Visualization Viewpoints

Editor:
Theresa-Marie Rhyne

Using GPU Shaders for Visualization

Mike Bailey,
Oregon State University

GPU shaders seem to be used mostly for gaming and other forms of entertainment and simulation. And why not? They can create stunning effects that definitely enhance the gaming experience. But GPU shaders have visualization uses for the same reasons: appearance and performance. In the drive to understand large, complex data sets, no method should be overlooked. Here I examine the use of shaders and the OpenGL Shading Language (GLSL) in two common visualization applications: *point clouds* and *cutting planes*. (For a brief history of shaders in visualization, see the sidebar.)

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 large amounts of general-purpose data into them, such as through uniform variables, is inefficient. It's better to find a way that's more consistent with the graphics intent of shaders. An excellent approach is to hide the data in a 3D texture.

Textures were designed to store RGBA color values. The most common texture format is still probably the unsigned-byte format, 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. So, textures can hold any scale of numbers we want, within those formats' limits. This makes them ideal for holding 3D data for visualization.

Point Clouds

However, a 3D texture is just data, and data, by itself, can't 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. Figure 1 shows a point cloud based on a temperature distribution data set.

One interesting aspect of this approach is that the point cloud's resolution doesn't have to exactly match the data set's resolution. 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 point cloud's resolution much less than that of the data is usually a bad idea because the display will skip some data values. A little less isn't desirable but isn't bad—the trilinear interpolation fills in the values nicely. We can also give the point cloud a higher resolution than the data and get a display with extra data-smoothing.

Using a higher point-cloud resolution assumes, of course, that interpolation makes sense for our particular data. 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 representing a fraction of a child. In this case, the point cloud's resolution should be the same as the data's.

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 like thinking of the data as living in a cube that ranges from –1 to 1 in all directions. It's easy to position geometry in this space and easy to view and transform it. This means that any 3D object in this space, not just a point cloud, can map itself to the 3D texture data space. So, if we

 Published by the IEEE Computer Society

Previous Work on GPU Shaders for Visualization

Using shaders for visualization started with experiments using RenderMan.[1] Interactive (that is, graphics-hardware-based) GPU shaders appeared in the early 2000's.[2] Since then, researchers have pushed them into a variety of applications. Many of these have involved scientific and data visualization involving volume rendering,[3] level-of-detail management,[4] volume segmentation,[5] and level sets.[6] Other research has used GPU programming to combine and filter visualization data to show particular features of interest.[7]

References

1. B. Corrie and P. Mackerras, "Data Shaders," *Proc. 1993 IEEE Conf. Visualization*, IEEE Press, 1993, pp. 275–282.
2. W. Mark et al., "Cg: A System for Programming Graphics Hardware in a C-Like Language," *ACM Trans. Graphics* (Proc. Siggraph), vol. 22, no. 3, 2003, pp. 896–907.
3. S. Stegmaier et al., "A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-Based Raycasting," *Proc. 4th Int'l Workshop Volume Graphics*, IEEE Press, 2005, pp. 187–241.
4. V. Petrovic, J. Fallon, and F. Kuester, "Visualizing Whole-Brain DTI Tractography with GPU-Based Tuboids and LoD Management," *IEEE Trans. Visualization and Computer Graphics*, vol. 13, no. 6, 2007, pp. 1488–1495.
5. A. Sherbondy, M. Houston, and S. Napel, "Fast Volume Segmentation with Simultaneous Visualization Using Programmable Graphics Hardware," *Proc. IEEE Visualization 2003* (VIS 03), IEEE Press, pp. 171–176.
6. A. Lefohn et al., "Interactive Deformation and Visualization of Level Set Surfaces Using Graphics Hardware," *Proc. IEEE Visualization 2003* (VIS 03), IEEE Press, pp. 75–82.
7. P. McCormick et al., "Scout: A Hardware-Accelerated System for Quantitatively Driven Visualization and Analysis," *Proc. IEEE Visualization 2004* (VIS 04), IEEE Press, pp. 171–179.

want the s texture coordinates to go from 0 to 1, the linear mapping from the physical x coordinate to the texture s coordinate is $s = (x + 1)/2$. 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 we then use to set this fragment's color:

```
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. );
}
```

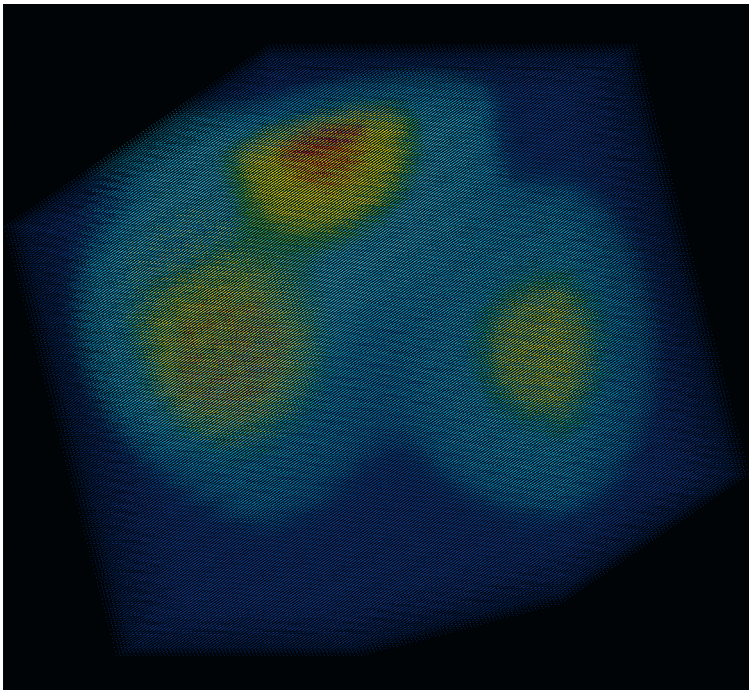
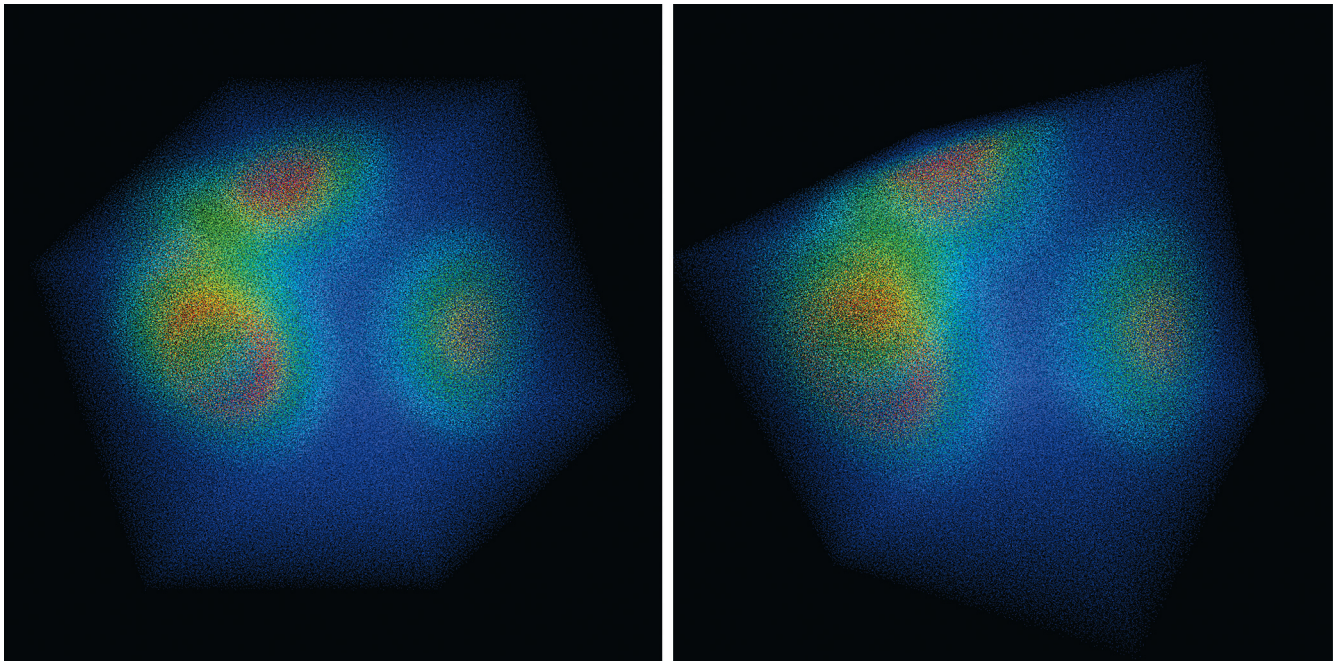


Figure 1. A point cloud in orthographic projection. The general pattern of the data is visible, but it's difficult to see into the data set.

We exploit the GPU's inherent SIMD (single instruction, multiple data) parallelism by

- computing the s-t-p mapping from x-y-z in a single statement and



(a) (b)

Figure 2. A jitter cloud in (a) orthographic and (b) perspective projections. Notice how the jittering eliminates the row and moiré artifacts that are common in unjittered point clouds.

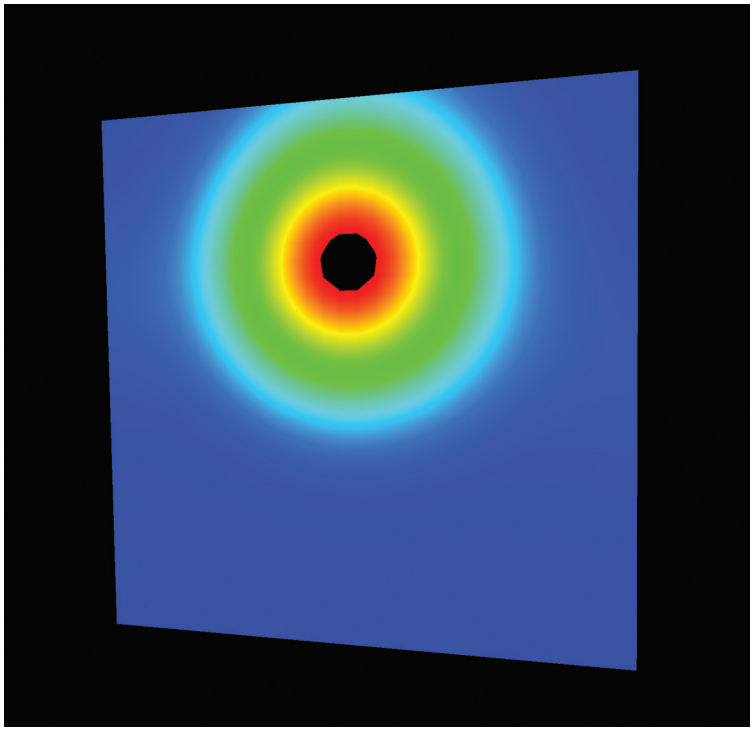


Figure 3. An interpolated-color cutting plane. This shows the full range of data values at all locations on the plane.

- using single If tests to check for fragments beyond the data's bounds.

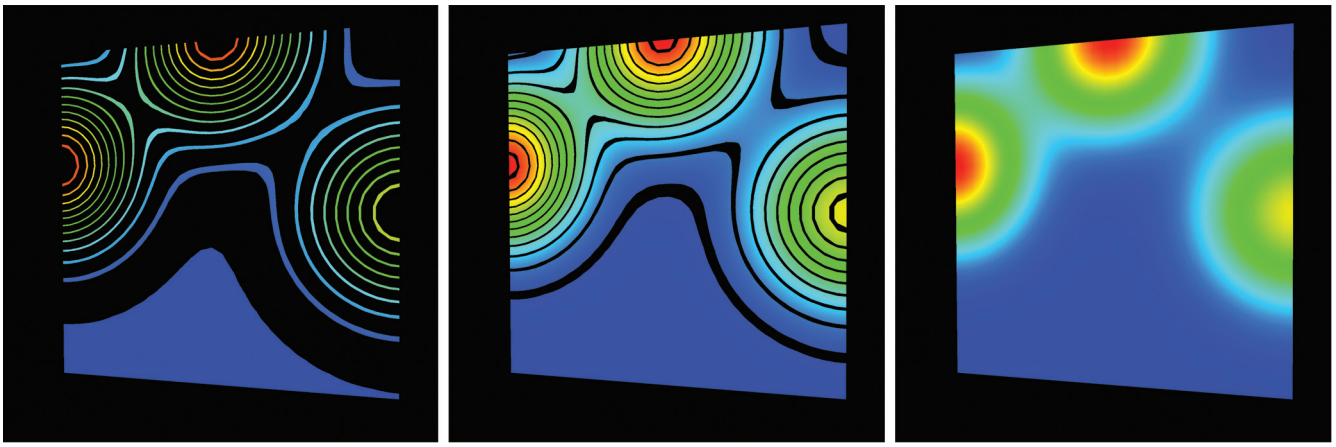
In this case, the `discard` operator lets us eliminate any points outside our data areas of interest. It doesn't have to end there, though. We could also have this shader cull data values on the basis of many criteria, such as physical location, or even on the basis of some derived properties such as data gradient or data curvature. Another variation could let us use the vertex shader code to set the point size on the basis of 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. (In the row-of-corn problem, the parallel rows of data align in a parallel projection, so you can see only the data point in front and can't see into the data set.) A common way to alleviate these artifacts is to use a different type of point cloud called a *jitter cloud*. In a jitter cloud, we randomly shift the dots by small amounts in *x*, *y*, and *z* and reinterpolate the data values to those new points. Because this approach automatically computes *s*-*t*-*p* texture coordinates from the *x*-*y*-*z* model coordinates, the point data display is still correct. Figure 2 shows a jitter cloud in orthographic and perspective projections.

Cutting Planes

There are two general kinds of cutting planes. In one, we interpolate data values (and thus colors) at each pixel in the plane; in the other, we create contour lines on a reduced subset of pixels. As with point clouds, 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 that for the point-cloud shaders. Figure 3 shows the resulting interpolated-color cutting plane.



(a) (b) (c)

Figure 4. Contour lines using `Tol` values of (a) 1, (b) 4, and (c) 5. This shows a subset of the data values on the plane. The width of the contours shows how slowly the data values are changing.

Let's change the fragment shader to create contour lines. There are geometric ways to create contour lines with real OpenGL line segments. However, for this example we'll use almost the same fragment shader code as we did previously. Let's say 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;
```

This code uses a uniform variable called `Tol`, which is read from a slider and ranges from 0 to 5. We use `Tol` to determine how close to an even multiple of 10 degrees we'll accept, and thus how thick we want the contours to be. Various values for `Tol` produce the images in Figure 4.

Look closely at what this fragment-based approach to contours gets us compared with a line-based approach. The contours have different thicknesses. This indicates how much area was within `Tol` of a 10-degree value. Standard contour lines show the gradient—that is, how fast the data is changing—by how closely spaced the contours are. This new method shows the gradient in a different way. It also lets us see how fast the data is changing, on the basis of the contour's thickness. So, we can tell that the data is changing more slowly in the blue areas than in the red areas.

When `Tol` is 5, the `Tol` If statement always fails, and we end up with the same display we had with the interpolated colors. So, we wouldn't actually need a separate cutting-plane shader at all. Shaders that can do double duty are always appreciated!

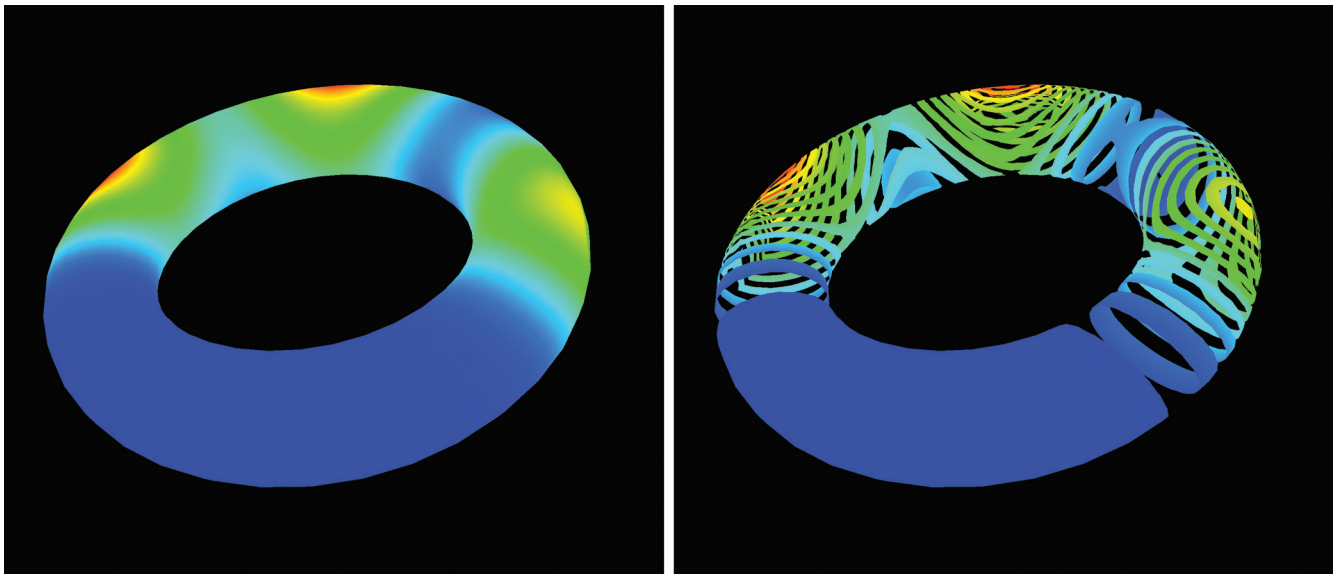
The shaders maintain the mapping from the cutting planes' coordinates to the texture coordinates holding the data. This means that we don't need to orient cutting planes parallel to principal axes, but can rotate them into any orientation. It also means that the cutting geometry doesn't even need to be a plane at all. It can be any shape for which we can produce the coordinates-to-texture mapping. We saw this before in the point- and jitter-cloud examples. We also see it when we use a torus (see Figure 5) as a "cutting plane" (although I would more likely call this a *data probe*). Like before, we played the "contouring trick" using `Tol`. This again shows that the data is changing more slowly in the blue areas than in the red ones.

Discard versus Setting Alpha

Another way to eliminate fragments might be to set the opacity, *alpha*, to 0. But this won't work. Can you figure out why? Figure 6 shows a 3D object (our favorite teapot) sitting behind a cutting plane. When we use `discard` (see Figure 6a), we see through the cutting plane just fine. When we use *alpha* = 0 (see Figure 6b), somehow we can't.

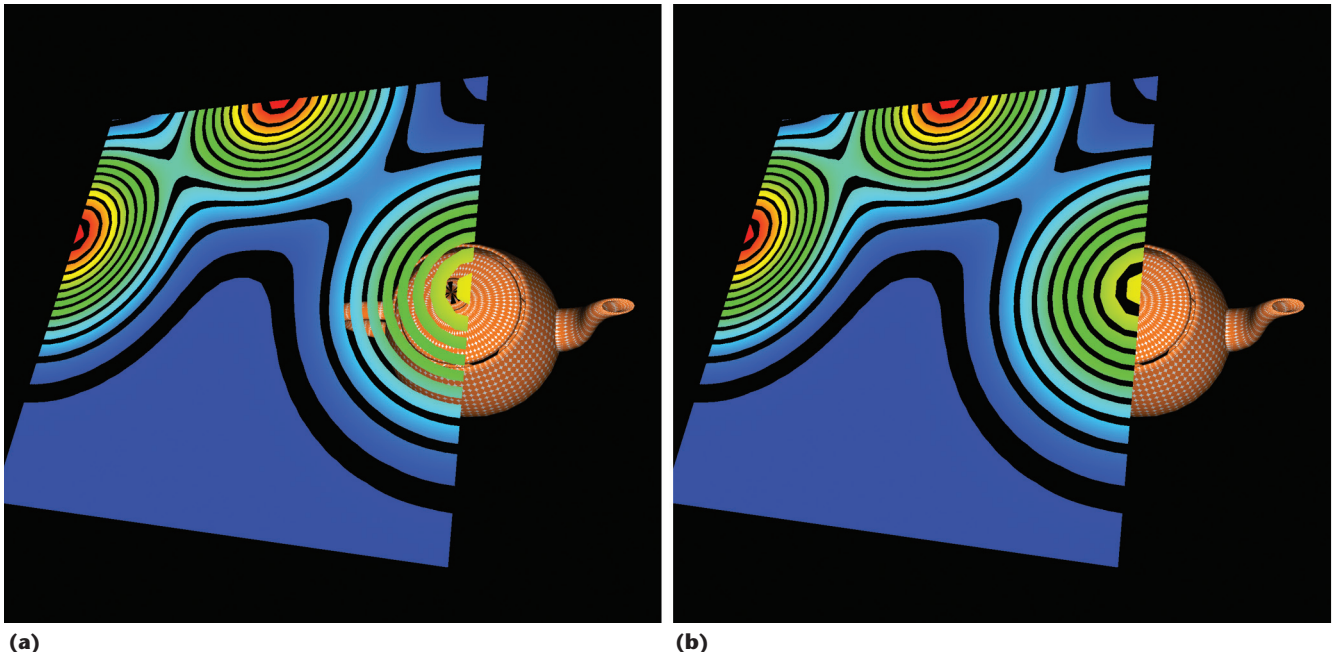
The answer is in how OpenGL performs its alpha blending. Even though you and I both know that *alpha* = 0 means not to display the fragment, OpenGL doesn't know this. It just knows to perform a blending using *alpha* = 0 and then put the resulting fragment back in the frame buffer. The problem is that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

We usually think of data-mapping visualization techniques such as point clouds, jitter clouds, cutting planes, contour planes, and data probes as different techniques. However, they have more similarities than differences—they're all part of a family of techniques that map data display to arbitrary geometry. We can especially see this in that the shader



(a) (b)

Figure 5. A torus 3D data probe (a) without and (b) with contour tolerances. Because the fragment shader always maps 3D coordinates to data values, any shape of probe can be used. This means that point clouds, cutting planes, and 3D data probes are really all the same technique.



(a) (b)

Figure 6. A 3D object sitting behind a cutting plane. (a) When we use the fragment shader `discard` operator to eliminate fragments, we can see through the cutting plane as we would hope. (b) When we set the opacity, alpha**, to 0, we can't. This is because OpenGL doesn't know that** alpha = **0 means not to display the fragment at all.**

code to implement them is largely the same—the underlying geometry is really what changes.

This gives a lot of freedom to the person doing the visualization programming. Programmers can choose the underlying geometry on the basis of what matches the visualization situation's inherent characteristics, rather than simply what's easily available. Hopefully, we can use this idea to uncover new geometric shapes to map the data to and, in doing so, will reveal new insights into the nature of the data itself.

***Mike Bailey** is a professor of computer science at Oregon State University. Contact him at mjb@cs.oregonstate.edu.*

Contact department editor Theresa-Marie Rhyne at tmrhyne@nscu.edu.