CIS565 Spring 2011 Assignment 2

Due Wednesday, 02/02 at 11:59pm via Blackboard

This assignment gets you up and programming shaders. It introduces you to GLSL. If you find yourself doing lots of work with C++ and OpenGL, you are on the wrong track; please check in with us.

The assignment includes both programming and written components, make sure to complete them all. Include a readme.txt file.

Part 1: Vertex Wave (25%)

Coding Deliverable:
Sinusoidal wave using a vertex shader based on the Grid sample. [Vertex Wave Result](#)

In this problem you will create a simple vertex shader to generate procedural wave-like motion in a grid of particles. This problem shows how to use vertex shaders, uniforms, and vertex attributes. The Grid sample will be your starting code for this problem.

There are two types of inputs to shaders: vertex attributes and uniform variables. Vertex attributes are data that is given on a per-vertex basis; uniform attributes are the same across one or more primitives, i.e., an OpenGL draw call. The Grid sample has two vertex attributes: one for Position and one for Texture Coordinate (called TexCoord, s-t, or u-v). Position describes the position of the particles in model space and texture coordinates describes 2D coordinates over the mesh. Texture coordinates are usually normalized, meaning they are in the range [0, 1]. Your wave effect should vary sinusoidally across the grid with the texture coordinates. The required vertex attributes are already included.

Your wave motion should be time-dependent, so you will need to add a new uniform variable to pass a time parameter. Increment the time parameter with each display step; this will make the wave roll with time. To add a new uniform attribute, first declare it in the vertex shader, then get the location of the attribute by name with glGetUniformLocation, then upload a value with glUniform*.

Concretely, your vertex shader should adjust the height of the particles based on the following equation

```glsl
vec2 texcoord;
float time;
float PI = 3.14159...
float s_con Contrib = sin(texcoord.s*2.0*PI + time);
float t_con Contrib = cos(texcoord.t*2.0*PI + time);
float height = s_con Contrib*t_con Contrib*0.5f; //scale height
```

Your vertex shader should output a position by writing to gl_Position.

Implementation: 10%
After implementing the vertex shader, answer the following questions. Include your answers in your readme.txt.

1. In GPU programming, it is common to hear that “size is speed.” How can you reduce the amount of vertex attribute memory required for your vertex shader? (5%)

2. Before programmable vertex shaders, this wave would be implemented on the CPU, and a dynamic vertex buffer would be uploaded to the GPU each time step. Provide three reasons why using a vertex shader is more efficient. (10%)

**Part 2: Ward Shader (45%)**

**Coding Deliverable:**  
Ward anisotropic fragment shader (with accompanying vertex shader) based on the Sphere sample.  
**Ward Shader Result**

**Background:**

Read the following excerpt from *Advanced Renderman*, explaining the formulation for the Ward Shading Model:

Anisotropic materials are not uncommon. Various manufacturing processes can produce materials with microscopic grooves that are all aligned to a particular direction (picture the surface being covered with tiny half-cylinders oriented in parallel or otherwise coherently). This gives rise to anisotropic BRDFs. A number of papers have been written about anisotropic reflection models, including Kajiya (1985) and Poulin and Fournier (1990).

Greg Ward Larson described an anisotropic reflection model in his SIGGRAPH ’92 paper, “Measuring and Modeling Anisotropic Reflection” (Ward, 1992). In this paper, anisotropic specular reflection was given as

\[
\frac{1}{\sqrt{\cos \theta_i \cos \theta_r}} \cdot \frac{1}{4\pi \alpha_x \alpha_y} \exp \left[ -\frac{2 \left( \frac{\hat{h} \cdot \hat{x}}{\alpha_x} \right)^2 + \left( \frac{\hat{h} \cdot \hat{y}}{\alpha_y} \right)^2}{1 + \hat{h} \cdot \hat{n}} \right],
\]

where

- \( \theta_i \) is the angle between the surface normal and the direction of the light source.
- \( \theta_r \) is the angle between the surface normal and the vector in the direction the light is reflected (i.e., toward the viewer).
- \( \hat{x} \) and \( \hat{y} \) are the two perpendicular tangent directions on the surface.
- \( \alpha_x \) and \( \alpha_y \) are the standard deviations of the slope in the \( \hat{x} \) and \( \hat{y} \) directions, respectively. We will call these xroughness and yroughness.
- \( \hat{n} \) is the unit surface normal \( \text{normalize}(N) \).
- \( \hat{h} \) is the half-angle between the incident and reflection rays (i.e., \( H = \text{normalize} (\text{normalize}(-I) + \text{normalize}(L)) \)).
Problem:

Modify the “blinnphong.frag” and “blinnphong.vert” shaders and Sphere.cpp from the Sphere sample (available on the course website) to implement Ward’s formulation for anisotropic specular highlights. (Hint: the cosine of an angle is equivalent to the dot product of unit vectors).

This will involve passing uniform x roughness and y roughness uniform variables to the program and coding the above equation in the fragment shader. We found 0.2 for x_roughness and 0.8 for y_roughness work well; that gives a “velvet” effect.

The Sphere.cpp code in the Sphere sample creates a sphere by discretely evaluating the following parametric equation. u and v are texture coordinates which range from 0 to 1 over the sphere. u corresponds roughly to longitude and v to latitude:

\[ P_x = r \cdot \sin(\theta) \cdot \sin(\phi) = r \cdot \sin\left(\pi \cdot v\right) \cdot \sin\left(2 \cdot \pi \cdot u\right) \]
\[ P_y = r \cdot \sin(\theta) \cdot \cos(\phi) = r \cdot \sin\left(\pi \cdot v\right) \cdot \cos\left(2 \cdot \pi \cdot u\right) \]
\[ P_z = r \cdot \cos(\theta) = r \cdot \cos\left(\pi \cdot v\right) \]

The tangents can be derived by taking the partial derivatives of the parametric equation with respect to u and v. We have calculated these per-vertex already and included the vertex attributes for you. What the background reading refers to as x_hat and y_hat we refer to as U_Tangent and V_Tangent.

<table>
<thead>
<tr>
<th>U Tangent:</th>
<th>V Tangent:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{dP_x}{du} = 2 \pi r \cdot \sin(\pi v) \cdot \cos(2 \pi u) )</td>
<td>( \frac{dP_x}{dv} = \pi r \cdot \cos(\pi v) \cdot \sin(2 \pi u) )</td>
</tr>
<tr>
<td>( \frac{dP_y}{du} = -2 \pi r \cdot \sin(\pi v) \cdot \sin(2 \pi u) )</td>
<td>( \frac{dP_y}{dv} = \pi r \cdot \cos(\pi v) \cdot \cos(2 \pi u) )</td>
</tr>
<tr>
<td>( \frac{dP_z}{du} = 0 )</td>
<td>( \frac{dP_z}{dv} = -\pi r \cdot \sin(\pi v) )</td>
</tr>
</tbody>
</table>

Implementation: 30%

In addition to implementing Ward shading, answer the following questions. Include your answers in your readme.txt.

1. Draw a diagram of Ward shading, similar to the diagram for Phong shading shown in class. Include the angles and vectors from the Advanced Renderman excerpt above. Submit this as a .jpg or .png file mentioned in your readme.txt. (10%)

2. Assuming you are shading a sphere, how can you avoid storing normals, u tangent, and v tangents as vertex attributes? (5%)

Part 3: Sobel Operator (30%)
**Coding Deliverable:**
Sobel Operator using a fragment shader based on the Box_Blur sample. [Sobel Filter Result](#)

Besides traditional vertex and fragment shaders, image processing is another key application for GPU programming, ranging from post-processing bloom and blur effects in games to filtering steps in computer vision. Image effects like these are accomplished by texture a full-screen quadrilateral with the source image, and then rendering the quadrilateral. The fragment shader that is executed accomplishes the processing steps.

The Box_Blur example shows this setup. The default view (press ‘1’ on your keyboard) simply displays the source image; press ‘2’ to turn on a box-blur effect, so-called because it weights all neighboring pixels equally.

The problem for this part of the homework is to implement a Sobel Operator in a fragment shader. The Sobel Operator is a form of discrete differentiation often used in edge detection algorithms.

The Sobel Operator convolves two 3x3 kernels with the source image, to compute the gradient in the x and y directions, $G_x$ and $G_y$. $G$, the magnitude of the gradient, is $\sqrt{(G_x)^2 + (G_y)^2}$. The Sobel Operator can be performed independently on each color channel Red, Green, and Blue, so $G$ is a 3-dimensional vector, with a dimension per color channel. The final output should be the length of this vector.

If you are unfamiliar with convolution, it amounts to sampling nearby pixels in the source image and summing the samples weighted by a sliding window (the kernel). The box-blur example is equivalent to a convolution with a kernel with all equal values. Here are the two kernels for the Sobel Operator:

### Y Kernel:

<table>
<thead>
<tr>
<th>-1</th>
<th>-2</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

### X Kernel:

<table>
<thead>
<tr>
<th>-1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Implementation: 20%

In addition to implementing the Sobel Operator, answer the following questions. Include your answers in your readme.txt.

1. If your implementation was ran on the old NV40 architecture, e.g., a GeForce 6800,
which does not have unified shader processors, explain the hardware utilization of vertex and fragment processors. (5%)
2. How many texture reads does your fragment shader perform? (5%)