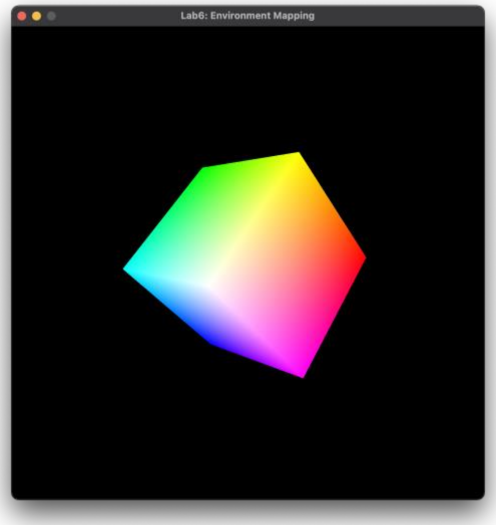
CSCI 444/544 - Lab 6
Friday, March 5, 2021

LAB IS DUE BY **FRIDAY March 19 11:59 PM**!!

This lab will have you create a cube map and compute reflection rays. The reflection technique can be added on top of our Blinn-Phong Illumination Model to augment the specular highlight. You can also add in noise to make a rusty object that is only reflective in some parts.

Start by running the program. We get a floating cube in space. There will be uniforms currently not found, but we'll be adding those in as we progress.



Step 1 – Generate a Skybox

Part I – Create the Cube Map

We have to first load in a cubemap. In `main.cpp` at `TODO 1`, begin by generating the premade `cubeMapTextureHandle` variable using `glGenTextures()`. Immediately after, we need to bind the cube map:

```
glBindTexture(GL_TEXTURE_CUBE_MAP, cubeMapTextureHandle);
```

Next, at `TODO 2` we need to load each of the cube map face textures. You have two options for images. They are located in the `assets/textures/` folder in either the `colosseum/` or `mountainsLake/` folder. Choose your favorite scene and use the helper function `loadCubeMapFaceTexture()`. The first argument is the cube face that the image corresponds to (such as `GL_TEXTURE_CUBE_MAP_POSITIVE_X`) and the name of the file to load. Be sure to map `right/left/front/back/top/bottom` to `posX/negX/negZ/posZ/posY/negY` appropriately.

Then at `TODO 3` set 5 parameters using the `glTexParameteri()` method. The first argument is always our target, in this case `GL_TEXTURE_CUBE_MAP`. The second and third arguments are the property and value. Use the following table to fill in these calls:

PROPERTY	**VALUE**
GL_TEXTURE_MAG_FILTER	GL_LINEAR
GL_TEXTURE_MIN_FILTER	GL_LINEAR
GL_TEXTURE_WRAP_S	GL_CLAMP_TO_EDGE
GL_TEXTURE_WRAP_T	GL_CLAMP_TO_EDGE
GL_TEXTURE_WRAP_R	GL_CLAMP_TO_EDGE

And that is all we need to do to load in the cube map.

Part II – Draw the Skybox

If you follow through the setup for the skybox VAO in `setupBuffers()`, you'll notice that it is made up of a single vertex located at the origin. We're going to play a trick. We'll process a single vertex to kick off our pipeline.

Then in our geometry shader, we'll create a quad that is always oriented to our camera and at the far clip plane. Working directly in clip space will make this easy.

Look at `shaders/skybox.v.glsl`. It is perhaps the most boring vertex shader we've ever written. But it isn't supposed to do anything. We will create our geometry procedurally in the geometry shader.

Now look at `shaders/skybox.g.glsl`. The geometry shader expects a point as input and outputs a triangle strip made up of 4 vertices – that will equate to a quad when wound properly.

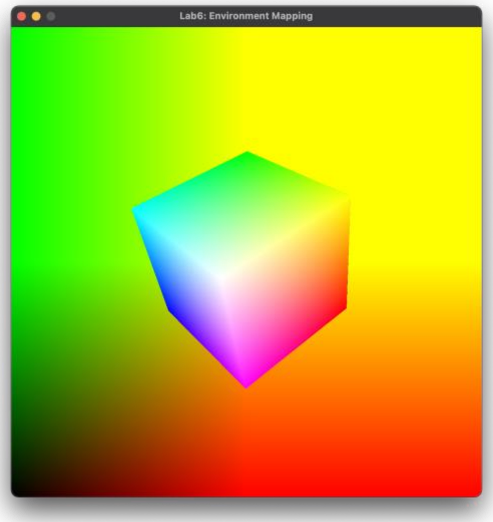
Let's first just make the quad. At each of `TODO A, B, C, D` we need to do the following:

1. set `gl_Position` equal to the corresponding point
2. emit the vertex

The z-coordinate for all the positions will be -1 which places it at the far clip plane. Then we'll match up the top coordinates to be a Y value of 1, bottom Y = -1. The right coordinates to be an X value of 1, left X = -1.

Quickly jump over to `shaders/skybox.f.glsl`. You'll notice we're setting the fragment color to be the fragment coordinate. That's ok for now.

Run your program and you should see an image similar to below:



Perfect, let's add the actual texture to it. To do a texture lookup, we need a texture coordinate.

At `TODO E, F` create the varying for the texture coordinate. Since we are working with cube maps, this needs to be a three dimensional value.

The first step, at `TODO G`, is to compute the camera basis. We are passing in the values used to compute the view matrix, so we need to do the math to generate the axes. We can compute them in the following order with the following equations:

```
Z = eyePosition - lookAtPoint
X = Z x Up
Y = X x Z
```

Where `x` represents cross product. Make sure each axis is normalized.

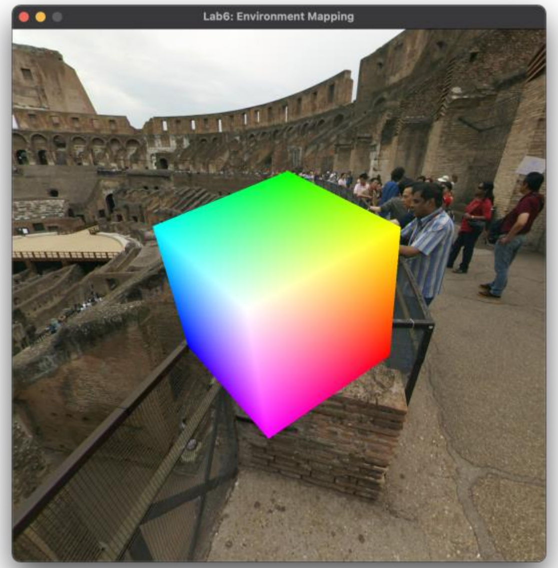
Again at `TODO A, B, C, D` we need to set our texture coordinate output. The texture coordinate will represent the direction from the camera to the corner of our frustum. The directions are a summation of our camera axes.

Corner	Direction
`Top Right`	`-Z + Y + X`
`Top Left`	`-Z + Y - X`
`Bottom Left`	`-Z -Y - X`
`Bottom Right`	`-Z - Y + X`

Now for the ultimate step, in the fragment shader at `TODO H` perform a texture lookup against the cube map with our texture coordinate and assign it to the fragment out color.

`texture(cubeMap, texCoord)` returns a `vec4`. We need only the RGB channels and can set A to 1.

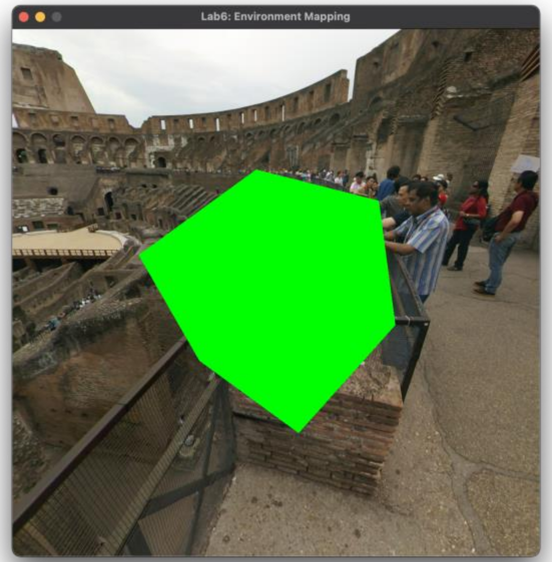
Now run and if everything went right, you'll see the following



Enjoy looking around the world!

Step 2 – Reflect the Skybox

Now, let's make our cube chrome and shiny. These modifications will go in `shaders/reflect.v.glsl` and `shaders/reflect.f.glsl`. Currently, the shaders are just transforming the point and assigning the color to green. When you run your program, press `'r'` to turn on reflection and see these shaders in action.



Ok, we know they're working. The fragment shader will need to know the world position and world normal of our object to perform the ray reflection.

In the vertex shader at `TODO I` compute these two values and pass them as a varying to the fragment shader.

The position is the XYZ components of the result of taking our vertex position and multiplying it by the model matrix.

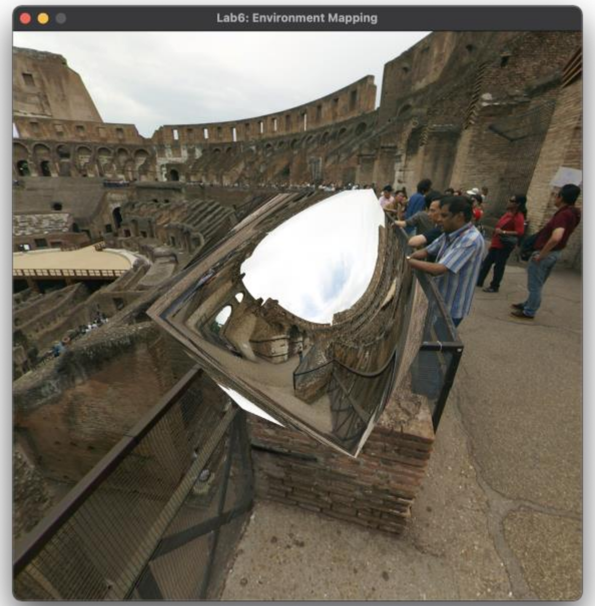
The normal needs to be transformed by the normal matrix.

The final two steps will occur in the fragment shader. At `TODO J` we'll first compute the incident ray. This is the ray that originates at the camera and hits the object. This is computed by `position - eyePoint`.

Now we can compute the reflected ray by using the `reflect()` function and passing in the incident ray and the normal.

The final step at `TODO K` is to again do a texture lookup against our cube map and supply the reflected ray as our texture coordinate.

Run, press `'r'` and look at it go!



For some added fun, press `'c'` when reflection is on.

LAB IS DUE BY **<u>FRIDAY March 19 11:59 PM</u>**!!