Exercise 1. Cube mapping

Part 1: Draw the environment

1. Start from the default OpenGL Position effect.
2. Add a cubemap texture to the effect.
3. Rename pass 0 as **Environment** and add a **texture object** to the pass and rename it as "Sky"
4. Open vertex shader program and add following variables:
   ```
   uniform vec4 EyePosition;
   varying vec3  vDir;
   ```
5. Find the view direction vDir in the body of the vertex shader:
   ```
   … …
   vDir = gl_Vertex.xyz;
   vec3 Pos2beViewed = EyePosition.xyz + vDir ;
   gl_Position = gl_ModelViewProjectionMatrix * vec4(Pos2beViewed, 1.0);
   … …
   ```
6. In the fragment shader, specify the fragment colour using the colour of cubemap corresponding to the view direction vDir:
   ```
   uniform samplerCube Sky;
   varying vec3 vDir;

   void main(void)
   {
       gl_FragColor = textureCube( Sky, vDir);
   }
   ```
7. Save and recompile your shader programs. You may need to specify cull mode as none or front to be able to view the environment you would like to draw. To specify cull mode, add Render State to the pass and edit the states.

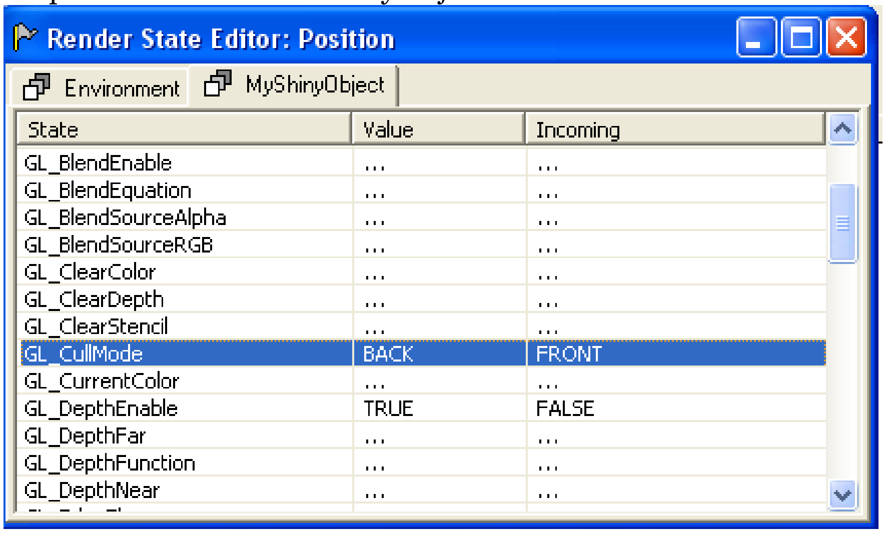
Part 2: Draw reflected environment mapped object

1. Add a default pass as the second pass to your effect and rename as MyShinyObject.
2. Add the cubemap texture object to the pass.
3. Similar to the way you implement per-fragment lighting, add necessary variables to both the vertex shader and fragment shader to find the reflection direction in the fragment shader. For example,
   ```
   uniform vec4 EyePosition;

   varying vec3 Normal;
   varying vec3 vDir;
   ```

```
void main(void)
{
  gl_Position = ftransform();
  vDir = EyePosition.xyz-gl_Vertex.xyz;
  Normal = gl_Normal;
   … …
}
```

4. The fragment shader of the pass should be similar to the one in the first pass (see point 6 in Part 1), except that you should use reflected direction to fetch the colour from the cubemap texture. So at least the following two lines of code need to be added:
   a) varying vec3 Normal;
   b) vec3 reflectView=reflect(vDir, Normal);
5. Save your programs and recompile. You need to edit the render states of the two passes to view the shiny object:



State	Value	Incoming
GL_BlendEnable	...	...
GL_BlendEquation	...	...
GL_BlendSourceAlpha	...	...
GL_BlendSourceRGB	...	...
GL_ClearColor	...	...
GL_ClearDepth	...	...
GL_ClearStencil	...	...
GL_CullMode	BACK	FRONT
GL_CurrentColor	...	...
GL_DepthEnable	TRUE	FALSE
GL_DepthFar	...	...
GL_DepthFunction	...	...
GL_DepthNear	...	...

Part 3: Draw refracted environment mapped object.

In this part, add refracted effect to make your object look translucent.

Exercise 2. Texture based animation

1. Start from a default OpenGL **Screen-Aligned Quad** effect.
2. Add a variable myTimer to the fragment shader and set the variable's semantic as the predefined time variable **time0_X** with default cycle period of 120 second. You can change the cycle period with any positive integer number to control the cycling of the time.

3. Specify a way to alter the incoming texture coordinate, for example, to make the texture texels move with velocity **v** by altering the texture coordinates in the following way

```
uniform float myTimer;
uniform vec2 v;

… …

void main(void)
{
   gl_FragColor = texture2D( Texture0, texCoord + v*myTimer);
}
```

4. Use the idea in step 2 to implement animate real world phenomenon, such as raining or snowing.

Exercise 3. Pulsating Objects

1. Open a default sample effect and change the model to whatever object you would like to use.
2. Add a variable **time** to the vertex shader and set the variable's semantic as the predefined time variable **time0_X**:
```
uniform float time;
```
3. Add float variables S and **freq** in the vertex program to specify how you would change the size of the object. You can start with some functions that have been considered in the lecture. For example,
```
float disp = S*sin(freq * time );
```
4. Input normal into the vertex shader program and scale it using the variable defined in step 3.
5. Translate the vertex position of the object by the vector **disp* Normal** before transform it into the clip space.
6. Save and recompile your programs. You should see a pulsating object. You may need to set the values for variables **S** and **freq** to be nonnegative.
7. Modify the function disp used in step 3 to create waving effects.

Exercise 4. Particle fire

1. As with exercise 1 of this part, you can do this exercise starting from a default sample code provided in RenderMonkey. Instead of using a collection of particles, you can begin with just one particle by loading a single model. You need to refer to my lecture notes to understand the basic principles on how to implement a particle system based effect in Rendermonkey in GLSL.
2. Scaling the size of the object into proper size.

3. Add a timer to the vertex shader in the way shown in step 2 of doing Exercise 3:

   ```
   uniform float time;
   ```
4. Specify a shooting velocity **v** for the object.
5. Translate the object's position along that direction using a function of time similar to the one used in the program for animating a pulsating object.
6. Alter the shooting velocity by modifying the time variable with an variable **alpha: time^{alpha}**
7. Then consider how to fade the colour of the object. It should be brightest at the time of its birth and completely dark at the time of its death.
8. Now texture mapping the object with a texture representing fire.
9. When you know how to emit one single particle and know how to control its lifetime and to fade its colour, you can replace the model with QuadArray model provided in Rendermonkey, which provides 100 rectangles differed by its **z** values.
10. Billboarding each rectangle in the QuadArray.
11. Reposition the quads within the array and define a function to specify how you want to shoot them.

    Refer to the Fire_Particle_System_OpenGL effect shown in RenderMonkey Particle System sample when you are not sure how to do this exercise.

==
Cantact: q.li@hull.ac.uk Dr Qingde Li
Department of computer science University of Hull