A Hands-on Environment for Teaching GPU Programming

Mike Bailey
Oregon State University
2117 Kelley Engineering Center
Corvallis, OR 97331-5501
+1.541.737.2542
mjb@cs.oregonstate.edu

Steve Cunningham
Grinnell College
715 Mesquite Drive
Coralville, IA 52241
+1.319.351.6800
cunningham@siggraph.org

ABSTRACT
GPU programming is fast becoming an essential skill for computer graphics students. It has immediate application in all areas of graphics including science, engineering, art, animation, and gaming. Because it is new, experience with teaching GPU programming is scarce. This paper describes the teaching of a GPU programming course with a hands-on program called *glman*. *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 shader 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 and fun learning curve for the students.

Categories and Subject Descriptors
1.3 [Computer Graphics], 1.3.1 [Graphics Processors], K.3.1 [Computer Uses in Education]

Keywords
Computer graphics, GPU, game development, graphics shaders, visualization

1. INTRODUCTION
GPU-programmable shaders are the most exciting development in computer graphics in a long time. Using shaders, programmers have the flexibility to perform amazing vertex-by-vertex and pixel-by-pixel effects, combined with the parallel-processor performance to use shaders in interactive graphics. The emergence of shader programming is having profound effects on all areas of computer graphics including science, engineering, art, animation, and gaming. But because GPU programming is fairly new, and because specialized hardware is needed to use it, teaching experience with this topic is scarce. Also, the mathematics of shader effects are such that the consequences of changing certain shader parameter values are not obvious. Converging on good values is difficult.

SIGCSE '07, March 7–11, 2007, Convington, Kentucky, USA.

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

The assignments consisted of several shader-creation projects which solidified the students' understanding of various shader programming and mathematics concepts. The class culminated in a final project, the *Shader Olympics*, in which each student chose their own area of interest and developed a shader-based application in that area.

The class lectures were in a hands-on lab. We took advantage of this by dividing the class into teams of 2-3 students each and starting most classes with a "Team Challenge" – a small assignment that each team worked on together. The goal was to have the students help each other with key concepts. We also took advantage of the hands-on lab by having the students run and modify live examples during the class to reinforce lecture topics.

With these pedagogic objectives in mind, 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. The answer was *glman*.

glman is a program that was written to help teach the OpenGL Shading Language (GLSL) [FER04, PF05, ROS06]. It uses an input file type 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 shaders. Our experience with using *glman* in an Oregon State University college class is that students get a maximum amount of quality learning in a minimum amount of time.

3. THE GRAPHICS PIPELINE AND GPU PROGRAMMING
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 GPU 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 projective transformation and normalized device coordinate mapping. When coordinates leave the vertex processing stage, they are then clipped and mapped 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.

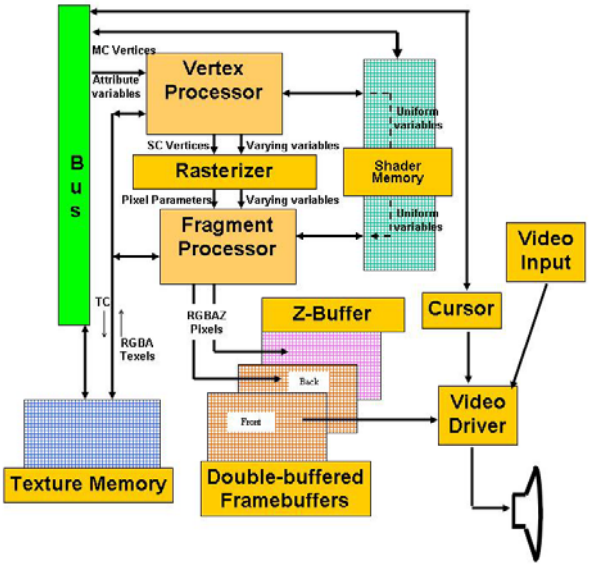
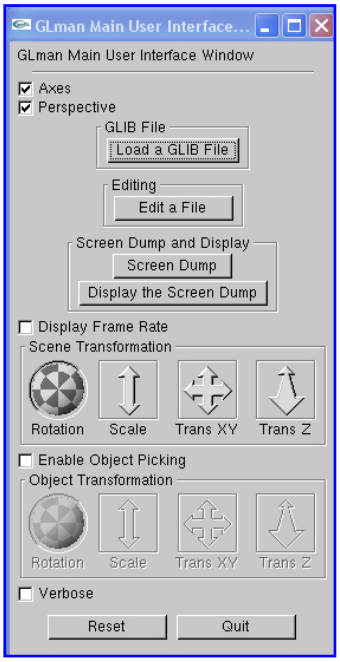


Figure 1: Generic Computer Graphics Process

The second location is the Fragment Processor (FP). Because the output of the rasterizer is already an interpolated red, green, blue color, students are usually 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. The pixel also has 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.



4. INTRODUCING SHADERS TO STUDENTS

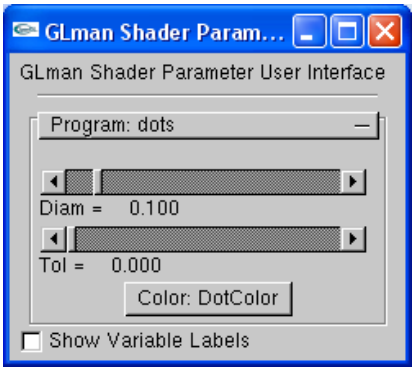
Our experience is that students learn shaders very slowly if they must go through the full edit-compile-execute sequence for every feature they want to try. We believe that learning shaders works best when the students are in a very tight try-it-myself loop. The *glman* user interface is shown here, and the *glman* tool was developed to give students the chance to have this experience.

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 (below) is shown here:

```
Perspective 90
Translate -2 0 0
Vertex dots.vert
Fragment dots.frag
Program dots Diam <0 0.1 1.0> Tol <0. 0. .005> \
      DotColor {1.,1.,1.}
Color [1 0 0]
Sphere 1
Color [1 0.5 0]
Translate 4 0 0
Teapot
```

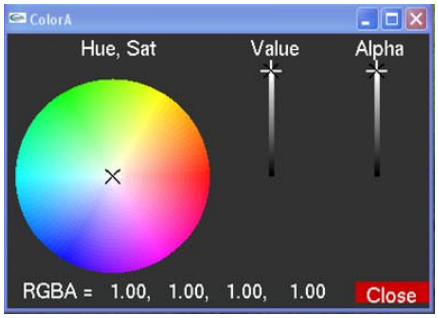
The lines:

```
Vertex dots.vert
Fragment dots.frag
Program dots Diam <0 0.1 1.0> Tol <0. 0. .005> \
      DotColor {1.,1.,1.}
```



are the most interesting. The first line causes the file dots.vert to be read and compiled as a vertex shader. The second line does the same for the fragment shader file dots.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 two uniform global variables **Diam** and **Tol**, and puts them in a rollout panel on sliders for the student to change interactively, as shown here.

The values in the GLIB 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.

Finally, the lines:

```
Color [1 0 0]
Sphere 1
Color [1 0.5 0]
Translate 4 0 0
Teapot
```

define the scene model that the shaders act on: in this case, a red sphere and an orange teapot. These objects are shown in Figure 2 with the shading applied.

The following code shows the dots vertex shader in action -- it computes diffuse light source shading based on the transformed surface normal. It sets up the variables **Color** 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 it into the rest of the pipeline.

```
varying vec4 Color;
varying float LightIntensity;

void
main( void )
{
 const vec3 LightPos = vec3( 0., 0., 10. );

 vec3 tnorm = normalize( gl_NormalMatrix * gl_Normal );
 vec3 ECposition = vec3( gl_ModelViewMatrix * gl_Vertex );
 LightIntensity = dot( normalize(LightPos - ECposition), tnorm );
 LightIntensity = abs( LightIntensity );

 Color = gl_Color;
 gl_TexCoord[0] = gl_MultiTexCoord0;
 gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

The following code shows the corresponding fragment shader, using the global variable values from the sliders. The fragment shader uses the varying and uniform variables to decide if this fragment is in a dot or not. It uses the GLSL-provided **smoothstep** function to create a blurred transition from the object's real color to the dot color so that the edges of the dot are blended rather than being blatantly aliased. It then passes this procedurally-determined color into the rest of the pipeline.

```
varying vec4 Color;
varying float LightIntensity;

uniform float Diam;
uniform float Tol;
uniform vec4 DotColor;

void
main( void )
{
 float sp = 2. * gl_TexCoord[0].s;
 float tp = gl_TexCoord[0].t;
 float numins = int( sp / Diam );
 float numint = int( tp / Diam );

 gl_FragColor = Color;
 if( mod( numins+numint, 2. ) == 0. )
 {
 sp = sp - numins*Diam;
 tp = tp - numint*Diam;
 float radius = Diam/2.;
 vec3 sptp = vec3( sp, tp, 0. );
 vec3 cntr = vec3( radius, radius, 0. );
 float d = distance( sptp, cntr );
 float t = smoothstep( radius-Tol, radius+Tol, d );
 gl_FragColor = mix( DotColor, Color, t );
 }

 gl_FragColor.rgb *= LightIntensity;
}
```

Figure 2 shows what this shader combination produced:

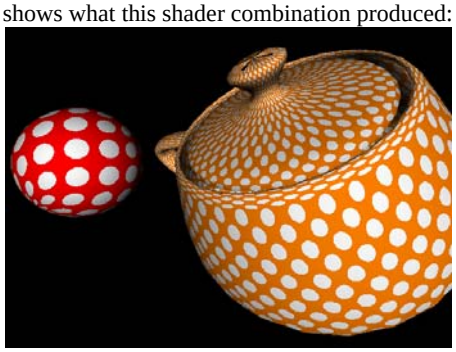


Figure 2: Procedural Dots Computed in Model Coordinates

Figure 3 shows the effect of blurring the dot transition with the **Tol** slider.

Figure 3: Procedural Color Blurring (left:Tol=0., right: Tol = .005)

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, vertex shaders and fragment shaders, and interactively test the effects of many different parameter combinations in seconds, rather than minutes or hours.

5. ASSIGNMENTS AND EXAMPLES

The following figures show some of the assignments and examples used in the class. Figure 4 uses a sine-sine function to displace the vertices of a sphere. Figure 5 uses a noise function to decimate geometry. The fragment shader uses the *discard* keyword to force certain pixels to not be displayed. Figure 6 shows "Toon Rendering", a fragment program which combines color quantization with an edge detection (programmed by performing a Sobel convolution with neighboring pixels in the image). Figure 7 shows a Line Integral Convolution programmed in the fragment shader very much like the edge detection, but by examining and blending pixel colors along flow lines. Figure 8 uses the vertex shader to displace the vertices of a 3D object in such a way that they follow a flow streamline and perform a time-based peristaltic motion to show flow speed. Figure 9 shows bump-mapping performed in the

fragment shader to give the appearance of 3D terrain surfaces. Figure 10 shows the use of the fragment shader to composite 3D volume slices to give a visualization volume rendering.

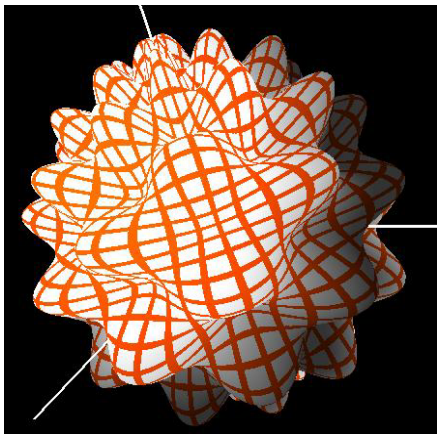


Figure 4: Surface displacement in the vertex shader and gridline assignment in the fragment shader

Figure 5: Noise-based erosion shader, using texture-space coordinates in the fragment shader

Figure 6: Interactive "Toon Rendering"

Figure 7: Interactive Line Integral Convolution

Figure 8: Flow visualization peristaltic object extrusion

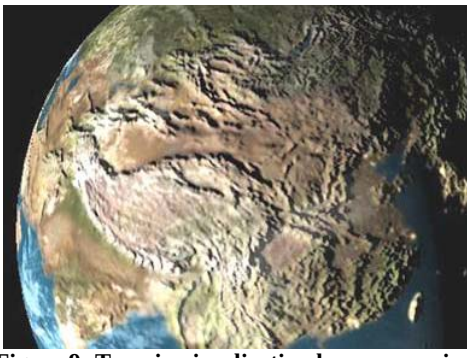


Figure 9: Terrain visualization bump-mapping

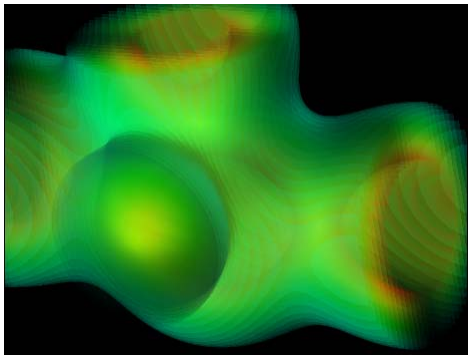


Figure 10: Volume visualization in the fragment shader

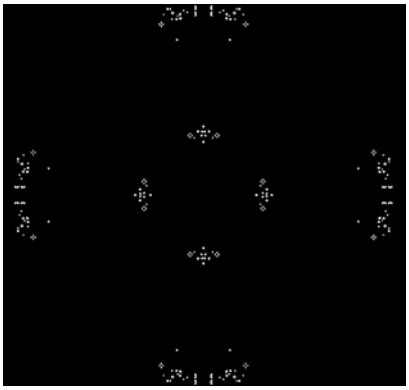


Figure 11: A Game-of-Life Parallel Fragment Program

6. FUTURE TOPICS: GEOMETRY SHADERS AND GPGPU

When the class is offered next year, we will incorporate two new topics: Geometry Shaders and General-Purpose GPU (GPGPU). The newly-announced Geometry Shader capability adds a user-programmable geometry-creation step into the graphics pipeline after the Vertex Shader. This allows a user to automatically tessellate complex geometry, or perform various geometry-dependent graphics operations such as silhouettes, sprites, and shadows.

In GPGPU ([GPGPU06]), the GPU is used for general equation-solving, taking advantage of its considerable parallel programming capability.[1] By treating a 2D array of data as a 2D image of pixels, equations can be solved by generating a large number of fragments, typically by drawing a single large quadrilateral. An example of this is using the GPU to solve John Conway's equations for the Game of Life [CONW70]. An output from this application is shown below. The speed of the GPU is apparent in that this application is able to solve these equations at a rate of 300 million computed-pixels per second, almost too fast to see.

7. CONCLUSIONS

glman has been used in our university class to teach GPU programming. We have found it to be an excellent tool to explain how certain shader parameters work and to let students quickly explore on their own. It also nicely supports our in-class hands-on examples and Team Challenges. 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 be so readily 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* program and documentation can be obtained at:

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

[1] For example, the new NVIDIA 8800 has up to 128 parallel vertex/fragment processors.

8. ACKNOWLEDGEMENTS

The base funding for the Computer Graphics Education Lab, in which this course is taught, came from Oregon State University's internal Technology Resource Fee fund. Our thanks to NVIDIA for their help with providing high-end graphics for this lab.

9. REFERENCES

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

[CONW70] John Conway, in "Mathematical Games", *Scientific American*, October 1970.

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

[GPGPU06] http://www.gpgpu.org

[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 and 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.