Homework 3: Programmable Shaders

Introduction to Computer Graphics and Imaging (Summer 2012), Stanford University
Due Monday, July 23, 11:59pm

Warning: The coding portion of this homework involves features of OpenGL that are much newer than ones we have used in previous assignments. Thus, there are some odds they won't work on your machine. Even if you don't start coding for the assignment early, do try compiling and running the skeleton code ASAP to make sure everything is functional. If there is an issue with your hardware, you may need to code this assignment in person on the Myth machines (forwarding may also fail).

In the written portion of this problem set, we'll explore properties of and computations involving barycentric coordinates, introduced in Lecture 4. These coordinates are used to interpolate normals, textures, depths, and other values across triangle faces, so it is important to understand how they behave.

To begin with, let's say we have a triangle with vertices $\vec{a}$, $\vec{b}$, and $\vec{c}$ in $\mathbb{R}^2$. We'll define the *signed area* of triangle abc as follows:

$$A(abc) = \frac{1}{2}\begin{vmatrix} b_x - a_x & c_x - a_x \\ b_y - a_y & c_y - a_y \end{vmatrix}$$

Problem 1 (10 points)**.** *(a) Use cross products to show that the absolute value of $A(abc)$ is indeed the area of the triangle that we expect. (b) Show that $A(abc)$ is positive when $\vec{a}$, $\vec{b}$, and $\vec{c}$ are in counterclockwise order and negative otherwise. Drawing pictures to demonstrate the solution for part (b) is good enough: no need for a formal proof.*

Now, let's take a point $\vec{p}$ in triangle abc. Assume that $\vec{a}$, $\vec{b}$, and $\vec{c}$ are given in counterclockwise order. Then, we can apply the formula above to yield straightforward expressions for barycentric coordinates.

Problem 2 (10 points)**.** *Define the barycentric coordinates of $\vec{p}$ using the signed area formula above. Show that your coordinates always sum to 1 and that they are all positive exactly when $\vec{p}$ is inside the triangle abc.*

Our definition of barycentric coordinates in terms of areas seems somewhat arbitrary! After all, we could take any three numbers as functions of vertex positions and normalize to 1. Thankfully, we won't make you prove it, but the following relationship holds uniquely:

$$\vec{p} = \alpha_a\vec{a} + \alpha_b\vec{b} + \alpha_c\vec{c}$$

where $\vec{p}$ has barycentric coordinates $(\alpha_a, \alpha_b, \alpha_c)$ summing to 1.

Problem 3 (10 points). *Use this formula to suggest a 2×2 linear system that will yield two of the three barycentric coordinates of $\vec{p}$. Provide a formula for the third.*

Hint: *Do the second part of the problem first. Then plug in the third coordinate in our expression for $\vec{p}$ and refactor into a linear system for α_a and α_b.*

If you've done everything correctly so far, you should find that applying the standard formula for the inverse of a 2×2 matrix to your solution to Problem 3 yields your signed area formula from Problem 2. You can check this on scrap paper if you desire.

Now, let's say we want to interpolate a function f smoothly given its values at $\vec{a}$, $\vec{b}$, and $\vec{c}$. We will label these values f_a, f_b, and f_c, resp., and will use $f(\vec{p})$ to denote the interpolated function with values everywhere on $\mathbb{R}^2$ given by

$$f(\vec{p}) = \alpha_a f_a + \alpha_b f_b + \alpha_c f_c$$

where the barycentric coordinates of $\vec{p}$ are $(\alpha_a, \alpha_b, \alpha_c)$. It's easy to see that $f(\vec{a}) = f_a$, $f(\vec{b}) = f_b$, and $f(\vec{c}) = f_c$ (make sure you understand why!). We also have one final important property:

Problem 4 (10 points). *Take $\vec{p}_1, \vec{p}_2 \in \mathbb{R}^2$. Show that $f(t\vec{p}_1 + (1-t)\vec{p}_2) = tf(\vec{p}_1) + (1-t)f(\vec{p}_2)$ for any $t \in \mathbb{R}$.*

Comprehensive hint: *Use barycentric coordinates to write*

$$\vec{p}_1 = \alpha_a^1 \vec{a} + \alpha_b^1 \vec{b} + \alpha_c^1 \vec{c}$$
$$\vec{p}_2 = \alpha_a^2 \vec{a} + \alpha_b^2 \vec{b} + \alpha_c^2 \vec{c}$$

Now, define $\vec{p} = t\vec{p}_1 + (1-t)\vec{p}_2$. Plug in our barycentric expressions for $\vec{p}_1$ and $\vec{p}_2$ above and refactor to yield the barycentric coordinates of $\vec{p}$ from the coefficients of $\vec{a}$, $\vec{b}$, and $\vec{c}$. Use these barycentric coordinates to find $f(\vec{p})$ as it is defined above the problem, and refactor once again to get the final result.

This final result actually shows us that barycentric coordinates are the right way to interpolate depths for the depth buffer (ignoring the nonlinearity we discussed in Lecture 4). Namely, if we restrict f to a line through $\vec{p}_1$ and $\vec{p}_2$, it looks linear.

By the way, just like Bresenham's algorithm, there are incremental tricks for computing barycentric coordinates inside triangles given their values on the edges. Modern GPUs make use of fragment-level parallelization and thus such tricks might not be useful.

In the coding half of this assignment,[1] you will get some practice using GLSL to write shaders, which implicitly make use of barycentric coordinates and other tools to modify fragments, vertex positions, and other data. Lecture 6 covers the basics of GLSL as well as the philosophy behind writing shaders. Recall that the two most basic types of shaders in OpenGL are:

Vertex shaders These shaders modify the vertices of a triangulated piece of geometry. The default OpenGL vertex shader multiplies vertex coordinates by model-view and projection matrices and applies other aspects of the fixed-function pipeline, but we can write a shader of our own to modify shape.

[1] Adapted from a nearly identical assignment used in Pat Hanrahan's CS 148 course in Fall 2011.

Fragment shaders These shaders let us modify the method used to compute pixel colors, potentially employing textures and other external data.

OpenGL also allows for *geometry shaders*, but we won't make use of them in CS 148.

Some useful facts beyond what was covered in lecture to get you started writing shaders:

- The OpenGL "Orange Book" documents details of GLSL and can help you look up advanced features and particular functionality. There also is some information in the "Red Book," in the course textbook, and in online tutorials.
- The entry point of a GLSL shader is the function `main()`, as in C. A vertex shader's main functionality is to compute `gl_Position`, the position of the vertex, and a fragment shader's functionality is to compute `gl_FragColor`, the color of the fragment.
- GLSL has all the simple C data types. It also has built-in vector types `vec2`, `vec3`, and `vec4` (with constructors like `vec2(x,y)`). You can access fields of these vectors as `v.x/v.y/v.z/v.w` or as `v.r/v.g/v.b/v.a`. You can extract the RGB part of an RGBA color by "swizzling:" `c.rgb`, and multiplying vectors does so *element-wise*. There are many useful helper functions including `dot()`, `cross()`, `length()`, `normalize()`, and `reflect()`.

The assignment this time comes with a makefile that should work on most systems, although it will be evaluated on the Myth machines. It makes use of the OpenGL Extension Wrangler Library (GLEW) to deal with shaders. Visual Studio and Xcode users shouldn't need any extra work to use this library. On Linux, you may need to install `libglew-dev` or download GLEW online (some systems may need to change the makefile to say "`-lGLEW`" rather than "`-lglew`").

Make sure you can compile and run the assignment ASAP, as GLSL support can be shaky from one computer to another. To do so, first compile the included `libst` library and then `ProgrammableShading`. You actually won't need to recompile these if you only change the shader files – shaders get compiled separately at runtime. To run the program, you will use the incantation:

```
./ProgrammableShading vertShader fragShader lightProbeImg normalMapImg
```

Here, `vertShader` and `fragShader` are the shader files, and `lightProbeImg` and `normalMapImg` are images that we will use in later parts of this assignment. For example, to run the default skeleton shaders, type:

```
./ProgrammableShading kernels/default.vert kernels/phong.frag
     images/lightprobe.jpg images/normalmap.png
```

Navigation in this simple viewer uses the left mouse button to rotate and the right mouse button to zoom. Pressing `r` resets the camera position, `t` toggles between displaying the plane and displaying the Utah teapot, and the arrow keys move the camera horizontally and vertically. Press `s` to take a screenshot and save it to `screenshot.jpg`.

Your first shader in GLSL will be a fragment shader implementing the Phong reflection model. Recall that Phong reflection separates specular, diffuse, and ambient reflection yielding a final intensity of

$$I = M_a I_a + (\vec{L}_m \cdot \vec{N}) M_d I_d + (\vec{R}_m \cdot \vec{V})^s M_s I_s$$

where

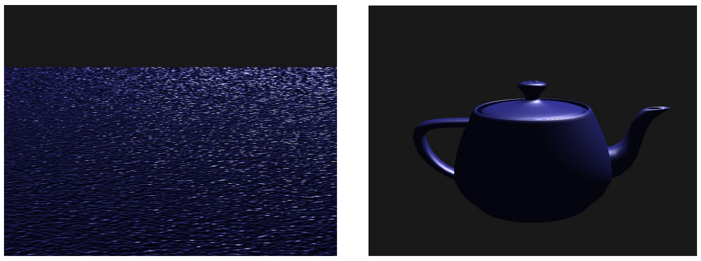

(a) (b)

Figure 1: For problem 5: (a) a normal-mapped plane; (b) a teapot shaded using the Phong model.

- I_a, I_d, and I_s are the ambient, diffuse, and spectral illuminations (we'll specify them as three-vectors of floats between 0 and 1 with one component for each color channel).
- M_a, M_d, and M_s are the material's ambient, diffuse, and spectral colors (we'll take $M_a = M_d$).
- $\vec{L}_m$ is a unit vector from the surface point to the light source.
- $\vec{N}$ is the unit surface normal.
- $\vec{R}_m$ is the *reflected* direction of a vector pointing from the surface to the light source over the normal $\vec{N}$.
- $\vec{V}$ is a unit vector pointing from the surface to the viewer.
- s controls the shininess of the material.

Note that multiplication like $M_a I_a$ happens component-by-component. Also note that the dot products could be negative, so you'll have to clamp them to be ≥ 0.

For fun, the skeleton code already implements *normal mapping*, which reads normal vectors from a texture to yield more interesting effects on the plane. Figure 1 shows this effect applied to the plane using `images/normalmap.png`, as well as the teapot with Phong shading but no normal map.

Problem 5 (30 points)**.** *Modify `kernels/phong.frag` to implement this shading model.*

Next, we'll try our hand at writing a vertex shader, implementing a species of displacement mapping where the (initially flat) plane is modified along its normal direction.

We'll simulate water ripples using a height field function specifying how each vertex on our plane should be moved up or down. Periodic functions are the easiest way to accomplish such an effect; for example, the function $s\cos(2\pi a u)\cos(2\pi a v)$ will make a rippling height field oscillating between s and $-s$, where u and v are texture coordinates. *Note: For full credit in your submission, come up with your own function rather than implementing this one!*

You should write a vertex shader that does the following:

1. Evaluate your height field on texture coordinates (u, v) to yield height $h(u, v)$.
2. Displace the vertex by $h(u, v)\vec{N}$.

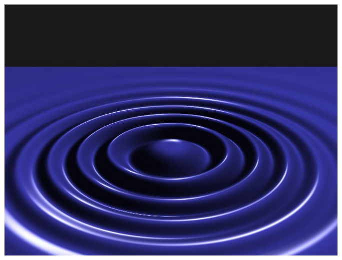


Figure 2: For problem 6.

Since you just modified the geometry, however, you'll need to do one more task. The normal to the plane used to be the constant $(0, 1, 0)$, but now our surface contains ripples and must be shaded as such! Thus, you must also:

3. Compute a new unit normal, using either a closed form or finite differencing (look it up or ask during office hours!).

Figure 2 shows proper output for one potential ripple function.

Problem 6 (30 points)**.** *Modify `kernels/displacement.vert` to implement your height field displacement shader. Be sure to output correct normals.*

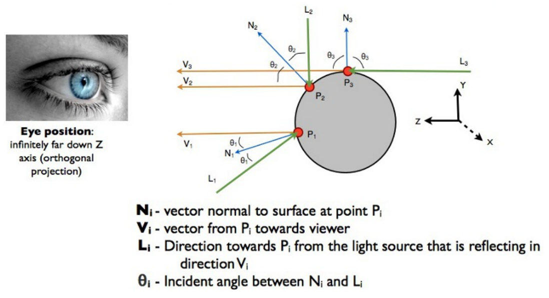
Optional Extra Credit

Shiny materials like metal reflect the surrounding environment. This is because these materials admit predominantly specular lighting. In class and in a future assignment, we'll talk about *ray tracing* as the "proper" solution to this problem, but it can be too slow for real-time applications. Instead, we'd like to compromise, using OpenGL's fast rendering to obtain a plausible reflectional rendering. We will use a technique called *environment mapping* (also known as *reflection mapping*) to achieve these effects. The trick is to use a *light probe* image encoding the color and intensity of the light coming in from all directions.

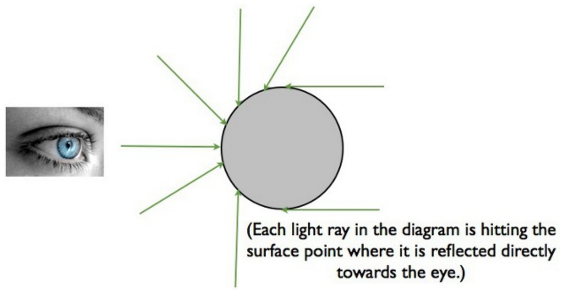
There are several ways to encode and decode such light information, but in this assignment we will use *spherical mapping* from a mirror ball image. Note that our math below assumes the light probe was photographed using an *orthographic projection*, an obviously nonphysical assumption but one that simplifies the math considerably.

Let's say we take a photo of a chrome ball in our scene, as in Figure 4(a). Since our ball only reflects specularly, the color we see at each surface point must come from a distinct light direction, illustrated in Figure 3(a). In the diagram, for each point on the surface, the direction of the light source that is illuminating that point from our camera perspective is shown with a green arrow. Note that on the top and bottom of the sphere, the light we see from the eye position is coming from directly behind the ball. In fact, as suggested in Figure 3(b), photographing *half* the sphere tells us about light in all directions!

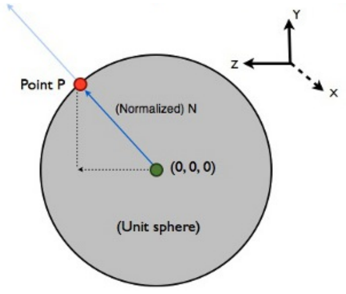
To extract lighting information from the light probe, we will assume that the direction and magnitude of the vector from the camera to the light probe is the same as the vector from the direction of our OpenGL scene camera to the point on the surface we are shading.


Eye position: infinitely far down Z axis (orthogonal projection)
N_i - vector normal to surface at point P_i
V_i - vector from P_i towards viewer
L_i - Direction towards P_i from the light source that is reflecting in direction V_i
θ_i - Incident angle between N_i and L_i

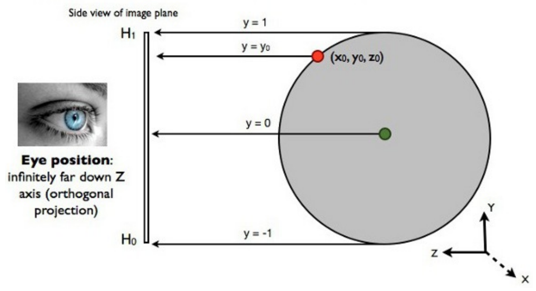
(a)


(Each light ray in the diagram is hitting the surface point where it is reflected directly towards the eye.)

(b)

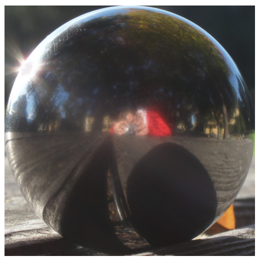


(c)

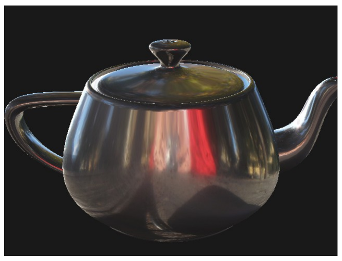


(d)

Figure 3: Images to explain the algorithm for problem 7.



(a)



(b)

Figure 4: For problem 7.

Recall that the source of specular light reaching the viewer at $\vec{V}$ is the reflection of $\vec{V}$ across the normal vector $\vec{N}$. We're assuming our probe and surface have approximately the same $\vec{V}$, so we only need to find the color of the point on the sphere that shares the same $\vec{N}$!

Let's say the center of the sphere is $(0,0,0)$. Then, the point on the sphere with normal $\vec{N}$ is simply $\vec{P} = (0,0,0) + \vec{N}$ (an interesting property of the unit sphere is that position and normal vectors are the same, shown in Figure 3(c)). We're assuming an orthographic projection of the unit sphere, shown in Figure 3(d), so to find the right lookup on the probe image we simply remove one of the three coordinates! Finally, we map $x, y \in [-1,1]$ to $x', y' \in [0,1]$ in image texture coordinates. Figure 4(b) shows a sample of the final result.

Problem 7 (15 points extra credit)**.** *Modify `kernels/lightprobe.frag` to implement the strategy for environment mapping described above. It should look good on the teapot; the normal-mapped plane will look messy since its normals change with such high frequency.*