Teaching OpenGL Shaders: Hands-on, Interactive, and Immediate Feedback

Mike Bailey
Oregon State University, USA, mjb@cs.oregonstate.edu

Abstract

This paper describes the teaching of OpenGL shaders with hands-on a program called *glman*. Hands-on education is at its best when the students' experimental feedback loop is very fast. *glman* allows students to create a shader scene description file which not only creates the 3D scene, but creates an interactive user interface to adjust parameters. Our experience in an experimental class taught in Spring 2006 is that glman is flexible enough to demonstrate and experiment with many shader concepts, and creates a fast learning curve for the students.

Categories: 1.3 [Computer Graphics], 1.3.1 [Graphics Processors], K.3.1 [Computer Uses in Education]

1. Introduction

GPU-programmable shaders are the most exciting development in computer graphics in a long time. For the first time, programmers can get both the flexibility to perform amazing vertex-by-vertex and pixel-by-pixel effects, combined with the performance to make it interactive. The emergence of shader programming will have profound effects on all areas of computer graphics including science, engineering, art, animation, and gaming. This is the good news. The bad news is that shaders are difficult to learn and teach. The effects of certain shader parameters in certain shader equations are not obvious. Converging on good values is difficult.

glman is a new program that was written to help teach the OpenGL Shading Language (GLSL) [FER04, PF05, ROS06]. It uses an input file called GLIB, (GL Interface Bytestream), which is modeled after the style of the RenderMan Interface Bytestream (RIB) [UPS90, AG99]. *glman* reads a GLIB file as well as one or more vertex and fragment shader files. It then creates the requested scene, activates the requested shaders, and creates sliders for user-defined global parameters. *glman* also provides a Perlin noise [PER85,PER02] 3D texture for use in the shader. Our experience with using *glman* in a college class is that students get a maximum amount of quality learning in the minimum amount of time.

2. Shaders in the Graphics Pipeline

Figure 1 shows a generic view of the computer graphics rendering process. There are two locations in this process into which an application developer can inject custom shader code: the vertex processing and the fragment processing. The Vertex Processor (VP) takes 3D coordinates in the modeling coordinate space. It transforms them into world coordinates using a modeling transform, then transforms them into eye-space coordinates using a viewing transform. It then performs clipping, projective transformation, and viewport mapping. When coordinates leave the vertex processing stage, they have been changed into screen-space coordinates, ready to be rasterized. The reason that the VP is a great location to place custom code is that there is considerable information about the geometry available at that point, and the VP can do a variety of things with it.

The second location is the Fragment Processor (FP). Because the output of the rasterizer is already an interpolated red, green, blue color, students are greatly confused about the function of the FP. The inputs to the FP are every piece of information that is currently available about this pixel. The most important pieces of information include the pixel's previously-assigned red, green, and blue color; its alpha (transparency) value; its texture coordinates; plus any information passed from the Vertex Processor and interpolated in the rasterizer such as the pixel's x, y, and z location and its surface normal. The FP also has access to any global information passed by the application program such as light positions. The Fragment Processor's job is to take all this information and produce the final red, green, blue, and alpha for that pixel. It also has the option to completely discard this pixel. The reason that the FP is a great place to write custom code is that the appearance of that pixel can be computed based on whatever mathematics, optics, physics, or whimsy one wants to program.

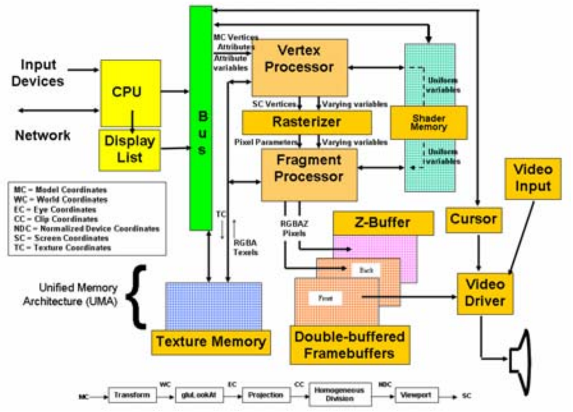


Figure 1: *Generic Computer Graphics Process*

3. The Course

This course, CS 519, is a multidisciplinary course, with students from Computer Science, Engineering, and Geosciences. The course teaches the theory behind how shaders work, enough graphics software and hardware to understand what is happening behind-the-scenes, the mathematics of shader effects, and shows their use in a variety of applications.

The assignments consist of several shader-creation projects which solidify the students' understanding of various shader programming and mathematics concepts. The class culminates in a Final Project, the *Shader Olympics*, in which each student chooses their own area of interest and develops a shader-based application in that area.

The class lectures are in a hands-on lab. Thus, it was important to be able to provide some sort of environment where the students could run instructor-provided examples, discover the effects of certain key parameters, and then quickly change the examples to perform new tasks.

4. What Do Shaders Look Like?

The following code shows a vertex shader example, one of the first given to the students. This vertex shader computes diffuse light source shading based on the transformed surface normal. It sets up variables **Color**, **X**, and **LightIntensity** to be interpolated by the rasterizer into each instance of the fragment shader. It also multiplies this model-space coordinate by the full Model-View-Projection matrix and passes into the rest of the pipeline.

```
varying float LightIntensity;
varying vec4 Color;
varying float X;

void
main( void )
{
   vec3 tnorm = normalize( gl_NormalMatrix * gl_Normal );
   vec3 LightPos = vec3( 0., 5., 10. );
   vec3  ECposition  =  vec3(  gl_ModelViewMatrix  *
gl_Vertex );
   LightIntensity  = dot( normalize(LightPos - ECposition),
tnorm );
   LightIntensity = abs( LightIntensity );
   Color = gl_Color;
   X = gl_Vertex.x;
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

The following shows the corresponding fragment shader code. Uniform variables are passed in by the application. The fragment shader then uses the varying and uniform variables to decide if this fragment is in a stripe or not. It uses two instances of the GLSL-provided **smoothstep** function to create a "smooth pulse" so that the edges of the stripe are blended rather than being blatantly aliased. It then passes this procedurally-determined color into the rest of the pipeline.

```
varying float X;
varying vec4 Color;
varying float LightIntensity;
uniform float A, P, Tol;

void
main( void )
{
   vec4 WHITE = vec4( 1., 1., 1., 1. );
   float f = fract( A*X );

   float t = smoothstep( 0.5-P-Tol, 0.5-P+Tol, f )  -
            smoothstep( 0.5+P-Tol, 0.5+P+Tol, f );
   gl_FragColor = mix( WHITE, Color, t );
   gl_FragColor.rgb *= LightIntensity;
}
```

Figure 2 shows what this shader combination produced when displaying a particular scene.

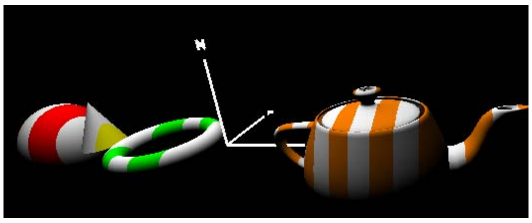
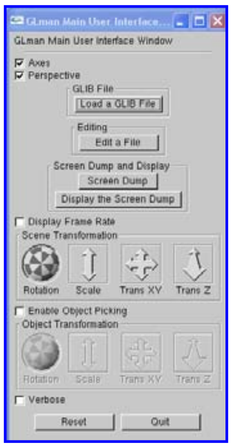


Figure 2: *Procedural Stripes Computed in Model Coordinates*

5. Introducing Shaders to Students



But, our experience is that students learn shaders very slowly if they must go through the full edit-compile-execute sequence for every little feature they want to try. We believe that learning shaders works best when the students are in a very tight try-it-myself loop. With that in mind, we created a program called *glman*. The *glman* user interface is shown in here.

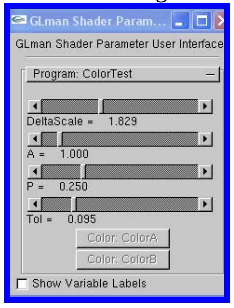
glman is so named because its input looks a lot like the RIB files of RenderMan. As such, its input files are called GLIB files, for GL Interface Bytestream. The .glib file that produced Figure 2 is shown here:

```
    Perspective 90
    Translate -5 0 0
    Vertex stripesMC.vert
    Fragment stripesMC.frag
    Program StripesMC A <0 1. 10>  P <0. .25 1.>  \
            Tol <0. 0. .5>
    Color [1 0 0]
    Sphere 1
    Color [1 1 0]
    Translate 1.5 0 0
    Cone 0.5 1.
    Color [0 1 0]
    Translate 2 0 0
    Torus .2 1.
    Color [1 0.5 0]
    Translate 4 0 0
    Teapot
```

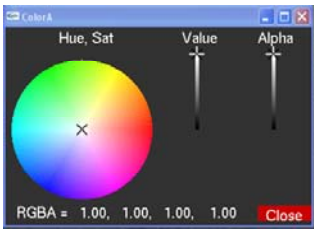
The lines:

```
Vertex stripesMC.vert
Fragment stripesMC.frag
Program StripesMC A <0 1. 10>  P <0. .25 1.>  \
          Tol <0. 0. .5>
```

are the most interesting. The first line causes the file stripesMC.vert to be read and compiled as a vertex shader. The second line does the same for the fragment shader file stripesMC.frag. The third line links the current vertex and fragment shaders into a single shader program, which will then be applied to subsequent geometry. That line also creates three uniform global variables **A**, **P**, and **Tol**, and puts them on sliders for the student to change interactively, as shown here. The values in the angle brackets are the minimum value on the slider, the initial value, and the maximum value. Uniform variables that represent colors are enclosed in curly brackets. They are {red green blue [alpha]} and will generate a button in the UI that, when clicked, brings up a color selector as shown here. The color selector allows the user to change this color variable on-the-fly.



Program: ColorTest
DeltaScale = 1.829
A = 1.000
P = 0.250
Tol = 0.095



RGBA = 1.00, 1.00, 1.00, 1.00

Multiple vertex-fragment-program combinations are allowed in the same GLIB file. If there is more than one combination, they will appear as separate rollout panels in the user interface

In this way, *glman* allows a student to create a scene, a vertex shader, and a fragment shader, and interactively test the effects of many different parameter combinations in minutes, rather than hours.

6. Teaching Noise as an Integral Part of Shaders

Noise is a major component of shader-writing. Originally developed by Ken Perlin [PER85, PER02], noise is used as a variation on surface properties to make the surfaces more interesting. But, noise is a difficult concept to explain to students. So, we have written another program, *NoiseGraph*, to give the students hands-on experience with creating and controlling noise functions.

The following figures show three scenarios from *NoiseGraph*. The first figure shows positional noise, that is, random values are chosen at integer intervals and a smooth function is fit through them.

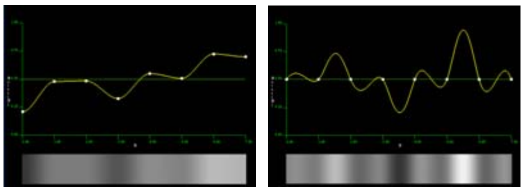


Figure 3: **(***a) Positional Noise, (b) Gradient Noise*

Figure 3a shows why *positional noise* is not used in shaders. Based on random chance, there is a good probability that the values at the integer positions will not be very evenly distributed. Figure 3b shows *gradient noise*, in which the integer points are forced to have values midway through the range with the slopes at those points chosen at random. As can be seen, the distribution is much more uniform, without being "too uniform".

The noise function is made more interesting by using it to create fractional Brownian motion (fBm), or 1/f behavior. In this method, multiple noise functions are summed. Each successive noise function is twice the frequency and half the amplitude of the previous one. This makes the noise more interesting. The low frequency functions give it definition, and the high frequency functions give it character. The students can interactively experiment with this too, to experience the effects of the different noise parameters for themselves. Figure 4 shows four octaves of 1/f noise.

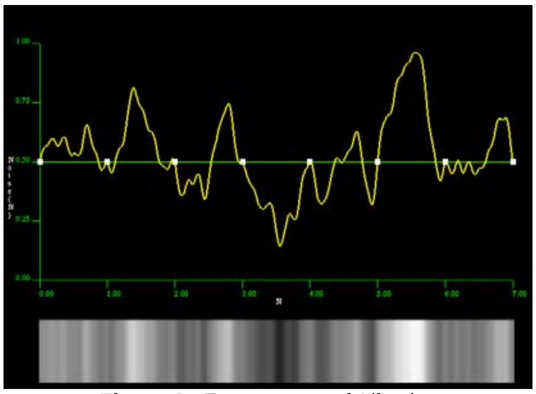


Figure 4: *Four octaves of 1/f noise*

Figure 5 shows an interesting noise example from class: the teapot with a rainbow shader. The rainbow shader uses the Model Coordinate position of each fragment to assign the colors. In the middle image, a *pulse* function is used to transition between the colors. The width of the smooth step is controlled by a slider. In the third image, a noise function is added to the coordinate position. The magnitude of the noise is also controlled by a slider.

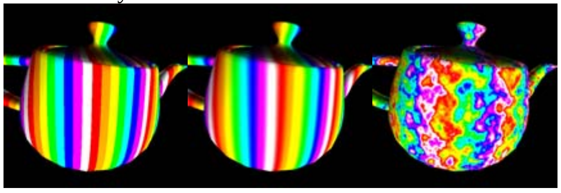


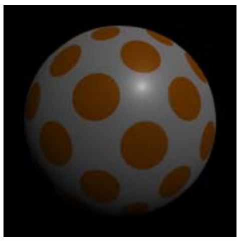
Figure 5: *Rainbow stripe shader with noise*

This is a very simple example, but it readily explains to students changing display parameters based on coordinate position, the *smoothstep* function, and noise.

It also lets them quickly try it themselves with their own functional parameters, instead of just being shown

7. Assignments and Examples

The following show some of the assignments used in the class:



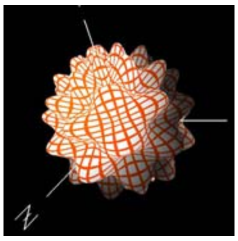
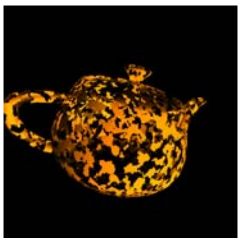


Figure 6: *Dots, based on texture space coordinates in the fragment shader*
Figure 7: *Surface displacement in the vertex shader and gridline assignment in the fragment shader*



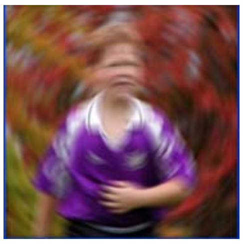
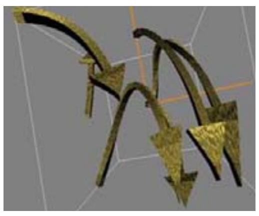


Figure 8: *Noise-based erosion shader, using texture-space coordinates in the fragment shader*
Figure 9: *Interactive Line Integral Convolution using texture manipulation in the fragment shader*



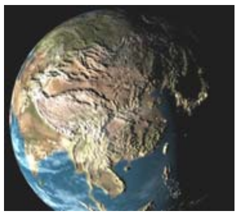


Figure 10: *Flow visualization object extrusion in the vertex shader*
Figure 11: *Terrain visualization bump-in the fragment shader*

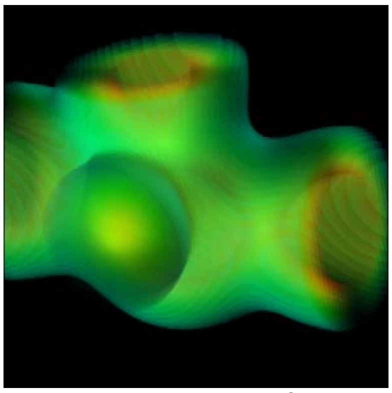


Figure 12: *Volume visualization in the fragment shader*

8. Conclusions

The combination of *glman* and *NoiseGraph* has been used in our college class to teach GLSL shaders. We have found them to be excellent tools to explain how certain shader parameters work and to let students quickly explore on their own. Because students don't need to write full programs, and because *glman* creates a user interface from user directives, it is fast and easy to get started, and encourages individual exploration. Because the uniform variables can so readily be manipulated, it is easy to create sophisticated shaders and determine what variables should be used and how they should be set.

The class syllabus is located at:

```
http://eecs.oregonstate.edu/~mjb/cs519
```

The *glman* and *NoiseGraph* programs and documentation can be obtained at:

```
http://eecs.oregonstate.edu/~mjb/glman
```

References

[AG99] Tony Apodaca and Larry Gritz, *Advanced RenderMan: Creating CGI for Motion Pictures*, Morgan Kaufmann, 1999.

[FER04] Randima Fernando, *GPU Gems*, NVIDIA, 2004.

[PER85] Perlin, K., An Image Synthesizer, *Proc. ACM SIGGRAPH '85*, Vol. 19, No.3, July, 1985, pp. 287-296.

[PER02] Perlin, K., Improving Noise, *Proc. ACM SIGGRAPH '02*, Vol. 21, No.3, July, 2002, pp. 681-682.

[PF05] Matt Pharr, Randima Fernando, *GPU Gems 2*, NVIDIA, 2005.

[ROS06] Randi Rost, *OpenGL Shading Language*, Addison-Wesley, 2006.

[UPS90] Steve Upstill, *The RenderMan Companion*, Addison-Wesley, 1990.