Phong Shading with Spatially Varying Material Properties

CS 442/542

Due 11:59 PM Friday, December 16, 2016

1 Introduction

Figure 1: Texture map (left) used to select between two material properties. Plane and sphere (middle and right) Phong shaded by spatially blending two different material properties. Note that the red material is more glossy.

For this project you will extend your WebGL program for rendering a surface created by extruding a disc along a toroidal spiral. The illumination calculations are moved from the vertex shader to the fragment shader and are thus performed for each fragment (much like the normal map example from class). A texture map is used to determine which of two sets of material properties to use for fragment shading as illustrated in Figure 1.

Begin with a simple color texture map as described in Section 2. This will require you to add texture coordinates to your tube geometry. The illumination computations will performed in the fragment shader described in Section 4. Section 3 describes how to construct the corresponding vertex shader; this vertex shader is reasonably generic and can be reused whenever shading is to be performed on a per fragment basis. This process will also require specifying the surface tangents for each vertex. The steps needed to modify the tube program to use these new shaders as described in Section 5. What you are to submit is outlined in Section 6.

2 Texturing the Tube

Figure 2: Modulated shading of a blue tube using the texture map on the left of Figure 1.

Before implementing the final shaders you should first texture the surface as a color map. Here I described the necessary modifications to your "tube" program to texture the surface as shown in Figure 2.

The mesh your created in the previous project consisted of $(\texttt{TUBE\_N}+1) \times (\texttt{TUBE\_M}+1)$ vertices and normals (each holding 3 float's). Start by adding another buffer holding 2D texture coordinates:

```
texCoords = new Float32Array(2*(N+1)*(M+1));
```

Compute texture coordinates that map one instance of the image onto the entire surface:

```
var index = 0; // index into texCoords buffer
for (var i = 0; i <= TUBE_N; i++)
  for (var j = 0; j <= TUBE_M; j++) {
    texCoords[index++] = i/TUBE_N;
    texCoords[index++] = j/TUBE_M;
  }
```

Add a texture matrix so we can tile multiple copies of the texture (the following we create $38 \times 2 = 76$ copies of my lion texture): over the surface:

```
TextureMatrix = new Matrix4x4;
TextureMatrix.scale(38, 2, 1);
```

Load a texture as described in class. We will ultimately consider the texture to represent a single gray-level value, but loading RGB images is simpler in WebGL; My shader will eventually just use the red channel for determining which material properties to use as described in Section 4. Test using the simple vertex and fragment shader from your superquadric project that performs illumination computations in the vertex shader, and modulates this with the texels in the fragment shader.

3 The Vertex Shader

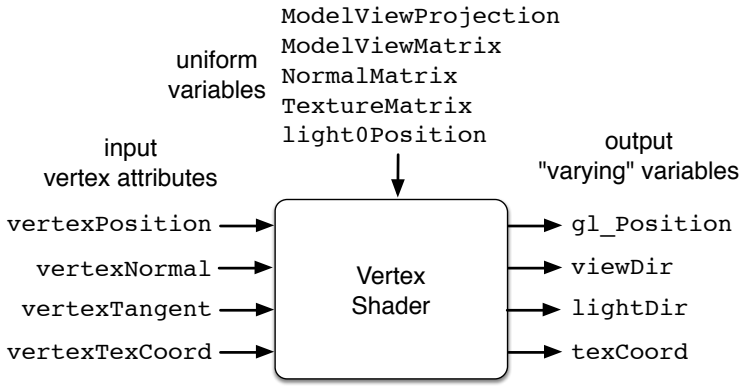

Figure 3: Vertex shader inputs and outputs.

The vertex shader (see Appendix A) will not actually shade anything, but it will pass the necessary data down the pipeline so that shading can be performed per fragment. Figure 3 lists the inputs and outputs of the vertex shader. We will assume there is only one point light source.

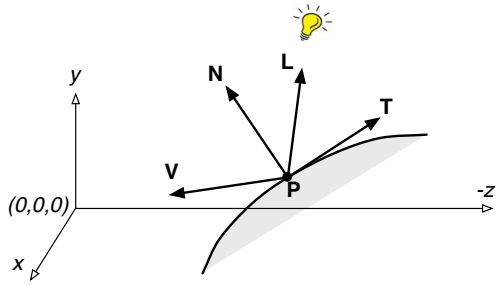

Figure 4: Vertex position $\mathbf{P}$, vertex normal $\mathbf{N}$, vertex tangent $\mathbf{T}$, and light direction $\mathbf{L}$ in view coordinates. The "eye" is at the origin and looks out the $-z$ axis.

Since shading is typically performed at surface points in view coordinates, the vertex shader must first transform the vertex (and possibly the light position) with the Model-View matrix. Then view vector $\mathbf{V}$ and light direction vector $\mathbf{L}$ are constructed as shown in Figure 4. The input normal $\mathbf{N}$ is transformed into view coordinate via the inverse-transpose up the upper 3×3 the Model-View matrix (aka `NormalMatrix`). Each tangent $\mathbf{T}$ is transformed into view coordinates using the upper 3×3 the Model-View matrix (ignores the translation component of the matrix).

We define a local coordinate system using the following set of orthonormal axes:

$$\hat{\mathbf{B}} = \frac{\mathbf{N} \times \mathbf{T}}{\|\mathbf{N} \times \mathbf{T}\|}, \tag{1}$$

$$\hat{\mathbf{T}} = \frac{\mathbf{B} \times \mathbf{N}}{\|\mathbf{B} \times \mathbf{N}\|}, \tag{2}$$

$$\hat{\mathbf{N}} = \frac{\mathbf{N}}{\|\mathbf{N}\|}. \tag{3}$$

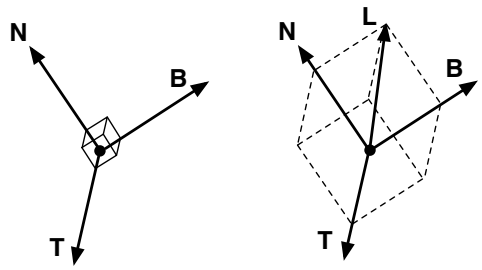

Figure 5: The tangent $\mathbf{T}$, binormal $\mathbf{B}$, and normal $\mathbf{N}$ form an orthonormal set of axes (left). We project the light direction $\mathbf{L}$ into this coordinate system (right).

The tangent $\hat{\mathbf{T}}$,binormal $\hat{\mathbf{B}}$, and normal $\hat{\mathbf{N}}$ act as the x, y, and z axes in this coordinate system (see Figure 5). The light and view vectors are transformed into this coordinate system by projecting them onto these three axes. This is actually a rotation defined by the 3×3 matrix whose rows are $\hat{\mathbf{T}}$, $\hat{\mathbf{B}}$, and $\hat{\mathbf{N}}$:

$$R = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{bmatrix}. \tag{4}$$

The vertex shader's output view and light direction are $R\mathbf{V}$ and $R\mathbf{L}$ respectively.

The input vertex position is transformed into *clip coordinates* by the Model-View-Projection matrix to create the output vertex position (`gl_Position`). We will transform the input 2-D texture coordinate with the Texture matrix (`TextureMatrix`) so we can alter the location and area of the material properties dynamically.

4 The Fragment Shader

The fragment shader ((see Appendix B) will "color" the fragment using the Phong Illumination model based on the environment lights and material properties (which vary from fragment to fragment). First we fetch the material property blending value β from the texture map:

```
float beta = texture2D(texUnit, texCoord).r;
```

For the majority of texels in the texture map, β will be either 0 or 1 (corresponding to the black and white pixels in the image on the left of Figure 1). For value between 0 and 1 we will create a mixture P of the texture properties P_0 and P_1 :

$$P = \texttt{mix}(P_0, P_1, \beta) = (1 - \beta) \cdot P_0 + \beta \cdot P_1. \tag{5}$$

For example, the diffuse factors are computed as

```
vec3 materialDiffuse = mix(material0Diffuse, material1Diffuse, beta);
```

Say we are given the following mixture of material properties:

a	`AmbientFactor`
d	`DiffuseFactor`
s	`SpecularFactor`
n	`Shininess`

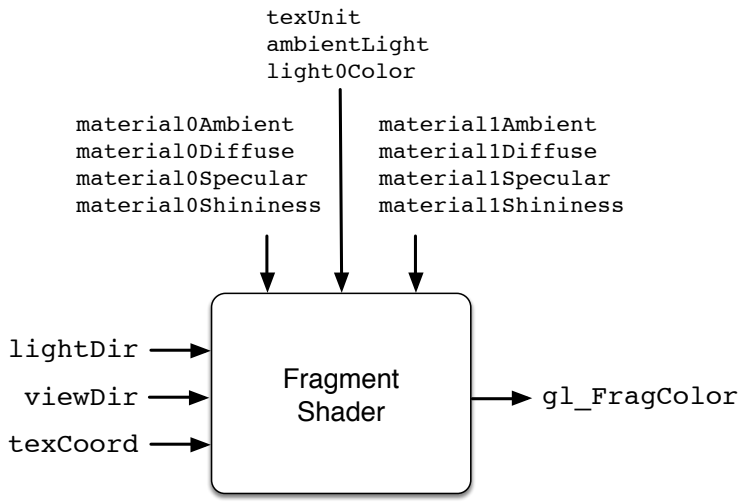

Figure 6: Fragment shader inputs and output. Note that there are two sets of material properties which will be blended using the texel fetched by the shader.

The reflected ambient light is

$$I_a = \texttt{ambientLight} \cdot a. \tag{6}$$

The surface color is modulated diffuse reflected light:

$$I_d = \texttt{light0Color} \cdot d \cdot \max(0,\ \mathbf{N} \bullet \mathbf{L}). \tag{7}$$

The white highlight is modulated by the amount of specular reflected light:

$$I_s = \texttt{light0Color} \cdot \max(0,\ s \cdot (\mathbf{R} \bullet \mathbf{V})^n) \tag{8}$$

And the resulting fragment color is the resulting sum

$$\texttt{gl\_FragColor} = I_a + I_d + I_s \tag{9}$$

Since $\mathbf{N} = (0, 0, 1)$ in BTN-coordinates, some of the math above is simplified:

$$\mathbf{N} \bullet \mathbf{L} = L_z, \tag{10}$$

$$\mathbf{R} = (-L_x,\ -L_y,\ L_z). \tag{11}$$

5 Shading the Tube

Now we describe the steps needed to modify the "tube" program render the surface with our vertex and fragment shaders as shown in Figure 7.

Our vertex shader now requires a `Tangent` attribute for each vertex. Therefore, we need to create another buffer to hold tangent vectors:

```
tangents = new Float32Array(3*(N+1)*(M+1));
```

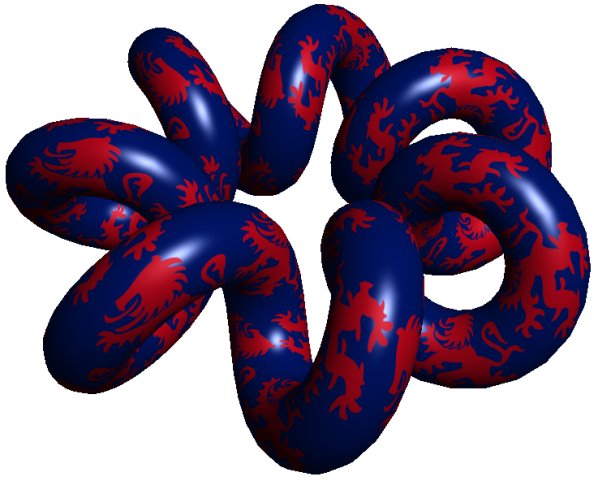

Figure 7: Fragment shaded tube. Note that the red lions are duller than the shiny blue background.

Fill the `tangents` buffer with the tangent vectors `T` computed while building the Frenet Frame . Make sure to replicate the appropriate vectors for the extra row and column in the "wrap around" vertices. Here is the added code snippet from my program (see the previous toroidal tube project description):

```
for (var k = 0; k < 3; k++) {
   this.normals[n] = cosu*N_[k] + sinu*B[k];
   this.verts[n] = C[k] + R*this.normals[n];
   this.tangents[n] = T[k]; // added tangent vector
   n++;
}
```

Then load the tangents into a VBO

```
tube.tangentBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, tube.tangentBuffer);
gl.bufferData(gl.ARRAY_BUFFER, tube.tangents, gl.STATIC_DRAW);
```

When you render the surface make sure to enable the extra vertex attribute:

```
gl.bindBuffer(gl.ARRAY_BUFFER, tube.tangentBuffer);
gl.enableVertexAttribArray(program.vertexTangent);
gl.vertexAttribPointer(program.vertexTangent, 3, gl.FLOAT, false, 0, 0);
```

6 What to Submit

You are free to use my lion texture map, but you might like to make one of your own. Your program should behave as described in the "tube" project, but with your new shaders in place. Create a `README` file with the usual stuff in it (name and contact, build/run instructions, content list, ...). Submit by midnight on the due date.

A Vertex Shader GLSL Source

```
attribute vec4 vertexPosition;
attribute vec3 vertexNormal;
attribute vec3 vertexTangent;
attribute vec2 vertexTexCoord;

varying vec2 texCoord;
varying vec3 lightDir;
varying vec3 viewDir;

uniform mat4 ModelViewProjection;
uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 TextureMatrix;

uniform vec3 light0Position;

void main() {
  gl_Position = ModelViewProjection*vertexPosition;
  texCoord = (TextureMatrix*vec4(vertexTexCoord, 0.0, 1.0)).st;

  vec3 P = vec3(ModelViewMatrix * vertexPosition);
  vec3 N = normalize(NormalMatrix * vertexNormal);
  vec3 T = normalize((ModelViewMatrix  * vec4(vertexTangent, 0.0)).xyz);
  vec3 B = normalize(cross(N, T));
  T = cross(B,N);

  vec3 L = normalize(light0Position - P);
  lightDir = vec3(dot(L,T), dot(L,B), dot(L,N));
  vec3 V = normalize(-P);
  viewDir = vec3(dot(V,T), dot(V,B), dot(V,N));
}
```

B Fragment Shader GLSL Source

```
precision mediump float;

varying vec2 texCoord;
varying vec3 lightDir;
varying vec3 viewDir;

uniform sampler2D texUnit;

uniform vec3 material0Ambient;
```

```
uniform vec3 material0Diffuse;
uniform vec3 material0Specular;
uniform float material0Shininess;

uniform vec3 material1Ambient;
uniform vec3 material1Diffuse;
uniform vec3 material1Specular;
uniform float material1Shininess;

uniform vec3 ambientLight;
uniform vec3 light0Color;

void main() {
  float beta = texture2D(texUnit, texCoord).r;
  vec3 materialAmbient = mix(material0Ambient, material1Ambient, beta);
  vec3 materialDiffuse = mix(material0Diffuse, material1Diffuse, beta);
  vec3 materialSpecular = mix(material0Specular, material1Specular, beta);
  float materialShininess = mix(material0Shininess, material1Shininess, beta);

  vec3 L = normalize(lightDir);
  vec3 V = normalize(viewDir);
  vec3 R = vec3(-L.x, -L.y, L.z);

  vec3 I_ambient = materialAmbient * ambientLight;
  float diffuse = max(L.z, 0.0);
  vec3 I_diffuse = materialDiffuse * light0Color * vec3(diffuse, diffuse, diffuse);
  float specular = pow(max(dot(V,R),0.0), materialShininess);
  vec3 I_specular = materialSpecular * light0Color * vec3(specular, specular, specular);

  vec3 color = I_ambient + I_diffuse + I_specular;

  gl_FragColor = vec4(color, 1.0);
}
```