Visualization Viewpoints

Editor:
Theresa-Marie Rhyne

Using GPU Shaders for Visualization, Part 2

Mike Bailey
Oregon State University

GPU shaders aren't just for special effects. Previously, I looked at some uses for them in visualization.[1] Here, the idea continues. Because visualization relies so much on high-speed interaction, we use shaders for the same reason we use them in effects programming: appearance and performance. In the drive to understand large, complex data sets, no method should be overlooked. This article describes two additional visualization applications: line integral convolution (LIC) and terrain bump-mapping. I also comment on the recent (and rapid) changes to OpenGL and what these mean to educators. (For a look at previous research on using shaders for 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, normals, and matrices, as input data. Passing large amounts of general-purpose data into them—for example, through uniform variables—is inefficient. It's better to find some way that looks more consistent with shaders' graphics intent. 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 probably still 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 almost any scale of numbers that we want, within those floating-point formats' limits. This makes them ideal for holding 3D data for visualization.

2D LIC

LIC is a technique for visualizing flow directions across a 2D field.[2] It essentially smears an image in the flow's directions. The input image is often a white-noise texture (see Figure 1a), so the result is a grayscale smear (see Figures 1b through 1d).

Figure 2 shows a fragment shader to implement the LIC algorithm. The algorithm will access two textures: one for the background image being smeared and one to hold the flow field. If the flow field image is an unsigned-byte texture, we must scale the flow velocities to fit in the range 0 to 255 when creating the texture. Once those byte-texture components are in the shader, they'll have been scaled into the range 0. to 1. and thus must be unscaled. In the example shown here, the algorithm packs the flow velocities into a floating-point texture so that it can use the values as is—with no scaling and unscaling. Because this is a 2D flow field, the *x* and *y* velocities are packed into just the red and green texture components.

The geometry drawn is a four-vertex 2D quadrilateral. Figure 3 shows the vertex shader. The `out` variable `vST` is the `s` and `t` texture coordinates at each vertex and will eventually be interpolated and sent to the fragment shader via the rasterizer.

In the fragment shader in Figure 2, `fFragColor` is the color that will fill each pixel. `uLength` is a uniform variable that controls how many samples are taken in each direction along the flow lines.

The highly parallelized fragment processors produce excellent interactive performance. I ran this on an Nvidia GTX 480 card with a window of 1,024 × 1,024 pixels. For a `uLength` of 25, the process ran at approximately 2,000 frames per second. (The timing just measured the graphics display time; it didn't wait around for the `SwapBuffers`.) When I increased `uLength` to 100, the speed dropped to a "mere" 900 fps. By doing the timing without the `SwapBuffers`, you get a sense of how many other things you can ask the graphics program to do before the update rate starts conflicting with the refresh rate. We call this "headroom."

If the 2D flow field is time-varying, you can also use LIC to create a time animation. A newer feature of OpenGL is the ability to gang a collection of 2D textures into a single 2D texture array. This

Previous Work on Shaders in Visualization

Using shaders for visualization started with experiments using RenderMan.[1] Interactive (that is, graphics-hardware-based) GPU shaders appeared in the early 2000s.[2] Since then, researchers have pushed them into a variety of applications. Many of these have involved scientific and data visualization using 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. IEEE Visualization 1993*, IEEE CS 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*, 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 Level of Detail 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*, IEEE CS Press, 2003, pp. 171–176.
6. A. Lefohn et al., "Interactive Deformation and Visualization of Level Set Surfaces Using Graphics Hardware," *Proc. IEEE Visualization 2003*, IEEE CS Press, 2003, pp. 75–82
7. P. McCormick et al., "Scout: A Hardware-Accelerated System for Quantitatively Driven Visualization and Analysis," *Proc. IEEE Visualization 2004*, IEEE CS Press, 2004, pp. 171–179.

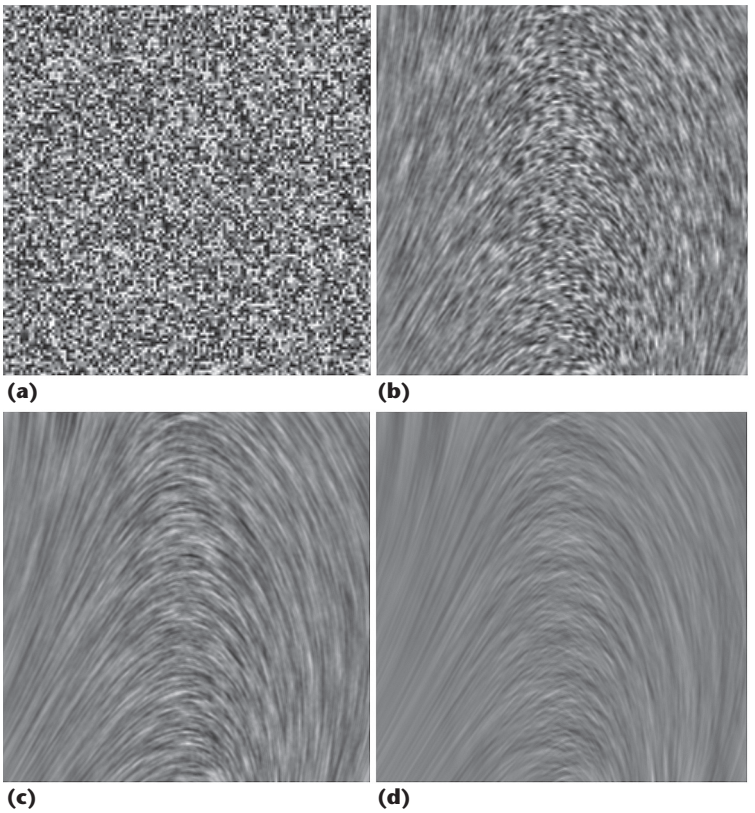


Figure 1. Line integral convolution (LIC) visualizes flow directions across a 2D field. For (a) an input image that's a white noise texture, the output consists of grayscale smears—here, for lengths of (b) 10, (c) 30, and (d) 100. The length indicates the number of samples taken in each direction along the flow lines.

array differs from a set of individual 2D textures in that it counts as a single texture. It wouldn't take many individual 2D textures in the set to overrun the maximum allowable active textures, so new textures would have to be constantly re-bound.

The array also differs from a 3D texture in that you can sample a specific slice without accidentally causing any interpolation with neighboring slices. To do this, you must make four changes. First, the flow field texture must be of type `GL_TEXTURE_2D_ARRAY` instead of `GL_TEXTURE_2D`. Second, in the fragment shader, you must declare the flow field texture sampler as `sampler2DArray` instead of `sampler2D`. Third, the application must pass a uniform variable indicating the slice number to the fragment shader. Finally, you must index the flow field with a `vec3` variable, just as if it were a 3D texture. The first two elements, `s` and `t`, would be in the range 0 to 1, as before. The third element, `p`, would be in the range 0 to `float(#slices - 1)`; that is, it would be a floating-point index specifying the slice number.

Why use the 2D texture array instead of just a 3D texture? They both can encapsulate multiple 2D images as a single active texture, decreasing the overhead of binding and rebinding many 2D textures. They both consume the same memory space. The big difference is that because a 2D texture array indexes the slices by slice number, not [0, 1], you don't get unwanted filtering between the slices.

You can also have fun with LIC outside the world of visualization. If you substitute a real image for the white noise image, interesting photographic effects result (see Figure 4). In this case, you could sculpt a flow field just to produce specific smearing effects in specific parts of the photograph. You can see this, for example, in sports photography.

```
#version 400
precision highp float;
precision highp int;

uniform int           uLength;
uniform sampler2D     uImageUnit;
uniform sampler2D     uFlowUnit;

in  vec2              vST;
out vec4              fFragColor;

void
main( void )
{
        ivec2 res=textureSize( uImageUnit, 0 );

        // flow field direction:

        vec2 st = vST;
        vec2 v = texture( uFlowUnit, st ).xy;
        v /= vec2(res);          // express velocities per-texel

        // starting location:

        st = vST;
        vec3 color = texture( uImageUnit, st ).rgb;
        int count = 1;

        st = vST;
        for( int i = 0; i < uLength; i++ )
        {
                st += v;
                vec3 new = texture( uImageUnit, st ).rgb;
                color += new;
                count++;
        }

        st = vST;
        for( int i = 0; i < uLength; i++ )
        {
                st -= v;
                vec3 new = texture( uImageUnit, st).rgb;
                color += new;
                count++;
        }

         color /= float(count);
         fFragColor = vec4( color, 1. );
}
```

Figure 2. The fragment shader. It samples the noise texture `uLength` times going forward and `uLength` times going backward.

```
#version 400 compatibility
precision highp float;
precision highp int;

out vec2 vST;

void
main( void )
{
                vST = gl_MultiTexCoord0.st;
                gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Figure 3. The vertex shader. This is called once for each of the four quadrilateral vertices. Its sole jobs are to produce the transformed vertex and arrange to have the texture coordinates interpolated through the rasterizer.

(a) (b)

Figure 4. Using LIC outside the world of visualization. (a) A real image. (b) LIC with that image. Unlike here, a purposely designed vector field can lead to a deliberate photographic effect.

Bump-Mapping Terrain Surfaces

Bump-mapping makes the display of an undetailed surface (such as a plane or a sphere) look like that of a very detailed surface. The idea is to create the same normal at each pixel on the undetailed surface that you would have gotten from the detailed surface, then use that normal in the lighting-model equations. You run into this in visualization a lot when you want to display data (such as weather, locations, geology, routes, or snow pack) on top of realistic-looking terrain that can be panned and zoomed, but you don't want to create the thousands (or millions) of triangles necessary to do it for real.

Figure 5 shows the fragment shader code, where all the work really happens, for terrain bump-mapping. The height values are stored in a 2D texture's red component, where they can be accessed quickly. (I wrote a tool to take a longitude-latitude range from the US Geological Survey National Elevation Dataset database [http://ned.usgs.gov], filter it to the right resolution, and convert it to a floating-point texture file.) At the current fragment location, the shader looks up the heights and constructs tangent vectors in `s` and `t`. You can exaggerate those heights if desired. Their cross product gives the surface normal there, which is then used in the lighting model, just as if the heights really were being displayed in those thousands of triangles.

Figure 6 shows this shader in action. I used a 4,096 × 2,276 texture. On the Nvidia GTX 480, performance was approximately 10,000 fps. (This number seems astonishingly high. It makes sense, though. Because the entire display geometry is a single quad, the graphics card can focus almost all its power on running the bump-mapping fragment shader, creating a lot of parallelism. The point is that this leaves you with a lot of performance headroom.) Figure 6b shows the heights exaggerated. Figure 6c shows colors mapped to height ranges, using the OpenGL Shading Language (GLSL) `smoothstep( )` function between levels to prevent abrupt color changes (which looks jarring—trust me).

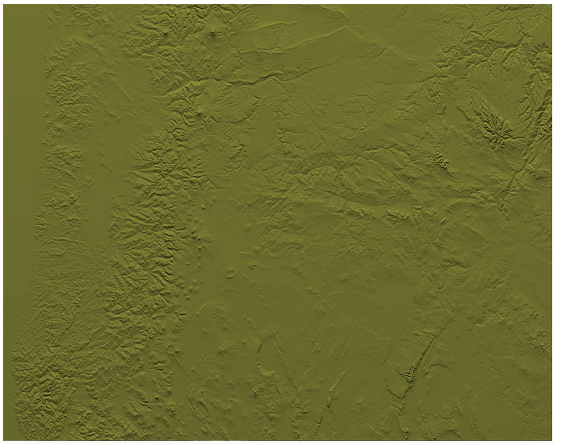
Textures are a good place to store height data. They can hold very large datasets, they can be sampled quickly and easily, and the sampler can linearly interpolate between samples. A nice bonus is that you can use mip-mapping. This gives a very smooth appearance when the scene is zoomed out and a very detailed appearance when the scene is zoomed in, with little or no performance penalty.

Bump-mapping is a neat visualization display trick. It makes it look like you have much more geometric detail than you actually do. But, it's still a trick. Don't change the eye position much, or it will be obvious that you've been cheating with underdetailed geometry. However, for applications in which you want to display data on top of realistic-looking birds-eye terrain data and be able to interactively pan and zoom, bump-mapping is effective.

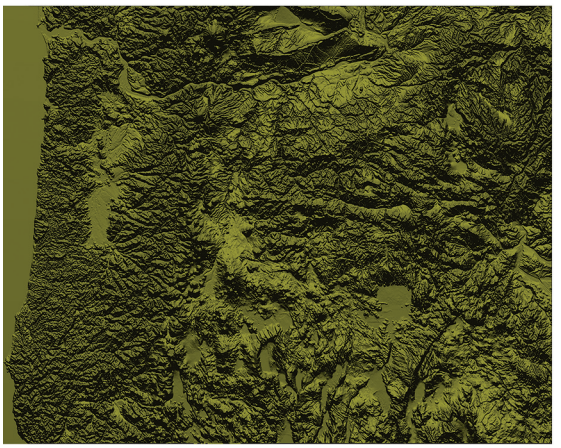
If you do feel like you need to add some real heights to your display, bump-mapping is still useful. Another trick is to use fewer points in the 3D geometric mesh than you have height values. The full-resolution bump-mapping will make the lower-resolution geometry look much better than you'd expect. You can even create triangle geometry that's adaptively based on screen extent, using the new OpenGL tessellation shaders, but that's a more involved topic for another time.

```
#version 400
precision highp float;
precision highp int;

uniform float       uLightX, uLightY, uLightZ;
                                // light position
uniform float       uExag;      // height exaggeration
uniform vec4        uColor;     // default color
uniform sampler2D   uHgtUnit;   // height texture unit

// parameters for setting the colors:
uniform bool        uUseColor;
uniform float       uLevel1;
uniform float       uLevel2;
uniform float       uTol;

in vec3             vMCposition;  // fragment's position
in vec2             vST;          // fragment's texture coords

out vec4            fFragColor;   // fragment color

const vec3 BLUE  = vec3( 0.1, 0.1, 0.5 );
const vec3 GREEN = vec3( 0.0, 0.8, 0.0 );
const vec3 BROWN = vec3( 0.6, 0.3, 0.1 );
const vec3 WHITE = vec3( 1.0, 1.0, 1.0 );

const float LNGMIN = -579240.;   // distance in meters
const float LNGMAX =  579240.;   // distance in meters
const float LATMIN = -419949.;   // distance in meters
const float LATMAX =  419949.;   // distance in meters

const float DELTA  = 0.001;
const float DS = 2. * DELTA * ( LNGMAX - LNGMIN );   // length of tangent vector in s
const float DT = 2. * DELTA * ( LATMAX - LATMIN );   // length of tangent vector in t

const float AMBIENT = 0.10;      // ambient intensity

void
main( void )
{
        vec2 s_step = vec2( DELTA, 0. );    // used to step in the height texture in s
        vec2 t_step = vec2( 0., DELTA );    // used to step in the height texture in t

        float west  = texture2D( uHgtUnit, vST-s_step ).r;  // height in meters
        float east  = texture2D( uHgtUnit, vST+s_step ).r;  // height in meters
        float south = texture2D( uHgtUnit, vST-t_step ).r;  // height in meters
        float north = texture2D( uHgtUnit, vST+t_step ).r;  // height in meters

        vec3 stangent = vec3( DS, 0., uExag * ( east  - west  ) );
        vec3 ttangent = vec3( 0., DT, uExag * ( north - south ) );
        vec3 normal   = normalize( cross( stangent, ttangent ) );

        float LightIntensity = dot( normalize(vec3(uLightX,uLightY,uLightZ) - vMCposition), normal );
        LightIntensity = clamp( LightIntensity + AMBIENT,  0., 1. );

         if( uUseColor )
         {
                float here = texture2D( uHgtUnit, vST ).r;
                vec3 color = BLUE;
                if( here > 0. )
                {
                        float t = smoothstep( uLevel1-uTol, uLevel1+uTol, here );
                        color = mix( GREEN, BROWN, t );
                }
                if( here > uLevel1+uTol )
                {
                        float t = smoothstep( uLevel2-uTol, uLevel2+uTol, here );
                        color = mix( BROWN, WHITE, t );
                }
                fFragColor = vec4( LightIntensity*color, 1. );
         }
         else
         {
                fFragColor = vec4( LightIntensity*uColor.rgb, 1. );
         }
}
```

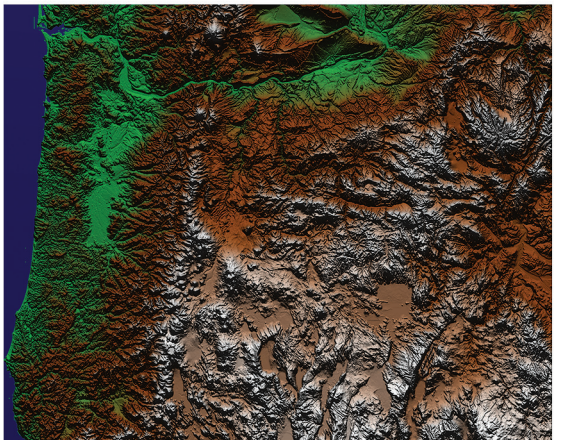
Figure 5. The fragment shader for terrain bump-mapping. It uses the heights from the height texture to compute a normal to use in the simple lighting model.



(a)



(b)



(c)

Figure 6. The shader in action. (a) Unexaggerated heights. (b) Exaggerated heights. (c) Colors assigned to height ranges. On an Nvidia GTX 480 card, performance was approximately 10,000 fps.

A Word on Recent Changes to OpenGL

Over the past year or two, OpenGL has undergone significant changes, particularly in the use of GLSL. First, some OpenGL and GLSL features have been deprecated. This means that they could eventually go away. Or, they might not. A lot of legacy code is out there, and it's doubtful that the driver-writing graphics vendors want to upset thousands of customers by breaking that code. However, it's good to be aware of what has been deprecated and perhaps stop using it in new applications.

In GLSL, the use of the variable qualifier `varying` has been deprecated. This is a good thing. The word "varying" made great sense when it described variables being output from the vertex shader, interpolated through the rasterizer (thus, the meaning of "varying"), and input to the fragment shader. However, these days, the next stage of the pipeline behind the vertex shader can be a tessellation control shader, a tessellation evaluation shader, a geometry shader, or a fragment shader. Only values passed into the fragment shader get rasterized, so calling the vertex shader outputs "varying" doesn't consistently make sense. Instead, the new preferred term is `out`, which is now used at all stages of pipeline shaders to indicate that a variable is being passed to the next stage, whatever that next stage is. Subsequently, that next stage must declare that same variable as `in`.

The shader code in Figures 2, 3, and 5 shows this convention in action. I like naming shader `out` and `in` variables with a word describing what they are, beginning with a capital letter (for example, `FragColor`). However, to keep the pipeline stages straight, I like to precede that word with lowercase letters indicating the variable's origin: `a` for attribute, `u` for uniform, `v` for vertex, `tc` for tessellation control, `te` for tessellation evaluation, and `f` for fragment (for example, `fFragColor`). (This makes the variable names look a little strange but helps me keep track of variables through multiple pipeline stages.) So, the interstage variable `vST` is set by the vertex shader, and the variable `fFragColor` is set by the fragment shader.

If you're used to the texture-sampling function `texture2D` and all its separately named derivatives, you'll enjoy that you can now use a function simply named `texture`. This function is overloaded, and its exact meaning depends on the type of sampler variable that's passed to it.

You'll also notice a line at the top defining the GLSL version that this code follows. By defining the version as `400`, you'll have all the functionality in GLSL 4.00 available. If you place nothing after `400`, this implies that you want to use the core functionality, and the compiler will whine if you use any deprecated features. If, instead, you add `compatibility` to the `#version` line, the compiler will let you get away with using all the deprecated features too. This way, you get to use everything. However, you can only put a version number on the `#version` line up to and including what your graphics card and driver can support.

The vertex shader in Figure 3 uses the deprecated built-in variables `gl_MultiTexCoord0` and `gl_ModelViewProjectionMatrix`, indicating that it wants to be compiled in compatibility mode. The fragment shaders in Figures 2 and 5, however, don't use any deprecated features, so they don't need to specify compatibility mode on the `#version` line, although that wouldn't hurt.

These examples conform to GLSL version 4.00. To make them work on pre-4.00 systems, you would have to change certain things (such as using `gl_FragColor` instead of `fFragColor`).

The second and third lines in Figures 2, 3, and 5,

```
precision highp float;
precision highp int;
```

are there for OpenGL-ES compatibility. OpenGL-ES, the OpenGL API for embedded systems (that is, mobile devices and so on) wants the programmer to specify at what precision it should do variable arithmetic. This is so that it can save power if possible. You can specify the precision variable-by-variable or, as shown here, specify the precision for all variables of a particular type.

OpenGL-desktop ignores these precision commands, but I've started adding them to all my new shader code anyway. Even though I'm not yet actively using OpenGL-ES (dabbling, yes; seriously using, no), I expect that soon I will be. When that happens, I would like all my shader code to be compatible.

Your attitude toward OpenGL-ES migration should greatly influence how you use OpenGL-desktop now and how quickly you abandon deprecated functionality. All educators who teach OpenGL-desktop need to think about this. If you're just teaching "graphics thinking," consider sticking with compatibility mode. It will be around for a long time—maybe forever. It makes learning OpenGL programming easy. If you're teaching serious graphics application development, you probably want to start migrating away from deprecated features and evolving toward OpenGL-ES compatibility.

Acknowledgments

Many thanks to Nvidia and Intel for their continued support in shaders and visualization.

References

1. M. Bailey, "Using GPU Shaders for Visualization," *IEEE Computer Graphics and Applications*, vol. 29, no. 5, 2009, pp. 96–100.
2. B. Cabral and L. Leedom, "Imaging Vector Fields Using Line Integral Convolution," *ACM Trans. Graphics*, vol. 22, no. 3, 2003, pp. 263–270.

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 theresamarierhyne@gmail.com.