UOW AUSTRALIA

SCIT

School of Computing & Information Technology

CSCI336 – Interactive Computer Graphics

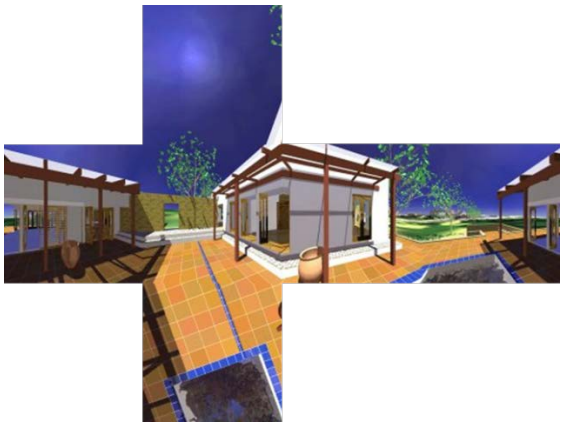
Environment Mapping, Blending, Buffers

In this lab, we will look at using cube environment mapping in OpenGL, as well as blending and buffers for mirror reflection.

Cube Environment Mapping

Environment mapping is a concept where we use textures to create a fake reflection effect. For cube environment mapping, we use 6 images that represent the 6 faces of a cube. When joined together, these images should not have any gaps at the seams.

Open the Lab8a project. In this example, we will use the following images, which can be found in the images folder:



The images are prefixed with "cm_" (i.e. cube map) then named: front, back, left, right, top, and bottom, respectively. If you look open the images files, 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 last lab, you would have seen the Texture class. In this lab, we will use one of the overloaded generate() member functions to load the image for the cube map. The function accepts 6 image file names:

```
void Texture::generate(const std::string fileFront, const std::string fileBack,
       const std::string fileLeft, const std::string fileRight,
       const std::string fileTop, const std::string fileBottom);
```

Have a look at the function in the Texture.cpp file. Using the file names, the function loads the image data of the 6 images using the stb_image library.

```
int width, height, channels;
unsigned char* imageFront = stbi_load(fileFront.c_str(), &width, &height, &channels, 0);
unsigned char* imageBack = stbi_load(fileBack.c_str(), &width, &height, &channels, 0);
unsigned char* imageLeft = stbi_load(fileLeft.c_str(), &width, &height, &channels, 0);
unsigned char* imageRight = stbi_load(fileRight.c_str(), &width, &height, &channels, 0);
unsigned char* imageTop = stbi_load(fileTop.c_str(), &width, &height, &channels, 0);
unsigned char* imageBottom = stbi_load(fileBottom.c_str(), &width, &height, &channels, 0);
```

If the images loaded successfully, the function generates a texture object for the cube map:

```
glGenTextures(1, &mTextureID);
glBindTexture(GL_TEXTURE_CUBE_MAP, mTextureID);
```

Note that the target is GL_TEXTURE_CUBE_MAP, unlike 2D textures where the target is GL_TEXTURE_2D.

Next, the faces of the cube map are created using the image data:

```
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGB, width, height, 0, GL_RGB,
       GL_UNSIGNED_BYTE, imageRight);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 0, GL_RGB, width, height, 0, GL_RGB,
       GL_UNSIGNED_BYTE, imageLeft);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y, 0, GL_RGB, width, height, 0, GL_RGB,
       GL_UNSIGNED_BYTE, imageTop);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, GL_RGB, width, height, 0, GL_RGB,
       GL_UNSIGNED_BYTE, imageBottom);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z, 0, GL_RGB, width, height, 0, GL_RGB,
       GL_UNSIGNED_BYTE, imageBack);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, 0, GL_RGB, width, height, 0, GL_RGB,
       GL_UNSIGNED_BYTE, imageFront);
```

Then the texture parameters are set:

```
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
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);
```

Note that filtering should be GL_LINEAR, and GL_CLAMP_TO_EDGE should be used to avoid any gaps at the seams of the cube map.

Once the cube map texture has been created, the image data is no longer required and can be released.

In the lab8a.cpp file, a Texture object is declared:

```
Texture gCubeEnvMap;
```

In initialised in init() function:

```
gCubeEnvMap.generate(
       "./images/cm_front.bmp", "./images/cm_back.bmp",
       "./images/cm_left.bmp", "./images/cm_right.bmp",
       "./images/cm_top.bmp", "./images/cm_bottom.bmp");
```

Using the cube map is similar to using a texture. In the render_scene() function the shader's cube sampler is set, the texture unit is activated and the texture object is bound:

```
gShader.setUniform("uEnvironmentMap", 0);
glActiveTexture(GL_TEXTURE0);
gCubeEnvMap.bind();
```

Now have a look at the fragment shader, a uniform cube sampler is used for the cube map:

```
uniform samplerCube uEnvironmentMap;
```

In the shader's main() function, the intensity of reflected light is calculated and modulated with the colour of the cube map.

First, the reflection direction must be computed, which is simply the reflection of the viewer's direction with respect to the fragment's normal vector.

```
vec3 reflectEnvMap = reflect(-v, n);
```

Using this, we can lookup the reflection from the cubemap:

```
fColor *= texture(uEnvironmentMap, reflectEnvMap).rgb;
```

Compile and run the program. You will see a mirror like surface, which is blended with the colour of the material.

- Rotate the object using the GUI to observe changes in the reflection.
- Change the material properties to observe changes to the object. If you set the ambient, diffuse and specular components of the material to white, the surface will look more mirror like.

Blending

The next example in the Lab8b project looks at blending. Using blending, we can blend the destination colour (already 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 code in the init() function 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 first function sets how values in the RGB and alpha channels are combined, respectively. The second function sets the blend factor to use for the RGB and alpha channels, respectively.

The following functions (in code for the rendering loop) 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.

```
uniform float uAlpha;
```

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.

```
fColor = vec4(vColor, uAlpha);
```

Compile and run the program.

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

- Toggle blending on and off. You will notice a greater contribution of the objects' colour with blending turned off.
- 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.

Note that Alpha, Rotation and TranslateZ affect the object on the right.

- With blending turned on, change the value of the Alpha component. At 0.0, the object on the right is completely transparent, whereas at 1.0 the object is completely opaque.
- 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, which is 0.1. Toggling the depth test will not make a difference, because if you look at the code, the object on the right is rendered later (the "painter's algorithm").
    - With the depth test on, move the object on the right behind the other object (i.e. change the z-value to a negative value). Notice that the colour of the right object is not blended. The colour of the left object simply overlaps the right object. This is because the left object is rendered and the depth buffer stores its depth values. When the right object is rendered, parts that are behind the left object will fail the depth test and will not be rendered at all.
    - Now disable the depth test. The objects 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.

Mirror Reflection

The program in Lab8c shows an example of adding reflection to a scene by rendering different parts in the colour, depth and stencil buffers. Most of the code in the program should be familiar. The section that creates the reflection effect is in the render_scene() function.

The first thing that we need to do in the render_scene() function is to clear the colour, depth and stencil buffers:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
```

We want the floor to have a mirror reflection effect. Hence, the reflection should only appear on the floor and nowhere else. To do this, we 'tag' the fragments in the stencil buffer. This way, when we draw the reflection, we only draw it to fragments that are 'tagged'.

To 'tag' the fragments in the stencil buffer, we first disable any modification to the colour buffer and depth buffer values:

```
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glDepthMask(GL_FALSE);
```

Next, we enable the stencil buffer:

```
glEnable(GL_STENCIL_TEST);
```

and set the stencil buffer's reference value and operation:

```
glStencilFunc(GL_ALWAYS, 1, 1);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
```

What this does is for anything that is rendered, *always replace* the stencil buffer value with the reference value (set to 1 in this case).

Then, the floor is rendered:

```
draw_floor(1.0f);
```

Note that as modification to the colour buffer and depth buffer values were disabled, rendering the floor only affects the stencil buffer. The stencil buffer was set in a way where any fragment covered by the floor will be tagged with the value of 1, while other values will remain as 0. The value that is passed to the draw_floor() function is an alpha value, which is used for blending. It is set to 1.0f here, because we are not performing blending.

Once the fragments are tagged, we render the reflected geometry only where the stencil buffer test passes. To do this, the stencil buffer operation is set:

```
glStencilFunc(GL_EQUAL, 1, 1);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
```

This means that the stencil buffer test will pass for fragments with a stencil buffer value *equal* to 1. We are *keeping* the stencil values and not overwriting them.

Next, we allow the colour buffer and depth buffer values to be modified:

```
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);
```

Then, draw the reflected geometry (only where the stencil buffer test passes):

```
draw_objects(true);
```

Look at the draw_objects() function. It accepts a Boolean value, which determines whether objects are rendered normally (false) or reflected (true). The code in the function starts by checking this value. If it is true, a reflection matrix is calculated.

```
glm::mat4 reflectMatrix = glm::mat4(1.0f);
glm::mat4 modelMatrix = glm::mat4(1.0f);
glm::vec3 lightPosition = gLight.pos;
...
if (reflection) {
	reflectMatrix = glm::scale(glm::vec3(1.0f, -1.0f, 1.0f));
```

The reflection matrix in this example is about the horizontal plane, as the floor is located where the *y*-axis = 0.0f.

When rendering the reflected geometry, the light's position must also be flipped:

```
	lightPosition = glm::vec3(reflectMatrix * glm::vec4(lightPosition, 1.0f));
}
```

The rest of the code in the function uses a shader and sets the shader uniform variables before rendering objects. Some highlights are that the light position is either the non-reflected light position or the reflected light position:

```
	gShader.setUniform("uLight.pos", lightPosition);
```

the model matrix is either the non-reflected model matrix or the reflected model matrix:

```
	modelMatrix = reflectMatrix * gModelMatrix["Cube"];
```

Returning to the render_scene() function, after the reflected geometry is rendered, we no longer need the stencil buffer:

```
glDisable(GL_STENCIL_TEST);
```

The next thing we want to do is to render the floor and blend it with the reflection. We enable blending and set the blend equation:

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

Then, we render the floor, which is blended with the reflection based on an alpha value.

```
draw_floor(gAlpha);
```

Look at the code in the fragment shader. It takes a uniform alpha value as input:

```
uniform float uAlpha;
```

It also outputs colour as a vec4, i.e. RGBA, as we use the alpha value for blending:

```
out vec4 fColor;
```

The uniform alpha value is used as the fourth component of the output colour:

```
	fColor = vec4(Ia + Id + Is, uAlpha);
```

Now return to the render_scene() function. After the floor is blended with the reflection, we disable blending as it is no longer require:

```
glDisable(GL_BLEND);
```

Finally, we render the non-reflected geometry:

```
draw_objects(false);
```

Compile and run the program.

You will see the reflection of a cube on the floor. The code uses the Camera class that was described in a previous lab. You can move the camera around using the WASD keys and moving the mouse while holding down the right mouse button. Notice that the reflection is only rendered on the floor. Try changing the blend value. This controls the amount of mirror reflection.

Exercises

Try modifying the programs to do the following:

- Add a GUI element that will allow the user to change the amount of reflection on the object.
- Load different models and render these using the cube environment mapping. Enable back face culling.

References

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

- Angel & Shreiner, “Interactive Computer Graphics: A Top-Down Approach with OpenGL”, Addison Wesley