SCIT

School of Computing & Information Technology

CSCI336 – Interactive Computer Graphics

Mapping and Lighting Techniques

In this tutorial, we will look at other techniques based on fragment processing and lighting.

Cube Environment Mapping

We can also use textures to create a fake reflection effect. To do this we use 6 images that represent the 6 faces of a cube. When joined together, these images should not have any gaps in between the seams. Open the Tutorial7a project.

First look at the code in Tutorial7a.cpp. For convenience, enumerated values are used to represent the faces of the cube:

```
enum CUBE_FACE { FRONT, BACK, LEFT, RIGHT, TOP, BOTTOM };
```

We store the images and create an identifier for the cube environment map:

```
unsigned char* g_texImage[6];
GLuint g_textureID;
```

In the init() function, we read in the contents of the images, which must have the same dimensions:

```
GLint imageWidth;
GLint imageHeight;

g_texImage[FRONT] = readBitmapRGBImage("images/cm_front.bmp", &imageWidth, &imageHeight);
g_texImage[BACK] = readBitmapRGBImage("images/cm_back.bmp", &imageWidth, &imageHeight);
g_texImage[LEFT] = readBitmapRGBImage("images/cm_left.bmp", &imageWidth, &imageHeight);
g_texImage[RIGHT] = readBitmapRGBImage("images/cm_right.bmp", &imageWidth, &imageHeight);
g_texImage[TOP] = readBitmapRGBImage("images/cm_top.bmp", &imageWidth, &imageHeight);
g_texImage[BOTTOM] = readBitmapRGBImage("images/cm_bottom.bmp", &imageWidth, &imageHeight);
```

Next, we generate a texture identifier and create the textures for the faces of the cube map:

```
glGenTextures(1, &g_textureID);

glBindTexture(GL_TEXTURE_CUBE_MAP, g_textureID);

glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGB, imageWidth, imageHeight, 0, GL_BGR,
       GL_UNSIGNED_BYTE, g_texImage[RIGHT]);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 0, GL_RGB, imageWidth, imageHeight, 0, GL_BGR,
       GL_UNSIGNED_BYTE, g_texImage[LEFT]);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y, 0, GL_RGB, imageWidth, imageHeight, 0, GL_BGR,
       GL_UNSIGNED_BYTE, g_texImage[TOP]);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, GL_RGB, imageWidth, imageHeight, 0, GL_BGR,
       GL_UNSIGNED_BYTE, g_texImage[BOTTOM]);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z, 0, GL_RGB, imageWidth, imageHeight, 0, GL_BGR,
       GL_UNSIGNED_BYTE, g_texImage[BACK]);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, 0, GL_RGB, imageWidth, imageHeight, 0, GL_BGR,
       GL_UNSIGNED_BYTE, g_texImage[FRONT]);
```

Then the texture parameters are defined:

```
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

Note that GL_CLAMP_TO_EDGE should be used to avoid any gaps in the seams of the cube map.

If you look at the images in the "images" folder, you will notice that they look upside down. This is due to the OpenGL cube map coordinate system, which is different from the other OpenGL coordinate systems.

In the render_scene() function, before we render the object we bind the texture. Note that we bind it to the GL_TEXTURE_CUBE_MAP rather than GL_TEXTURE_2D:

```
glUniform1i(g_envMapSamplerIndex, 0);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, g_textureID);

glDrawElements(GL_TRIANGLES, g_numberOfFaces * 3, GL_UNSIGNED_INT, 0);
```

In the vertex shader, we simply transform the normal vector to world space. Note that we do not require any texture coordinates, as we can infer which coordinates to use to sample the texture from the normal vector.

The fragment shader, receives the interpolated normal vector and we work out the texture coordinate based on the reflected vector from the eye by the normal vector:

```
vec3 ReflectEnvMap = reflect(-Eye, Normal);
```

This is then used to sample the cube map:

```
fColor = texture(uEnvironmentMap, ReflectEnvMap).rgb;
```

```
And modulated with the lighting colour:
fColor *= (diffuse + specular + ambient).rgb;
```

Compile and run the program. Rotate the object and move the camera around, you will see that it creates the illusion of reflection.

To see a perfectly mirrored surface that does not have any colour by itself, but simply reflects everything, comment out the following line:

```
//fColor *= (diffuse + specular + ambient).rgb;
```

Compile and run the program. You will see a mirror like surface.

Blending

The next example in the Tutorial7b project looks at blending. Using blending, we can blend the destination colour (in the colour buffer) with the source colour (colour coming in) by a certain blend factor, typically given by the alpha channel (i.e. the fourth component in RGBA).

The most common blending function is to multiply the source colour with alpha and add it with the destination colour which is multiplied by 1-alpha. The following is used to set up this blending function:

```
glBlendEquationSeparate(GL_FUNC_ADD, GL_FUNC_ADD);
glBlendFuncSeparate(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_ONE, GL_ZERO);
```

The following commands are used to enable and disable blending:

```
glEnable(GL_BLEND);

glDisable(GL_BLEND);
```

In this example, instead of making the alpha component a part of the colour, we will pass it to the fragment shader as a uniform variable.

The vertex and fragment shaders in this example are very simple. The vertex shader simply passes the colour to the fragment shader, which outputs the colour with the value of alpha as the fourth colour channel.

Compile and run the program.

To get a better understanding of how blending works, do the following:

- Toggle blending on and off. You will notice a greater contribution of the objects' colour with blending turned off.
- With blending turned on, change the value of the Alpha component. At 0.0, the object is completely transparent, whereas at 1.0 the object is completely opaque.
- Use the user interface to change the background colour. Notice how the source colour is blended with the destination colour (colour in the colour buffer). Observe the effect of toggling blending on and off.
- The next thing to consider is how the depth test affects rendering translucent objects.
    - At the moment, the object on the right is in front of the other object. You can see this from the z value in the interface. Toggling the depth test will not make a difference, because if you look at the code, the object on the right is rendered later (remember the "painter's algorithm")
    - Now with the depth test on, move the object behind the first object (i.e. change the translation z to a negative value). Notice that nothing is blended. The first object simply overlaps the second object. This is because the first object is rendered and the depth buffer stores its depth values. When the second object is rendered, parts that are behind the first object will fail the depth test and will not be rendered at all.

- Now disable the depth test. The object colours will be blended.
- With the depth test disabled, switch the order in which the polygons are rendered by unchecking the Order checkbox. Toggle this a few times. Notice that the order in which the objects are rendered makes a difference to how they are blended. This is because the source and destinations colours will be different depending on which is rendered first.

Toon Shading

In previous tutorials, you have seen how to implement per fragment lighting and shading. We can modify the code slightly to perform basic toon shading.

First thing that we want to do is to draw an outline of an object. We can obtain this based on the surface normal and the direction towards the camera. This is illustrated in Figure 1. Figure 1 shows a scene from a top-down point of view, where the camera is looking at an object. As can be seen from the figure, we can determine the edges of an object if the surface normal is almost perpendicular to the direction of the camera. To obtain this information, we can use the dot product between the surface normal and the vector in the direction of the camera.

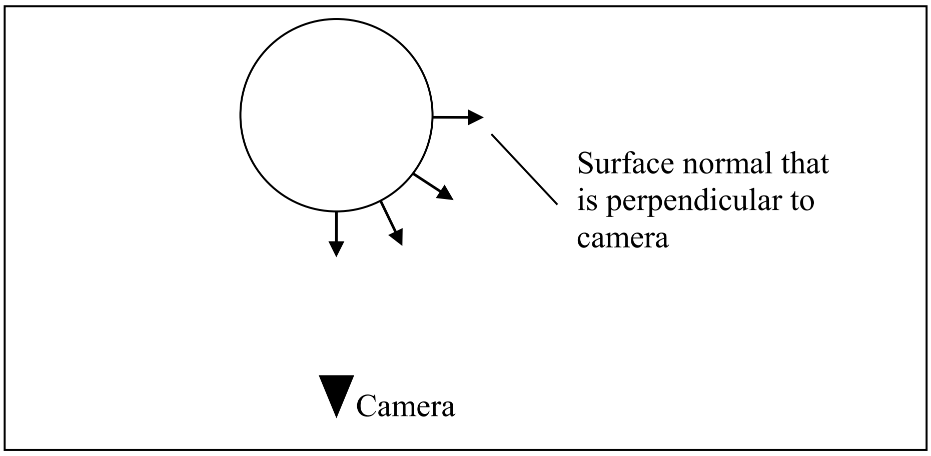

Figure 1

Open the Tutorial7c project.

Run and compile the program. You will see that it is initially no different from the lighting and shading examples from the previous tutorial.

Now, have a look at the code in the fragment shader. The vector towards the camera for a fragment is obtained using:

```
	vec3 Eye = normalize(-vPosition);
```

This is dot product with the normal. A resulting value of 0 indicates that the surface is perpendicular to the camera. We can define a threshold to determine the thickness of the toon outline, anything below this threshold will be rendered in black and we skip the rest of the computation by simply returning. Uncomment the following code:

```
       if(dot(Eye, Normal) < uEdgeThickness)
       {
              fColor = vec3(0.0f, 0.0f, 0.0f);
              return;
       }
```

Note that in this program, the threshold is passed as a uniform variable to the shader to allow you to see the effects of adjusting the thickness threshold.

We do the usual things in the Tutorial7c.cpp code for obtaining the location of this uniform shader variable and setting it in the shader before rendering. This should be familiar from the previous tutorials, so the explanation will not be repeated here.

Compile and run the program. You will now see the object with a black outline. You can adjust the thickness of this outline.

Next, for toon shading we want to threshold the diffuse colours to a few levels (we do not need to touch the ambient colour since this component is uniform over the entire object). What this means is that instead of allowing the full range of colours, we restrict colours to be within a fix number of intensities. One way of doing this is to do something like this:

```
if (intensity > 0.0f && intensity <= 2.5f)
       colour = someColour1;
else if (intensity > 2.5f && intensity <= 5.0f)
       colour = someColour2;
else if (intensity > 5.0f && intensity <= 7.5f)
       colour = someColour3;
...
```

Note that the above is just an example, you do not need uniform thresholds levels.

The code in this program does something similar by using a number of uniform thresholds. The user can adjust the number of thresholds using a uniform shader variable. We threshold the colour values based on the angle that vector towards the light makes with the surface normal.

Uncomment the following (which will overwrite the diffuse calculation from the previous line):

```
float thresholdFactor = floor(max(dot(Light, Normal), 0.0) *
       uNumOfThresholds)/uNumOfThresholds;
diffuse = thresholdFactor * uLightingProperties.diffuse * uMaterialProperties.diffuse;
```

The dot product between the vector towards the light and the surface normal will result in a value between 0.0f and 1.0f (the `max` function ensures a non-negative value). This value is multiplied with the number of thresholds. The built-in `floor` function returns the closes integer value, which is then divided by the number of thresholds. The result of this is that the threshold factor will be quantized based on the number of thresholds.

Compile and run the program. You will see that the resulting colour of the object has been quantized based on the number of thresholds. Try using the interface to change the number of thresholds. Also, try changing the material's diffuse colour.

The specular component still does not look right. We want something similar to the diffuse component, i.e. to restrict the intensity levels. For the specular component, we do not need that many levels, hence in this example we will use a single cutoff threshold. For any intensity below this threshold, we will ignore the specular computation.

The specular intensity was calculated as:

```
float specularFactor = pow(max(dot(Eye, Reflect), 0.0), uLightingProperties.shininess);
```

Uncomment the following line of code in the fragment shader:

```
	if(specularFactor > uSpecularCutoff)
```

What this does is if the specular intensity is below the cutoff threshold, do not calculate the specular contribution.

Compile and run the program. The specular component now cuts off after the specified intensity, which you can adjust using the interface.

Try moving the light source and rotating the object to see how the resulting colours change.

Gooch Shading

Like toon shading, Gooch shading is a non-photorealistic rendering technique. In Gooch shading, in addition to the base surface colour, the shading transitions between a warm colour (surface facing the light) and a cool colour (surface facing away from the light). An example of this is implemented in the Tutorial7d project.

Examine the code in the fragment shader. Only the light position and specular exponent is required. Also, for the material properties, a surface colour, a cool colour and a warm colour must be provided:

```
struct LightProperties
{
	vec4 position;
	float shininess;
};

struct MaterialProperties
{
	vec4 surfaceColour;
	vec4 coolColour;
	vec4 warmColour;
};
```

The contribution of the surface colour is controlled by an alpha and a beta factor, which are passed to the shader as uniform variables:

```
uniform float uAlpha;
uniform float uBeta;
```

In the fragment shader's main() function, first the usual lighting vectors are computed:

```
vec3 Normal = normalize(vNormal);
vec3 Light = normalize((uViewMatrix * uLightingProperties.position).xyz - vPosition);
vec3 Eye = normalize(-vPosition);
vec3 Reflect = reflect(-Light, Normal);
```

Similar to the toon shading explanation above, the dot product between the direction towards the light and the surface normal indicates the direction of the surface with respect to the light. Hence, the cool and warm colours are transitioned based on this value. The portion of the surface facing the light receives a higher contribution of the warm colour, whereas the surface facing away from the light receives a higher contribution of the cool colour. The built-in GLSL `mix` function linearly interpolates between two values.

```
float dotLightNormal = max(dot(Light, Normal), 0.0);

vec4 kCool = uMaterialProperties.coolColour + uAlpha * uMaterialProperties.surfaceColour;
vec4 kWarm = uMaterialProperties.warmColour + uBeta * uMaterialProperties.surfaceColour;
vec4 colour = mix(kCool, kWarm, dotLightNormal);
```

Next, the specular contribution is calculated:

```
float specular = 0.0f;

if(dotLightNormal > 0.0f)
{
       specular = pow(max(dot(Eye, Reflect), 0.0), uLightingProperties.shininess);
}
```

The final colour is computed by adding the colour with the specular contribution and clamping the value to a maximum of 1.0f:

```
fColor = min(colour + specular, 1.0f).rgb;
```

Compile and run the program. You can adjust the various properties using the interface to see how they change the rendered image.

References

Among others, much of the material in this tutorial was sourced from:

- Angel & Shreiner, "Interactive Computer Graphics: A Top-Down Approach with OpenGL", Addison Wesley
- http://www.opengl-tutorial.org/
- http://ogldev.atspace.co.uk/index.html