CS123 Lab 09 - Shaders III

“Let the form of an object be what it may, light, shade, and perspective will always make it beautiful.”
  -John Constable

1 Introduction

This lab will be entirely focused on shaders and GLSL. You will use the CS123 shader editor in /course/cs123/bin, which you can run by typing cs123_shadervis in a terminal.

In this lab you will be writing two material shaders which try to mimic the properties and appearance of various materials. The default Blinn-Phong lighting used by OpenGL is often insufficient for creating convincing replicas of materials such as glass, water, metal, skin, or ice. This is partially due to the fact that the Phong lighting model has no physical foundation—it just “looks okay.” Shaders allow for much more control over lighting and shading.

For this lab you will be writing two different shaders: a glass shader and a metal shader. Both these shaders will be using physically-based models to calculate illumination. In particular, they will try to mimic Fresnel reflectance and refractance, and will use the physically-based Cook-Torrance model to determine specular reflection.

Fig. 1: Glass and metal shaders applied to the dragon.obj model
1.1 Cube Environment Mapping

Notice that both of these materials interact with the scene, by reflecting and/or refracting it. For now, simulating this perfectly is too hard, so we’ll approximate using a technique called environment mapping.

The idea behind environment mapping is to encode into a texture all data that can be reflected or refracted. To simulate reflection, for example, we simply figure out where in the scene the reflected data should come from and sample the texture (called an “environment map”) correspondingly.

Though computationally cheap, this technique only allows objects to reflect the environment: objects do not show reflections of each other. Later in this course you’ll write a raytracer, which will be able to handle inter-object reflection among other things.

In this lab we will encode our environment in a cube map. Cube maps are a collection of six texture maps that form a cube, where each face of the cube is covered with one of the six 2D textures:

![Cube Map](image)

**Fig. 2:** Example of an environment cube map, unrolled into a single image

We’ll choose textures so that, when you stand inside the cube, the sides of the cube work together and appear to form an entire scene. We’ll then surround the object we’re shading in a “skybox” to make it appear as if the object being rendered is inside the environment, and we’ll have the shader index into the environment cube map to determine what data to reflect and/or refract. Since the skybox and the shader will use the same cube map, it will appear as though the object is reflecting and refracting the scene. Remember to make sure you always use the same cube map for the skybox and the shader! Otherwise it will look like your object is reflecting a different scene than the one it’s in.

The skybox has already been written for you (you get one for free in the Preview pane of the shader editor); the shader that indexes into the cube map will be up to you.
1 Introduction

1.2 Fresnel Equations

The following section details the Fresnel equations\textsuperscript{1} which model how light interacts with objects. If you aren’t concerned with the physical basis, you may skip ahead to Schlick’s approximation, which you will need for this lab.

![Diagram of light with a surface](shamelessly lifted from Wikipedia)

Fig. 3: Diagram of light with a surface (shamelessly lifted from Wikipedia)

The Fresnel equations describe the behavior of light as it moves between media with different indices of refraction. When light passes from one medium into the next, it may be reflected or refracted. The Fresnel equations can be used to compute the fraction of light that is reflected and the fraction of light that is refracted. This will be useful in both the glass and metal shaders in order to properly model the behaviors of each material.

In practice, the Fresnel equations are too expensive to compute for real-time graphics applications, so we will instead be using Schlick’s approximation for this lab. If you would like to learn more about the Fresnel equations, \url{http://en.wikipedia.org/wiki/Fresnel_equations} is a good place to start.

1.3 Schlick’s Approximation

In your metal and glass shaders, you will use Schlick’s approximation to calculate the reflectance $F$ and the transmittance $T$. The reflectance (or reflection coefficient) is the fraction of light that is reflected, and the transmittance (or transmission coefficient) is the fraction of light that is refracted. Under our model all light is either reflected or transmitted, so naturally we have $T = 1 - F$.

We determine $F$ by Schlick’s approximation as follows:

$$F \approx R_0 + (1 - R_0)(1 - \cos \theta_i)^5$$

\textsuperscript{1} pronounced FRAY-nell
In this equation, $\theta_i$ is the same as $\theta_i$ in the diagram from the previous section, and $R_0$ is the reflectance at normal incidence (i.e. the fraction of light reflected when $\theta_i$ is zero).

# 2 Metal Shader

## 2.1 Background

Metals and other shiny materials have large specular reflections. These specular highlights are important visual cues, helping viewers determine the shape and location of an object as well as the location of light sources acting on the object.

Real materials don’t have perfect specular reflections. Even surfaces that appear smooth are not perfectly so: their surfaces are covered in very tiny facets (“microfacets”), each of which is a perfect specular reflector. The material appears smooth because these microfacets don’t differ greatly in their orientations. However, the material will still appear somewhat rough because the microfacets are not aligned perfectly.

The Phong and Blinn-Phong illumination models were the first illumination models to try mimicking specular reflection phenomena. However, these models are not based on reality. Recall that the Phong model splits the lighting into three different terms: ambient, diffuse, and specular. For more information, see the Wikipedia article: [http://en.wikipedia.org/wiki/Phong_shading#Phong_reflection_model](http://en.wikipedia.org/wiki/Phong_shading#Phong_reflection_model)

It is relatively easy to implement Phong in shader code:

```cpp
// Example Phong vertex shader in eyespace (viewspace)
varying vec3 normal, vertexToLight, vertex;

void main() {
  // Transform normal vector into eye coordinates
  normal = normalize(gl_NormalMatrix * gl_Normal).xyz;

  // Transform position into eye coordinates
  vec3 vertex = (gl_ModelViewMatrix * gl_Vertex).xyz;

  // Get the vector from the vertex to the light source
  vertexToLight = normalize(gl_LightSource[0].position.xyz - vertex);

  // Project the normal and position into screen coordinates
  gl_Position = ftransform();
}

// Example Phong fragment shader in eyespace (viewspace)
uniform vec4 color; // User-supplied material color ...
uniform float specular; // ... and specular power coefficient
varying vec3 normal, vertexToLight, vertex; // Computed in vertex shader

void main() {
  // Re-normalise interpolated values
```

vec3 n = normalize(normal);
vec3 l = normalize(vertexToLight);

// The vector from the eye to the vertex is vertex - (0, 0, 0), since the vertex is in eye coordinates
vec3 eyeToVertex = normalize(vertex);

// Calculate the light here using the equation from the Illumination slide deck....

gl_FragColor = color * intensity;

In the Phong model,

$$k_{spec} = \cos^n(E, R)$$

where $n$ is the specular coefficient, $E$ points in the direction of the viewer, and $R$ is the reflection of the incoming light vector about the normal vector. Larger values of $n$ produce brighter, more localized specular highlights.

Notice that the normal vector is normalized in both the vertex shader and the fragment shader. In general, if you need a vector normalized within a fragment shader, it should also be normalized in the vertex shader. The full reason for this can be found at the bottom of this page: [http://zac.in.tu-clausthal.de/teaching/cg_literatur/gsl_tutorial/index.html](http://zac.in.tu-clausthal.de/teaching/cg_literatur/gsl_tutorial/index.html).

Also notice that the vertex shader uses several different matrices to calculate $N$, $E$, and $L$ (gl_ModelViewMatrix and gl_NormalMatrix). Since OpenGL lighting is mostly done in eye coordinate space, we need to transform object coordinates into eye coordinates using these matrices. From the OpenGL FAQ (http://www.opengl.org/resources/faq/technical/transformations.htm), remember that

- Object Coordinates are transformed by the ModelView matrix to produce Eye Coordinates
  - Object coordinates are the raw coordinates you submit to OpenGL with a call to glVertex*(n). They represent the coordinates of your object or other geometry you want to render
- Eye Coordinates are transformed by the Projection matrix to produce Clip Coordinates
- Clip Coordinates $X$, $Y$, and $Z$ are divided by $W$ to produce Normalized Device Coordinates
- Normalized Device Coordinates are scaled and translated by the viewport parameters to produce Window Coordinates
2 Metal Shader

2.1.1 A Better Illumination Model

There are many other illumination techniques which better model the distribution of microfacets on a surface. One of these is the physically-based Cook-Torrance model. You will be using this model to calculate the specular term. In particular,

\[ k_{\text{spec}} = \frac{DFG}{E \cdot N} \]

\( D \) is the Beckmann distribution factor,

\[ D = \exp\left(-\frac{\tan^2(\alpha)/m^2}{4m^2\cos^4(\alpha)}\right), \quad \alpha = \arccos(N \cdot H) \]

\( m \in [0, 1] \) controls the roughness of the surface, and \( H \) is the “half-vector,” named because it is halfway between the eye vector (which goes from the vertex to the viewer) and the light vector (which goes from the vertex to the light). See [http://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model](http://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model) If \( u \) is the vector from the vertex to the light and \( v \) is the vector from the vertex to the eye, then we find \( H \) by adding \( u + v \) and then normalizing. This is pictured in the diagram below.

\[ F \] is the Fresnel coefficient (calculated using Schlick’s approximation), and \( G \) is the geometric attenuation term caused by selfshadowing of the microfacets in the surface,

\[ G = \min \left( 1, \frac{2(H \cdot N)(E \cdot N)}{E \cdot H}, \frac{2(H \cdot N)(L \cdot N)}{E \cdot H} \right) \]

In the above formulas, \( E \) is the look vector of the eye, \( H \) is the half angle vector, \( L \) is the vector to the light source, and \( N \) is the normal vector. Consider which values should not be less than 0 (this will save you a lot of headache later).
2.2 Implementation

2.2.1 Sampling a Cube Map

Recall texture sampling from the Shader 1 lab:

```cpp
uniform sampler2D tex;
void main() {
    gl_FragColor = texture2D(tex, gl_TexCoord[0].st);
}
```

Sampling a cube map works similarly, except OpenGL uses a 3D vector to index into the cube instead of 2D texcoords. OpenGL will do all the work of figuring out which face to sample and which pixels on that face to use. For example, the input \((0, -1, 0)\) would sample the center of the bottom face, while \((1, 1, 1)\) would sample the corner where the \(+X\), \(+Y\), and \(+Z\) faces meet. The GLSL for sampling a texture cube is:

```cpp
uniform samplerCube tex;
void main() {
    gl_FragColor = textureCube(tex, myVec3);
}
```

When using cube environment maps, the \texttt{vec3} you pass to \texttt{textureCube()} will always be a normalized direction vector pointing where in the environment you want to read a color.

Take a look at the diagram in section 1.2. We want to simulate reflection by doing the following:

- Using the normal and the eye direction vector, compute the incident direction vector. You may find the GLSL \texttt{reflect} function useful for this.
  - Don’t forget: If you want to sample from the cube map correctly you need to calculate your reflected vector from the world coordinates. You’ll need to find a way to transform from eye coordinates to world coordinates for this one step.

- The incident vector obtained may be pointing towards the point being rendered. Reverse it so that it points into the environment, away from the object being rendered.

- Sample the environment cube map using that vector.

2.2.2 Metal Shader

To get started, copy the stencil code from /course/cs123/src/labs/lab09. Open \texttt{metal.vert} / \texttt{metal.frag} (if you open either, Shadervis will automatically open the other one for you).

For this shader, you will first compute the ambient /diffuse/specular material color, and then you will compute the reflected environment color by sampling properly from the cube map. Then you will blend these colors according to the
Fresnel term in order to determine the final fragment color. Use the following high-level steps:

- Determine the diffuse and ambient terms exactly as you would in a Phong shader.
- Instead of using the Phong model to calculate specular reflections, use the Cook-Torrance model. That is, determine the coefficient of specular reflection $k_s$ using the equations presented in section 2.1.1.
- Compute the object color (without the reflection component) as $ambient + lambertTerm \times diffuse + k_s \times specular$.
- Sample the cube map to determine the reflection color.
- Calculate the transmittance $F$ by Schlick’s approximation as use it to blend in the reflection color. You might consider using GLSL’s mix() function.

3 Glass Shader

3.1 Background

When you look at a material like glass, you see a mixture of reflected and refracted (or transmitted) light. You already did reflection in the previous shader; now it is time to implement refraction too.

When light is refracted at the interface between two media, its direction changes according to Snell’s law:

$$\frac{\sin(\theta_i)}{\sin(\theta_t)} = \frac{n_2}{n_1}$$

The variables in this equation correspond to the diagram in section 1.2; $n_1$ and $n_2$ are the indices of refraction of the media.

Our strategy for refraction will be similar to reflection: using the GLSL refract function, compute the direction of the incident vector that refracts in the direction of the eye vector, and then sample the cube map to simulate the incoming light.

The following refraction shader samples a refracted ray from the surrounding cube environment map:

```c
// Vertex shader
varying vec3 normal;
varying vec3 vertex;
void main() {
    vertex = gl_ModelViewMatrix * gl_Vertex;
    normal = normalize(gl_NormalMatrix * gl_Normal);
    gl_Position = ftransform();
}
```
3 Glass Shader

```glsl
// Fragment shader
uniform samplerCube envMap;
varying vec3 normal;
varying vec3 vertex;
float eta = .77; // ratio of IORs (n2/n1 in Snell's law)
void main() {
    vec3 n = normalize(normal);
    vec3 eyeToVertex = normalize(vertex);
    vec3 t = gl_ModelViewMatrixInverse * vec4(refract(eyeToVertex, n, eta), 0.0);
    gl_FragColor = vec4(textureCube(envMap, t).rgb, 1.0);
}
```

However, this is not very realistic. Recall the Fresnel equations described above. When light hits an interface between two media, some is reflected and some is refracted. Furthermore, different wavelengths of light get refracted different amounts.

### 3.2 Implementation

Open `glass.vert` / `glass.frag`. You will need to fill in the fragment shader. (The vertex shader is already complete.) Your shader should use Schlick's approximation to determine how much light is reflected and how much is refracted. The ratio of reflected to refracted light should be the Fresnel term, like in the metal shader.

Once you have reflection and refraction, you need to refract light by different amounts for each color channel (R, G, and B). To do this, change your shader to calculate three refraction vectors (one for each color channel), each of which is refracted by a slightly different amount (use the values in the `uniform vec3 eta` declared for you in the stencil code).

To summarize, here are the high-level steps which should be performed by `glass.frag`:

- Sample the cube map to determine the reflection color.
- For each of the R, G, and B channels, determine the direction from which refracted light originated. Use these direction vectors to respectively sample the R, G, and B channels in the environment map.
- Compute the reflection coefficient using Schlick's approximation, and use it to blend the reflected and refracted light.

Once you see both the glass and metal shaders working in `cs123_shadervis`, see a TA to get checked off for lab²

---

² Yay!