Assignment 8: Programmable Shading

CS 148 Autumn 2012-2013

Due Date: Wednesday, November 28th, 2012 at 7pm

Introduction The goal of this assignment is to try your hand at writing GLSL shaders and to learn various techniques for performing surface perturbation and lighting tasks. This assignment uses advanced OpenGL features that we have not used for previous assignments. Even if you don't start the project early, you may want to *confirm that the computer you are using supports these features*, and get familiar with the `myth` cluster if it does not.

Background

Programmable Shaders We have already covered some aspects of GPU programming in lecture, but we will cover the basics here as well. A modern GPU (Graphics Processing Unit) is responsible for computing the position of every vertex and the color of every fragment/pixel that it draws to the screen. The original definition of the OpenGL machine (as given in the Red Book) describes a particular way of computing these positions and colors that we call the *fixed-function pipeline*. To enhance or alter the fixed-function pipeline, we may insert small programs called shaders that will do the computation of vertex positions and fragment colors. In OpenGL, these programs can be specified in the C-like **OpenGL Shading Language (GLSL)**. You can find some information on GLSL in Chapter 7 of the Red Book. In addition, the OpenGL "Orange Book" is entirely about GLSL.

For this assignment, we will be using the following two types of GLSL programs:

- **Vertex shaders** allow us to determine the final position of each vertex we tell OpenGL to draw (among other things). The default OpenGL vertex shader simply multiplies the vertex coordinates by the modelview and the projection matrices. Normal tranformation/normalization/rescaling, lighting, and texture coordinate generation/transformation also occur at this point in the pipeline if they are enabled. If you have these features enabled and you replace the OpenGL's default vertex shader with a vertex shader of your own, you will need to re-implement them.
- **Fragment shaders** let us compute the color that will be written to each pixel. Textures are accessed and applied here.

Note that OpenGL also has a 3rd type of shader called a geometry shader. We will not require you to learn about these for the purposes of this assignment.

A GLSL program (i.e. *shader*) is defined by a function `main`(), just like a C program. The `main()` function of a vertex shader is responsible for computing the value of `gl_Position`, the final position of the vertex. The `main()` function of a fragment program is responsible for computing the value of `gl_FragColor`, the final color of the fragment.

Note: The difference between 'fragments' and 'pixels' is a subtle one. Remember that geometry gets rasterized into fragments which are then composited onto pixels. There may, in general, be multiple fragments drawn to the same pixel (when things overlap). For this assignment, the difference isn't especially important, but we will do our best to use the terms correctly in our explanations.

A GLSL program can use simple C types like `int` and `float`, along with several built-in vector types: `vec2`, `vec3` and `vec4`. These vector types are a bit like the `STPoint`, `STVector` and `STColor` types all rolled into one. They support operations such as:

- Constructors – You can create a vec2 using an expression like `vec2(1.2, 3.4)`.
- Field access – You can get the components of a vec3 with `v.x, v.y, v.z`.
  - GLSL supports two variations on the field names: `x/y/z/w` and `r/g/b/a` are equivalent.
- "Swizzling" – You can extract the RGB part of an RGBA color `c` with `c.rgb`.
- Component-wise math – The standard addition, subtraction, multiplication and division operators operate component-wise on GLSL vectors.
  - Warning! Because GLSL uses the same type for vectors, points and colors, there are no restrictions on these operations like in `libST`.
  - In addition, many standard math functions (`sin()`, `sqrt()`, etc.) also operate component-wise on vectors.
- Vector operations – There are `dot()` and `cross()` functions for dot- and cross-product respectively. There is also a `length()` function that gives the length of a vector, and `normalize()` which returns a normalized copy of its vector parameter, as well as a `reflect()` function.

While the `main()` routine of a GLSL fragment program takes no arguments and returns void, it has access to a number of inputs. Each of the inputs is of one of two basic kinds:

- `uniform` parameters are specified through the OpenGL API. They may be either scalar types, vectors or texture types like 'sampler2D'. They are called uniform parameters because their value is the same across all fragments being shaded.
- `varying` parameters are those whose value may be different for each fragment. In the case of a fragment program, these are the values that were written by the vertex program, and interpolated by the rasterizer.

GLSL fragment programs cannot perform input/output (no `printf`), cannot write any results except for `gl_FragColor`, and cannot use pointers or read/write memory. These restrictions all mean that a GPU can run many distinct "instances" of a GLSL program in parallel without changing their meanings. This also forces us to be much more creative when debugging.

GPU Rendering For this assignment, we will be using vertex and fragment shaders to implement the Phong reflection model, simulate complex geometry on a flat plane with some analytically determined vertex and normal perturbation, and do some environment mapping. Because GPUs can run many program instances in parallel and include special hardware for operations like texture sampling they can often run these per-pixel and per-vertex operations many times faster than a CPU of similar size/cost.

Lots of applications that do graphical processing use shaders to reduce processing time. The vast majority of video games on the market today are using shaders as well, whether they are striving for lifelike realism (Battlefield 3) or more artistic effects (Limbo, Braid, Darkness II, Flower, Journey, etc.). By completing this assignment, you will gain experience with the OpenGL tools necessary to make these kinds of cutting-edge applications in real-time graphics and visualization.

Your Starting Package

- Code + Light Probe Images: assignment8.zip

As usual, we have just placed all the different versions (Visual Studio, Xcode, Makefile) into one package. Just download it and ignore the files that aren't related to your system. This starter package also includes `libST`. You'll need to use this version of `libST` (not an old one from a previous assignment) because we've made some modifications to the `STShaderProgram` class.

Build the `ProgrammableShading` program. Before starting on the code for this assignment, you'll want to run the starter program and make sure that the machine you are using displays everything correctly.

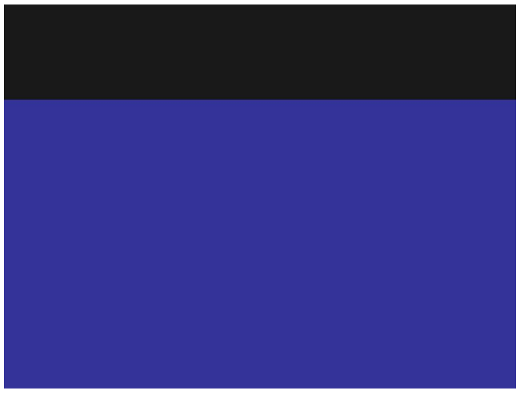


Figure 1: Output of the starter `ProgrammableShading` program

GLEW With this project we will start using another external library, the OpenGL Extension Wrangler Library (GLEW). For Visual Studio and Xcode users, there should be no additional steps for using this package – We have tried to set up the projects so things are automatic.

For Linux users, you will need to install this library to your own system. You might be able to install the `libglew-dev` package on your machine using the package manager for your distro, or get the source code directly from GLEW on SourceForge. You may find that you need to change the project make to use `-lGLEW` instead of `-lglew` to get things to build.

For instructions on how to get the assignment running on myth machines, please see how to get GLEW running on `myth` machines in the Appendix.

Try the Starter Program Because we will be using OpenGL functionality that your machine may not support, the first step after you have built `libST` is to build the `ProgrammableShading` program and try to run it. If everything works, you should see something like Fig. 1:

If It Didn't Work If the program prints out error messages saying that your GPU supports neither OpenGL 2.0 nor ARB extensions, your GPU may not be new enough for the assignment. If you have not done it already, try updating your graphics card drivers. Depending on your machine, you may have to go to the web site of your graphics card manufacturer (NVIDIA, AMD/ATI, Intel) or the website of your computer manufacturer. If things still don't work after you've updated your drivers, this means that your GPU is incapable of handling the shaders we will write for this assignment (as of last year, even the latest revisions of Intel integrated graphics were insufficient). Try the `myth` machines (Gates B08). We have successfully tested the application in the `myth` lab. You will probably need to do the assignment there **in person** – It is unlikely that you will be able to run these advanced OpenGL programs remotely.

Understanding/Using `ProgrammableShading`

Behavior: `ProgrammableShading` takes a set of command line arguments

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

Explanation:

- `vertShader` - The source file for the vertex shader program you want OpenGL to use.
- `fragShader` - The source file for the fragment shader program you want OpenGL to use.
- `lightProbeImg` - The light probe image that will be used by the environment map shader (See Step 3).
- `normalMapImg` - The normal map that will be used by the Phong shader (See Step 1).

To get started, run `ProgrammableShading` with the following arguments:

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

When the application starts up, it will draw a single blue plane on a black background.

- You can rotate the object you are looking at by dragging the mouse on the screen with a left-click.
- A right-button dragging will zoom in and out.
- You can move the camera (vertically and horizontally) with the arrow keys.
- You can reset the camera (position and orientation) by hitting the 'r' key.
- You can switch display between the plane and the Utah Teapot by hitting the 't' key.

The starter code also includes some screenshot functionality. Just press the 's' key to write the screen contents to a 'screenshot.jpg' file in your current working directory.

Implementation Notes The purpose of the code in `main.cpp` is to define and bind all necessary variables to the shader programs and then draw a finely tesselated plane, invoking our user-defined shaders when rendering the plane. All of our final images will result from implementing something more interesting in the shaders!

The `STShaderProgram` class is being used to load shader programs and bind uniform variables from the main program into the shader programs.

Instructions

Step 1: Implementing the Phong Reflection Model Your job here is to implement the Phong Reflection model for the plane using the material properties that are defined in `main.cpp`. Since we don't want you to get too bogged down in the minutiae of GLSL built-ins, we have already provided the normal vectors, all of the material properties and lighting information to you in the `phong.frag` shader file. You should take a look at how the material and light properties are set up in `main.cpp` for your edification.

Details of the Phong Reflection Model The **Phong Reflection Model** distinguishes between 3 types of reflection:

1. **Specular reflection -** This type of reflection occurs when a surface reflects light from a single light source in a single outgoing direction. Specular reflection is most easily observed on mirrorlike surfaces, such as polished metal or the surface of a body of still water, but can also be observed in combination with other types of reflectance on smooth, glossy/shiny materials (e.g. plastics, polished surfaces, etc.)
2. **Diffuse reflection -** This type of reflection occurs when a surface reflects light from a single light source in multiple outgoing directions. This type of reflection typically occurs on materials that are very rough at a fine level. Examples of such surfaces include pretty much any surface that's not liquid or polished metal, but some particularly good examples might include paper, anything made out of cotton, and plaster.
3. **Ambient reflection -** This is a non-physical type of reflection (ie, one which has no physical analogue). The ambient lighting term in the Phong model accounts for the small amount of light that has reflected off of so many surfaces before hitting your eye, that as an aggregate this light doesn't really have any direction anymore. Higher "ambient reflection" usually occurs in enclosed spaces where light is trapped and has plenty of opportunities to bounce back and forth. This is why an indoor shadow will probably be softer than, say, a shadow cast by a single light source outside at night.

Check out the visual summary in Fig. 2.

The formula for the Phong Shading model (with only one light source) defined in OpenGL state terms:

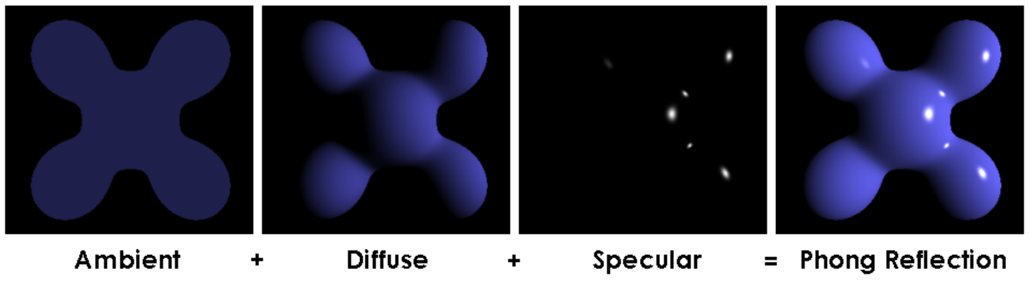


Figure 2: Phong Reflection

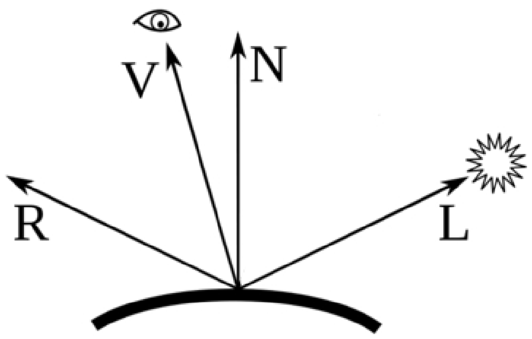


Figure 3: Phong Shading Model

$$I = M_a I_a + (\hat{\mathbf{L_m}} \cdot \hat{\mathbf{N}}) M_d I_d + (\hat{\mathbf{R_m}} \cdot \hat{\mathbf{V}})^{\text{shine}} M_s l_s \tag{1}$$

where:

- I_a, I_d, and I_s are the ambient, diffuse, and specular illuminations (In OpenGL, these are specified as a 3-vector 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 specular colors. If it seems odd to you that a material can simultaneously be 3 different colors, that's because it is odd - at least, its certainly odd to think of a material that way. But these terms don't really represent the color of the material itself, but rather the color of the material multiplied by some constants that, together, describe how the material reflects different kinds of light. In practice, we usually consider the ambient and diffuse material colors to be the same. We often set the specular material color to white (or at least something very bright) since specular reflections are usually the color of the specular light source, regardless of the color of the material.
- $\hat{\mathbf{L_m}}$ is the normalized vector from the surface point to the light source.
- $\hat{\mathbf{N}}$ is the normalized vector normal to the surface at this point.
- $\hat{\mathbf{R_m}}$ is the normalized direction to which that a light ray from the light source to this surface point would reflect (dependent on the surface normal),
- $\hat{\mathbf{V}}$ is the normalized direction pointing from the surface point towards the viewer.
- shine is a constant representing the shininess of the material. Higher shininess means a brighter, more spatially concentrated specular reflection.

Remember that these dot products could be negative, so you'll have to clamp them to (>= 0).

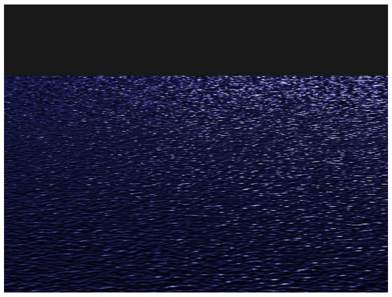


Figure 4: Ripple Effect with Normal Mapping

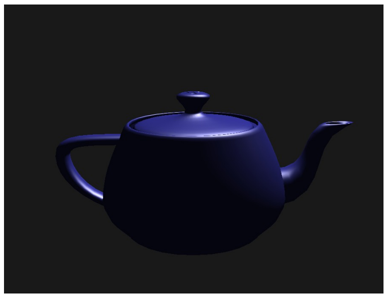


Figure 5: Utah Teapot with Phong Shading

Normal mapping The simple plane geometry that `ProgrammableShading` renders doesn't have very interesting normals; in fact, the normal at every vertex is exactly the same! To make shading results more interesting, the fragment shaders provided in the starter code use **normal mapping**. The shaders determine the normal vector at each fragment by looking up values from a texture. This works because the `(r,g,b)` components of a texel can be interpreted as the `(x,y,z)` coordinates of a normal vector. Using the file `images/normalmap.png` we obtain ripple effect in our shading as shown in Fig. 4.

This looks more like the surface of water than a simple, flat plane, but there's definitely room for improvement. Our rendering would look better if we could somehow move the actual **geometry** of the plane so that it rippled like water. Normal mapping doesn't do this; it only perturbs the normals on the plane. To get the effect we're looking for, we'll turn to a technique called **displacement mapping**.

In addition, the teapot with Phong shading is displayed in Fig. 5.

Step 2: Displacement Mapping in the Vertex Shader Displacement mapping refers to the process of using a programmable shader to displace the vertices of a piece of geometry. In some applications, this is accomplished by using a texture look-up, similarly to the way we've done normal mapping. In this assignment, you'll implement displacement mapping by writing a simple function that will determine how much each vertex should move.

Height Fields A height field is a function that takes an (x, y) position in on the plane and returns a single number indicating the 'height' of that point in space. Looking at a height field from the top-down is like looking at a topographical map; the larger the value at any point on the map, the higher the elevation. Height fields are commonly used to generate large amounts of outdoor terrain in video games. The two screenshots in Fig. 6 show an example height field and its resulting topography.

Simulating Water Ripples with Height Fields In this part of the assignment, you'll write a simple height field function that will specify, for each vertex on our simple plane, how high it should be moved up.

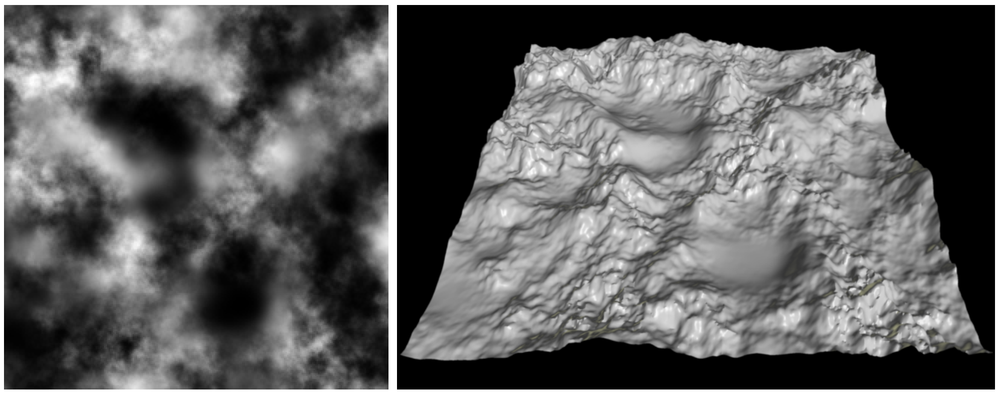


Figure 6: Example of a Height Field

Ideally, this function should try to produce an effect that looks like water ripples. Periodic functions are a great way to accomplish this. For example, the function

```
s*cos(a*2*pi*u)*cos(a*2*pi*v)
```

will generate a ripply height field that oscillates between `-s` and `s`. The parameter a controls the frequency of the ripples. Do not use this exact function in your implementation – You will only receive full credit for using your own displacement function. You should experiment with different functions until you get the effect that you want. Note that in the equation above, `u` and `v` refer to texture coordinates.

Displacing Vertices Now that you have a nice height field, you can use it in `displacement.vert` to displace the vertices of the plane. Suppose the vertex shader is running on vertex `x`. Here's what you need to do:

1. Evaluate your height field function using the texture coordinates '`(u,v)`' of `x`. Call the returned value `h(u,v)`.
2. Displace the position of `x` by `h(u,v) * normal`.

That's all there is to it! Well, almost...

Recomputing Normals Since your vertex shader has modified the plane's geometry, the normals at each vertex are no longer all $(0, 1, 0)$. The new surface created by the displacement has a new set of normals, so it would be a good idea to pass those normals on to the fragment shader for lighting calculations.

So how do we compute the new normals? Recall that the surface normal is the cross product of two surface tangent vectors; the diagram in Fig. 7 illustrates this point.

If we think of our surface as being defined by a 2D function `P(u,v)`, then the two tangent vectors are just the partial derivatives of this function with respect to `u` and `v`. The above diagram illustrates this, too.

Now back to your code: `displacement.vert` will need to compute the two surface tangent vectors in order to find the new normal. Taking exact partial derivatives of your height field function can be a pain, especially if your function is complicated. Instead, we'll use a technique called **finite differencing**. Specifically, you can approximate the partial derivatives of your height field function by:

```
dh/du = (h(u+delta,v)-h(u,v))/delta
dh/dv = (h(u,v+delta)-h(u,v))/delta
```

where `delta` is a very small number.

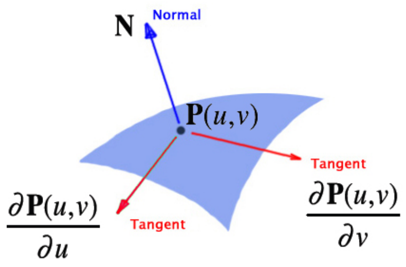

N
Normal
$\mathbf{P}(u,v)$
Tangent
Tangent
$\frac{\partial \mathbf{P}(u,v)}{\partial u}$
$\frac{\partial \mathbf{P}(u,v)}{\partial v}$

Figure 7: Surface Normal

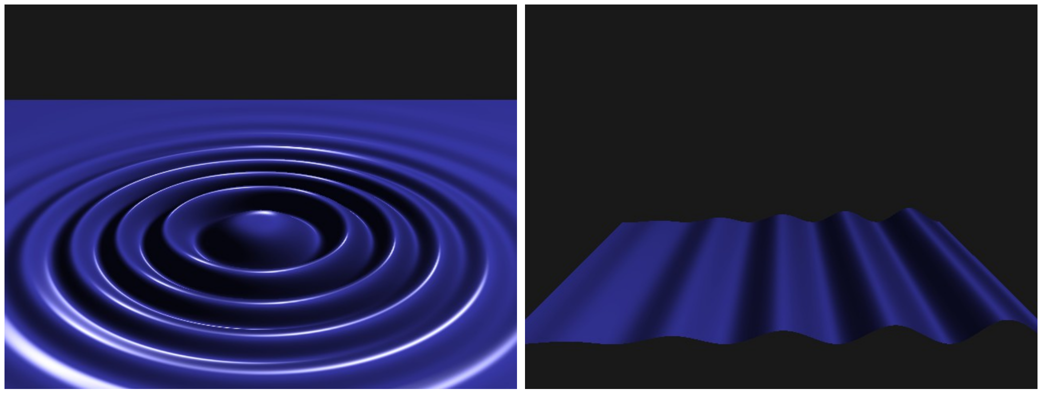


Figure 8: Displacement Mapping

Armed with finite differencing, computing the surface tangent vectors, and then the normals, is straightforward:

```
T1 = Normalize([1,dh/du,0])
T2 = Normalize([0,-dh/dv,-1])
N = T1 x T2
```

With displacement mapping implemented, your code should be able to produce images that look like those in Fig. 8.

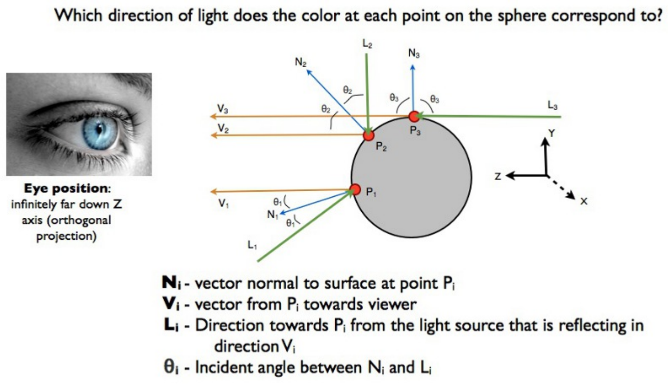
Step 3: Simulating Reflections on a Mirror-like Material with the Light Probe In this step, we will be simulating the reflective properties of water. You'll take your tonemapped light probe image, load it into the shader, and use it as an environment map for determining the specular reflection color of the plane at each fragment.

(Spherical) Environment Mapping In real life, shiny, mirror-like materials will have reflections of the surrounding environment on their surfaces. This is because the material is only (or mostly) reflecting the light specularly. In order to accurately produce this effect computationally, we would need to run a ray tracer and simulate the possible paths that light could take before bouncing off the material towards your eye to get a realistic reflection on the surface of the material. This kind of photo-realism is nice, but it is expensive to compute, especially compared to the native, simple reflection model that OpenGL uses. Of course, since we are running real-time graphics, we would like to find a way to capture some of this realism without paying the computational cost.

The goal of environment mapping (a.k.a. reflection mapping) is to produce realistic reflections by using a precomputed texture image that encodes all the lighting information of the scene (we call this image a light probe). Specifically, the texture encodes the color and intensity of the light coming into a single point in space from all directions. If our reflective surface is close enough to the source of the **light probe**, we can approximate the the lighting conditions of that point with the light probe and get some impressively realistic results.

There are many techniques for encoding and decoding light information into/from an image, but for this assignment, we will be using **spherical mapping** on a texture of our clipped mirror ball image. In order to understand how we extract lighting information out of the light probe image, let's first investigate how light is reflected off the surface of a mirror ball. **Note** that we are assuming our light probe was taken with an **orthographic projection** instead of a perspective projection, because this greatly simplifies our model and calculations and turns out to have very good results. (This detail is pretty important for understanding the following diagrams.)

When we took our light probes, we set up the camera so that it was pointed directly towards the chrome ball in our scene. Let's investigate how light in such a scene would reflect from the sphere towards the camera ("eye").


Which direction of light does the color at each point on the sphere correspond to?
Eye position: infinitely far down Z axis (orthogonal projection)
L_2
N_2
N_3
θ_2
θ_2
θ_3
θ_3
L_3
V_3
V_2
P_3
P_2
Y
Z
X
P_1
V_1
θ_1
N_1
θ_1
L_1

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

Figure 9: Light Reflection from Sphere towards Camera

Since the mirror ball only reflects specularly, the color we perceive at each surface point on the ball must come from a distinct light direction. 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 drawn with a green arrow. Note that on the top of the sphere, the light we are "seeing" from the camera/eye position is coming from directly behind the ball. In fact...

From the diagram, you should be able to see that for any direction, there is a point somewhere on the surface of the half of the sphere facing the camera where light from that direction is reflected to the camera. Since our light probe image contains half a sphere, the image should have the complete lighting information for that point in space!

Extracting Lighting Information from a Spherical Light Probe There are a few different ways to approach extracting lighting information from the light probe, depending on how you want to orient the coordinate system of the light probe with respect to your surface. Here, we will discuss a method which assumes 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 and the point on the surface we are shading. You can use this model and get full credit on the assignment.

Recall from our previous discussions that the source direction of specular light will be the reflection of **V**

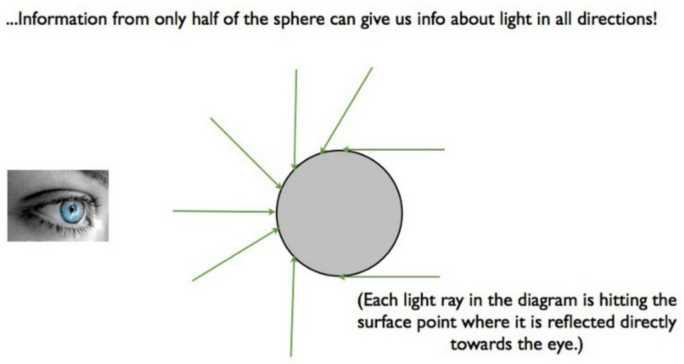

...Information from only half of the sphere can give us info about light in all directions!
(Each light ray in the diagram is hitting the surface point where it is reflected directly towards the eye.)

Figure 10: Light from Half of the Sphere

(the vector from the eye/camera position to the point) across the normal vector **N** at that surface. Since our light probe and our surface point have (approximately) the same **V** vector, we only need to find the color of the point on the sphere that shares the same surface normal to find the illuminance hitting our surface point from the relevant light source direction. So how do we find this point on the sphere?

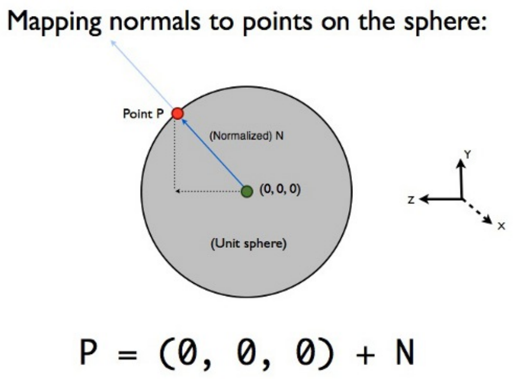


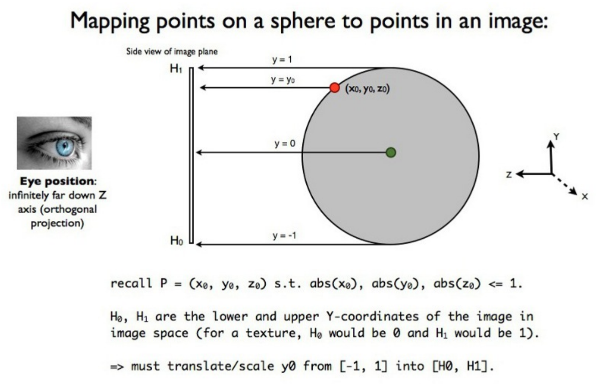
Figure 11: Mapping Normals to Points on the Sphere

As the above diagram shows, due to the unique properties of the sphere, a vector from the sphere's center to any point on its surface will be parallel to the surface normal at that point, so figuring out a surface point from a specified normal turns out to be trivial. Now that we have the correct point on the sphere...

Since we consider our light probe image to be the orthographic image of the mirror ball, we can linearly map between the `(x, y)` coordinates of the point on the surface and the `(x', y')` coordinates in the image.

Now that you know the basics behind spherical reflection mapping, you are ready to implement this in a fragment shader! Please do so and put your environment mapping code in `lightprobe.frag`. Note that you can either use the normal map surface, your displacement map surface, or some other surface of your choosing, just as long as you make sure the surface is smooth.

Sample Light Probes If your light probe from assignment 6 is unusable, we have provided a whole bunch of nice light probe images you can use. Download them here: lightProbes.zip. These were adapted from the HDR environment maps found at Paul Debevec's website.


Mapping points on a sphere to points in an image:
Side view of image plane
H₁
H₀
y = 1
y = y₀
y = 0
y = -1
(x₀, y₀, z₀)
Eye position: infinitely far down Z axis (orthogonal projection)
Y
Z
X

```
recall P = (x₀, y₀, z₀) s.t. abs(x₀), abs(y₀), abs(z₀) <= 1.

H₀, H₁ are the lower and upper Y-coordinates of the image in
image space (for a texture, H₀ would be 0 and H₁ would be 1).

=> must translate/scale y0 from [-1, 1] into [H0, H1].
```

Figure 12: Mapping Points on a Sphere to Points in a Image

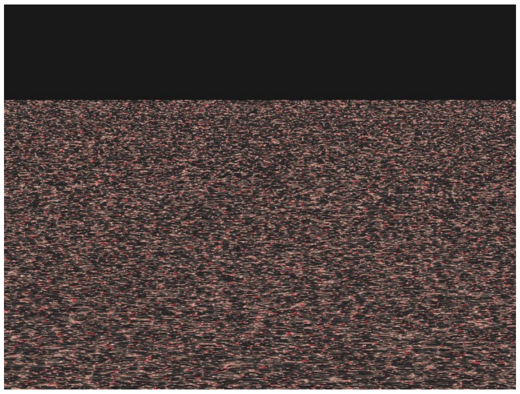


Figure 13: Output from default vertex shader, normal map, and light probe

What is it supposed to look like? The image generated by using the default vertex shader and the normal map and light probe files in the `images` directory is shown in Fig. 13.

Since the normals for a surface like water are often repeated, the full potential of the light probe environment map is not really used. If you switch to the Utah Teapot, you should see something like Fig. 14

Assignment Questions

1. In GLSL, we can compute values in the vertex shader and pass them along to the fragment shader. However, when we are computing values in the vertex shader, we are computing them on a per-vertex basis, not a per-fragment basis. How does OpenGL determine what values to assign to each fragment?
2. In displacement mapping, we perturb the vertices and the normals. However, in normal mapping, we only perturb the normals. Name at least one advantage and one disadvantage of using displacement mapping rather than normal mapping when simulating more complex geometry.
3. Why does OpenGL maintain separate modelview and projection matrices?
4. Environment mapping assumes that (a) all radiance hitting the illuminated surface is coming from infinitely far away and (b) the reflective object being environment-mapped is convex. Can you think of any visual artifacts that might show up if these assumptions are violated?

Possible (Non-required) Extensions Here are just a few ideas:

1. Design a particularly amazing/water-like displacement function.

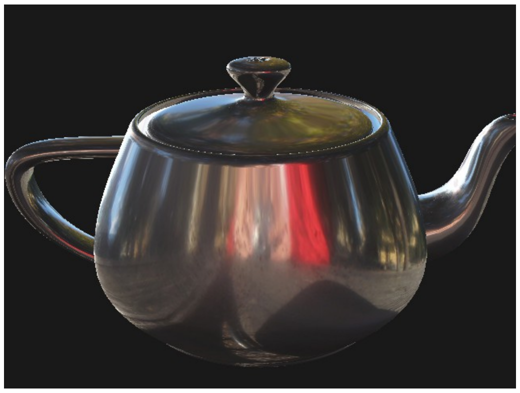


Figure 14: Utah Teapot with environment mapping

2. Animate your displacement function over time to create the impression of moving water. You can do this by passing a uniform float variable called 'time' to your vertex shader.
3. Add other objects that use other GLSL shaders to the scene. These other shaders can use other lighting models, do some bump mapping, etc, in order to simulate different kinds of materials. You might put a boat on the water, for example.
4. Make normal/displacement mapping work for non-planar surfaces. Think about the relationship between normal and tangent vectors, and how, if you had a little more information in the shader, you could correctly perturb any normal, not just normals that are facing straight up.

Grading

- 1 pt – Correctly answered questions
- 1 pt – Phong reflection
- 1 pt – Displacement mapping
- 1 pt – Environment mapping

How to get GLEW running on Myth machines

- Download GLEW. Grab the tarball since it has Linux config files. (To untar `file.tar.gz`, type `tar -xf file.tar.gz`).
- Unzip the assignment. This should create the `ProgrammableShading` directory.
- Unzip/Untar the GLEW source into the `ProgrammableShading` directory. This will result in the creation of a `ProgrammableShading/glew` directory.
- Run `make` inside `ProgrammableShading/glew`.
- Change directories to `ProgrammableShading` and open the `Makefile`. Modify the `LIBS += st GL glew` by replacing `glew` with `GLEW`. Make sure you modify the `LIBS` in the `#building` on Linux section of the `Makefile`.
- Build `libST`.
- Now you should be able to build `ProgrammableShading` as well. If you are still having problems linking to `glew`, try changing `glew` in the `Makefile` `LIBS` list to `GLEW`.
- To run `ProgrammableShading`, you will also need to add the `glew` shared objects to your dynamic library loading path. To do this, you need to modify the `LD_LIBRARY_PATH` environment variable to include the path to the `glew/lib` directory. Here is the command you would run if you had placed your `assignment8` directory directly in your home directory:

```
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:$HOME/assignment8/ProgrammableShading/glew/lib
```

If your `assignment8` is somewhere else, modify the `$HOME/assignment8` portion of the above command to include the correct path to `assignment8`. Note that you will need to re-run this command every time you log in to a `myth` machine.

Now you should be ready to run the assignment! If this does not work, make a Piazza post.