COMPUTER GRAPHICS: HOMEWORK #6
DUE 10/12/08

1. An Introduction to the Programmable GPU.

The programmable GPU is a very powerful and versatile tool for creating realistic graphical environments in realtime. In this assignment, you will be upgrading the program you wrote in HW5 by adding a simple shader program to perform your lighting calculations on the GPU. You'll again use Hermite splines to control a bump map, but, rather than computing the lighting equations entirely on the CPU for each pixel – which is very slow – your shader program will allow your lighting transforms to run in (pseudo)parallel on the GPU for many pixels at once.

1.1. **Display Setup and Rendering.** The layout of your display window is the same as in HW5; it should again be divided into 2 viewports. In the upper viewport, display the image plate `pic.jpg`. We denote by $II[u, v]$ the $[u, v]$ pixel of the Input image. We assume that the width of this image is 512 pixels. Display the image using an orthographic projection, where the image is in the plane z=-10, and assume that there is a source of light white at $(0, 512, 0)$. The function $F[u, v]$, whose details are described below, defines a displacement of the normal at each pixel for shading.

Implement a shader program to perform lighting. Use the standard Phong illumination model to calculate the image intensity at each image pixel location. As in HW5, use the normals computed from the bump map as discussed below. Implement the full Phong lighting model as discussed in class, including the specular, diffuse, and ambient terms (if you are having difficulty remembering the lighting equation, refer to the lecture slides and HW2). The intensity of the image pixels should vary according to the displaced normals given by F and the direction from that pixel to the light source. You can assume the viewing position is fixed at (0,0,0).

Additionally, allow the user to *move the light position* incrementally in the plane $z = 0$ by pressing the 'a', 'd', 'w' and 's' keys, which are mapped to left, right, up, and down movements, respectively. Output the current light position to the console after each keystroke.

The lower viewport of the display window again shows the curve C, through which the user can control C. There is no requirement to implement a GPU program to render this lower 2D viewport (although it is possible). For this viewport, you can just disable your GPU program and use the default GL rendering capabilities as before (see shader section below for details).

1.2. **A Shader Program for Lighting.** Recall that a valid shader program requires at least two parts: a vertex shader, which is called on every GL vertex, and a fragment shader, which operates on every interpolated pixel. You may choose *either* type of shader to render your lit, bump-mapped image. *Extra credit:* For modest extra credit, implement both approaches and comment on the differences between them.

Hints on building your shader:

- The type of shader you choose to implement, i.e. vertex or fragment, will dictate the approach you use to model and render your Input image. For example, if you choose to write a vertex program for lighting, you'll need to treat each pixel in your image as a vertex in order to maintain fidelity; in essence, your image becomes a collection of vertices. Conversely, if you implement lighting via a fragment shader, you might represent your image as a single quadrilateral (i.e. GL_QUAD) with a texture – the image pixels – "painted" onto it. There are pros and cons to each approach, such as a modest difference in implementation complexity and overall performance (framerate). Pick the approach with which you feel most comfortable; we'll accept either.
- A valid shader program requires function definitions for both the vertex and fragment shader components. Thus one of your shaders will likely serve as a pass-through "dummy" that doesn't do much. Consult the slides for details of setting up a pass-through for each shader type.
- You are not required to use a custom shader for rendering the 2D spline viewport. For this viewport, you can default back to the standard OpenGL rendering. Fortunately, GL provides an easy way to do this via a call to $glUseProgram(GLuint\ handle)$. To use your custom shader program, input the handle received during initialization (returned from $glCreateProgram$); to use the default GL rendering, simply pass zero (0) to $glUseProgram$.
- Your shader program "lives" on the GPU and has no direct access to the variables in main memory. Additionally, the functions in your C program *cannot* be called from within your GPU program. Thus specific channels of communication are needed to transmit values to the shader from your C program. Data should be communicated between your C program and shader using either built-in system globals or user-defined globals that you declare at the top of your shaders. When using custom-defined globals, you should take care to assign the proper qualifiers (i.e. $varying$, $attribute$, or $uniform$) and to use the correct functions to access them from within your C program (see the $glGet*Location$ functions for details). You'll find more information on data communication in the lecture slides as well as in the tutorials suggested on the webpage.
- A simple demo program has been provided on the class webpage. It includes the skeleton of a working shader program.

1.3. **The Curve $C(u)$ and the Function $F(u,v)$.** As in HW5, the curve C is controlled by the user and is the concatenation of 4 curves $C_1(u)\ldots C_4(u)$. They are determined via cubic splines (as defined in the slides). This curve C is the concatenation of 4 cubics C_1, C_2, C_3, C_4, where

- $C_1(0) = P_0,\ \ C_4(511) = P_4,$
- $C_1'(0) = C_1''(0) = 0$
- $C_1(127) = C_2(127) = P_1$; $C_2(255) = C_3(255) = P_2$; $C_3(384) = C_4(384) = P_3$
- $C_1'(127) = C_2'(127)$; $C_2'(255) = C_3'(255)$; $C_3'(384) = C_4'(384)$;
- $C_1''(127) = C_2''(127)$; $C_2''(255) = C_3''(255)$; $C_3''(384) = C_4''(384)$;

To compute this curve, you should use Hermite bases. The spline is specified by 5 control points $(P_1 \ldots P_5)$. Initially $P_i = ((i-1)128, 0)$, but the user can increase/decrease its y-value using the up and down arrows. So C is always smooth (that is, $C'(u)$ is continuous), and in addition, $C_1''(0)$ always equals 0.

The user should be allowed to select an individual control point for manipulation by using the left and right arrows to move along the splines. On each press of the $*$ key, the value of active control point $P_i = (u_i, v_i)$ is replaced by $(u_i, 2v_i)$. This allows the spline curvature to be exaggerated very quickly. Hitting the 'ESC' button resets each P_i to its original value.

Extra credit: If you're tired of Hermite bases and are feeling ambitious, you might choose to implement one of the other popular cubic spline forms, such as Bezier or Catmull-Rom, for modest extra credit. (Just make sure to explain what you did and the constraints you used in your readme.)

1.4. **The Bump Map.** As in HW5, the idea is to specify the normal to the plate as follows: We define the function $F(u,v) = (C(u), 0)$, and the normal $F(u,v)$ is approximated as the normal to plane passing through the points

$$(u, v, F(u,v)),\ (u+1, v, F(u+1, v)) \text{ and } (u+1, v+1, F(u+1, v+1))\ .$$

You will need a formula here to help you compute the normal from this data, and don't forget that the length of the normal vector is always 1.