

Stegmaier2005
S. Stegmaier, M. Strengert, T. Klein, T. Ertl, "A simple and flexible volume rendering framework for graphics-hardware-based raycasting". *Fourth International Workshop on Volume Graphics*, 2005, pp. 187-241.