[index]

Cube Maps: Sky Boxes and Environment Mapping

Anton Gerdelan. Last Updated 2 October 2016

OpenGL has a special kind of texture for cubes that allows us to pack 6 textures into it. It has a matching sampler in GLSL that takes a 3d texture coordinate - with R, S, and T, components. The trick here is that we use a 3d direction vector for the texture coordinates. The component with the biggest magnitude tells GL which of the 6 sides to sample from, and the balance between the remaining 2 components tells GL where on that 2d texture to sample a texel from.

Two popular techniques that make use of cube maps are **sky boxes**, which provide the illusion of a 3d background behind the scenery, and **environment mapping**, which can make a very-shiny surface (think lakes, chrome, or car paint) appear to **reflect the environment** scenery. We can also use it to approximate **refraction** for translucent materials like water.

Sky Box

This technique effectively replaces the GL background colour with a more interesting 3d "painted backdrop". The idea with the sky box technique is to have a big box encase the camera as it moves around. If the box appears to be the same distance even when the camera is moving, then we get a **parallax** effect - the illusion of it being relatively very far away. We don't rotate it as the camera rotates though - this allows us to pan around. If the texture has been well constructed then we won't notice the seams and it will have been projected onto the box so that it **appears as if we are inside a sphere** when we sample it with our view 3d vector. We can actually use a dome or a sphere instead of a box, but boxes are easier to texture map.

The size of the box is not going to affect how big it looks. Think about it - if it's moving with the camera it will look exactly the same close up as far away. The size only matters if it intersects with the clip planes - in this case you will see a plane cutting off part of your box. We will **draw the box before anything else**, and turn off **depth-masking**. Remember that this means that it doesn't write anything into the depth buffer - any subsequent draws to the same pixel will draw on top of it. Then it is never drawn in front of your scenery, even if it is closer to the camera.

Make a Big Cube

We need to make a big cube and put it into a vertex buffer. We don't need to add texture coordinates. We are going to be inside the cube, so make sure that you get the winding order of the vertices so that the "front" faces are on the inside. You can load a cube from a mesh file if you like, but I think that's cheating.

```
float points[] = {
  -10.0f,  10.0f, -10.0f,
  -10.0f, -10.0f, -10.0f,
   10.0f, -10.0f, -10.0f,
   10.0f, -10.0f, -10.0f,
   10.0f,  10.0f, -10.0f,
  -10.0f,  10.0f, -10.0f,

  -10.0f, -10.0f,  10.0f,
  -10.0f, -10.0f, -10.0f,
  -10.0f,  10.0f, -10.0f,
  -10.0f,  10.0f, -10.0f,
  -10.0f,  10.0f,  10.0f,
  -10.0f, -10.0f,  10.0f,

   10.0f, -10.0f, -10.0f,
   10.0f, -10.0f,  10.0f,
   10.0f,  10.0f,  10.0f,
   10.0f,  10.0f,  10.0f,
   10.0f,  10.0f, -10.0f,
```

Create a Cube-Map Texture

Now, we need to make or find a suitable set of textures for the sky box. There are some great examples at Humus' (Emil Persson's) site: http://www.humus.name/index.php?page=Textures, and many older games will have sky box textures that you can experiment with. You might need to swap the "front" and "back" textures around if they are intended for the opposing coordinate system. Cube maps are sometimes stored inside a single texture file, but more often than not, as 6 separate textures. I'll assume that we are loading from 6 separate textures here. Note that we don't generate mip-maps for a sky box because it's always going to be much the same distance from the camera, and we can just choose an appropriately-sized texture. To test my cube I drew a different coloured border onto each textured face to show where the seams were and check if I needed to swap the front and back textures.

I have a `create_cube_map()` function which takes file names for the 6 separate image files, and generates the opengl texture. This will call another function to load the separate image for each of the 6 sides. Note that each side is going to bind to the same texture that we generated - we will pass this as a parameter. Finally we apply some texture filtering to the texture, and the texture handle is passed back as a function parameter.

```
void create_cube_map(
  const char* front,
  const char* back,
  const char* top,
  const char* bottom,
  const char* left,
  const char* right,
  GLuint* tex_cube) {
  // generate a cube-map texture to hold all the sides
  glActiveTexture(GL_TEXTURE0);
  glGenTextures(1, tex_cube);

  // load each image and copy into a side of the cube-map texture
  load_cube_map_side(*tex_cube, GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, front);
  load_cube_map_side(*tex_cube, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, back);
  load_cube_map_side(*tex_cube, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, top);
  load_cube_map_side(*tex_cube, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, bottom);
  load_cube_map_side(*tex_cube, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, left);
  load_cube_map_side(*tex_cube, GL_TEXTURE_CUBE_MAP_POSITIVE_X, right);
  // format cube map texture
```

The texture filtering parameters are interesting. Make sure that you set "clamp to edge" for all three components. If you don't clamp to edge then you might get a visible seam on the edges of your textures (we'll see why shortly). We have some bi-linear filtering to clean up any minor aliasing when the camera rotates. The function that loads each image is very similar to the one that we wrote in the texture mapping tutorial, and also uses Sean Barrett's image loader (stb_image):

```
bool load_cube_map_side(
  GLuint texture, GLenum side_target, const char* file_name) {
  glBindTexture(GL_TEXTURE_CUBE_MAP, texture);

  int x, y, n;
  int force_channels = 4;
  unsigned char*  image_data = stbi_load(
    file_name, &x, &y, &n, force_channels);
  if (!image_data) {
    fprintf(stderr, "ERROR: could not load %s\n", file_name);
    return false;
  }
  // non-power-of-2 dimensions check
  if ((x & (x - 1)) != 0 || (y & (y - 1)) != 0) {
    fprintf(stderr,
        "WARNING: image %s is not power-of-2 dimensions\n",
        file_name);
  }

  // copy image data into 'target' side of cube map
```

The code is almost identical to creating a single 2d texture in OpenGL, except that we bind a texture of type **GL_TEXTURE_CUBE_MAP** instead of **GL_TEXTURE_2D**. Then **glTexImage2D()** is called 6 times, with the macro for each side as the first parameter, rather than **GL_TEXTURE_2D**.

Basic Cube-Map Shaders

Our vertex shader just needs position the cube and output texture coordinates. In this case a 3d coordinate as a **vec3**. This should be a direction, but I can use the actual local vertex points as a direction because it's a cube. If you think about it, the points along the front face are going to be interpolated with varying values of x and y, but all will have z value -10.0 (or the biggest size in the cube). You don't need to normalise these first - it will still work. The only problem is that the points

exactly on the corners are not going to have a "biggest" component, and pixels may appear black as the coordinate is invalid. This is why we enabled clamp-to-edge.

Note: I've seen other tutorials that manipulate the Z and W components of `gl_Position` such that, after perspective division the Z distance will always be 1.0; maximum distance in normalised device space, and therefore always in the background. This is a bad idea. It will clash, and possibly visibly flicker or draw all-black, when depth testing is enabled. It's better to just disable depth masking when drawing, as we will do shortly, and make sure that it's the first thing that you draw in your drawing loop.

```
#version 400

in vec3 vp;
uniform mat4 P, V;
out vec3 texcoords;

void main() {
  texcoords = vp;
  gl_Position = P * V * vec4(vp, 1.0);
}
```

The P and V matrices are my camera's projection, and view matrices, respectively. The view matrix here is a special version of the camera's view matrix that **does not contain the camera translation**. Inside my main loop I check if the camera has moved. If so I build a very simple view matrix and update its uniform for the cube map shader. This camera only can only rotate on the Y-Axis, but you might use a quaternion to generate a matrix for a free-look camera. Remember that view matrices use negated orientations so that the scene is rotated around the camera. Of course if you are using a different maths library then you will have different matrix functions here.

```
if (cam_moved) {
  mat4 R = rotate_y_deg(identity_mat4(), -cam_heading);
  glUseProgram(shader_programme);
  glUniformMatrix4fv(V_loc, 1, GL_FALSE, R.m);
}
```

The fragment shader uses a special sampler; `samplerCube` which accepts the 3d texture coordinate. Other than that there is nothing special about the fragment shader. Use the `texture()` function, but older GL implementations may prefer `textureCube()`.

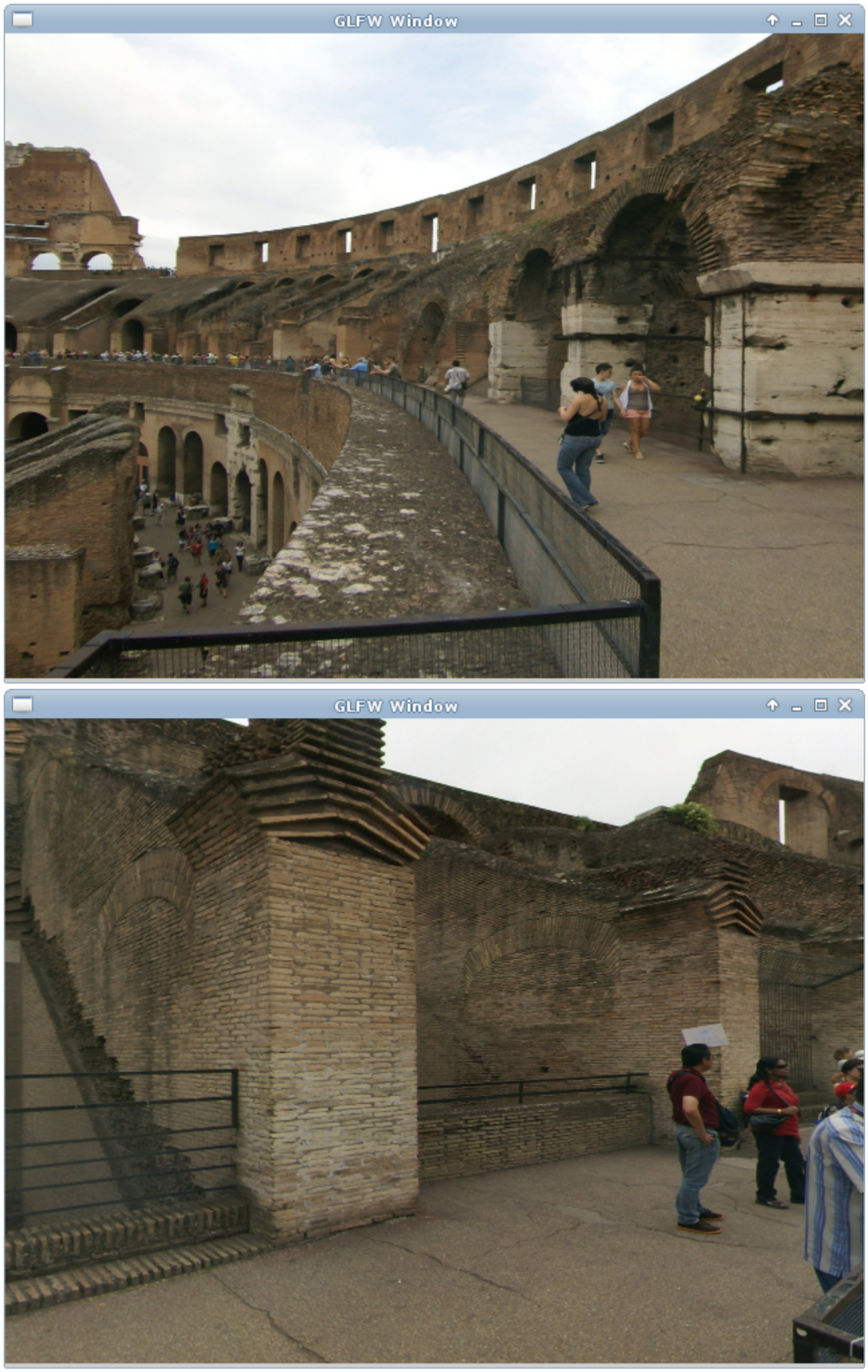
```
#version 400

in vec3 texcoords;
uniform samplerCube cube_texture;
out vec4 frag_colour;

void main() {
  frag_colour = texture(cube_texture, texcoords);
}
```

Rendering

Use your shaders, and bind your VAO as usual. Remember that we should bind the texture into an active slot before rendering. This time it has a different texture type; `GL_TEXTURE_CUBE_MAP`. With depth-masking disabled it won't write anything to the depth buffer; allowing all subsequently drawn scenery to be in

front of the sky box.

```
glDepthMask(GL_FALSE);
glUseProgram(shader_programme);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, tex_cube);
glBindVertexArray(vao);
glDrawArrays(GL_TRIANGLES, 0, 36);
glDepthMask(GL_TRUE);
```



I found this set of sky box textures on Emil Perssons' website, which have a Creative Commons Attribution licence, so they are excellent for demos. (top) points to a corner of the box, viewing the left and front textures, (bottom) points to another corner, viewing the front, and right textures. I suggest testing it with a camera that can rotate.

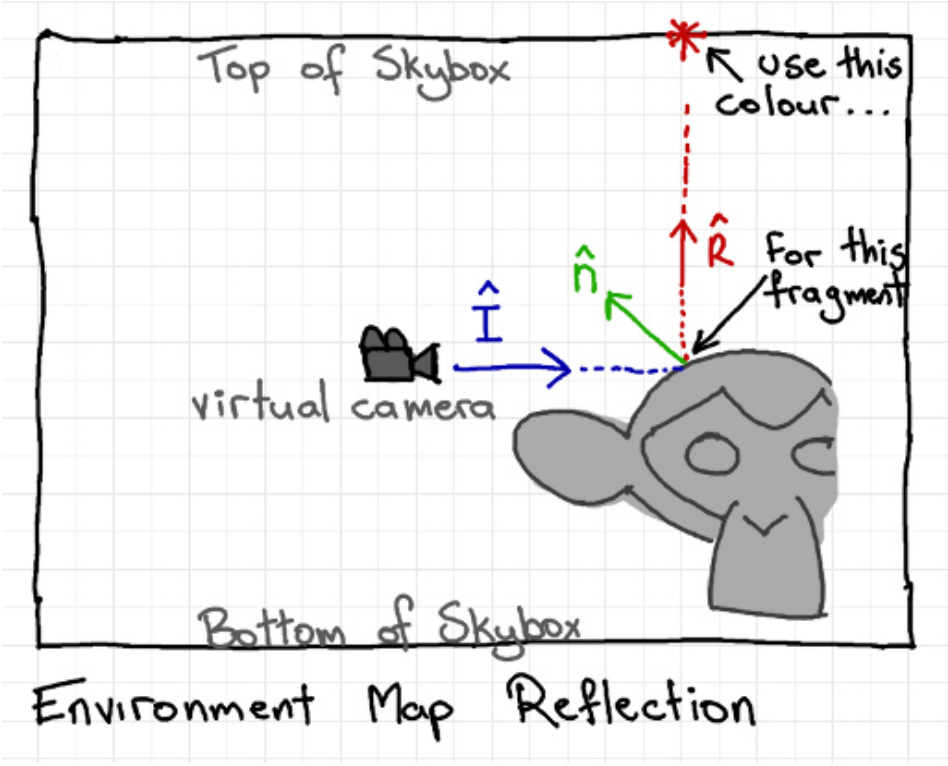
Reflection and Refraction with Static Environment

Maps

So far we have only used specular highlights to show if a surface is highly reflective, which appears as a brightly-coloured dot. Suppose we want to reflect a more detailed image of the sky box or part of the environment onto the surface. If we have a cube map with an image of a brightly-coloured square window on it, then we can use this to create a much more interesting reflection of light on the surface.

Reflecting the Sky Box

The most common type of environment map just reflects a scene's sky box onto a surface. We can say that this is a "static" environment map because it won't reflect any moving pieces of the scene. It can produce quite a convincing effect as you can see that the sky box matches the reflection as you rotate the view. You don't need to have a sky box displayed for this to work though - an indoor scene might not have any visible sky, but you might have a cube map that is not displayed, but contains an indoor-like cube map which you only use for reflections.



With the incident direction vector from the camera to the surface position, **I**, we calculate a reflection R around the surface normal N. We use R to sample the cube map, and the texel is then applied to the fragment of the surface.

The principle is very simple. We work out a direction vector from the view point (camera position) to the surface. We then reflect that direction off the surface based on the surface normal. We use the reflected direction vector to sample the cube map. GLSL has a built-in `reflect()` function which takes an input vector and a normal, and produces the output vector. I had some trouble getting this to compile on all drivers, so I prefer to code it manually.

I loaded a mesh of the Suzanne monkey head, and created a vertex simple shader for it. I calculate the vertex positions and normals in eye space.

```
#version 400

in vec3 vp; // positions from mesh
in vec3 vn; // normals from mesh
uniform mat4 P, V, M; // proj, view, model matrices
out vec3 pos_eye;
out vec3 n_eye;

void main() {
  pos_eye = vec3(V * M * vec4(vp, 1.0));
  n_eye = vec3(V * M * vec4(vn, 0.0));
  gl_Position = P * V * M * vec4(vp, 1.0);
}
```

The fragment shader calculates the direction vector from the camera to the surface as `incident_eye`; the incident direction in eye space. Note that this is normally `vec3 dir = normalize(to - from)` but the "from" here is the camera position (0,0,0) in eye space. The built-in reflection function then gets the reflected vector, which is converted to world space and used as the texture coordinates.

```
#version 400

in vec3 pos_eye;
in vec3 n_eye;
uniform samplerCube cube_texture;
uniform mat4 V; // view matrix
out vec4 frag_colour;

void main() {
  /* reflect ray around normal from eye to surface */
  vec3 incident_eye = normalize(pos_eye);
  vec3 normal = normalize(n_eye);

  vec3 reflected = reflect(incident_eye, normal);
  // convert from eye to world space
  reflected = vec3(inverse(V) * vec4(reflected, 0.0));

  frag_colour = texture(cube_texture, reflected);
}
```

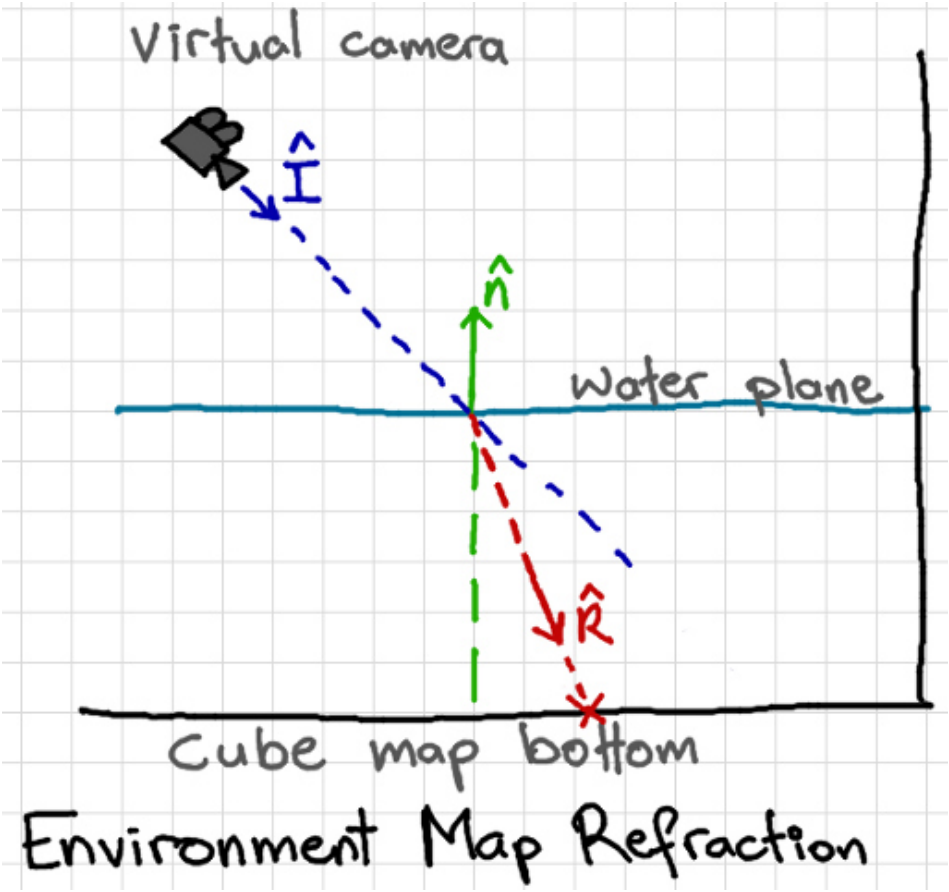
You can reduce the number of instructions in the shader if you compute the reflection in world space, but then you need a camera world position uniform.

Environment maps reflecting the sky box onto a mesh. Using a highly detailed cube map texture will significantly improve the quality of the effect.

Common Mistakes

- My reflection is upside-down or the wrong side! - Check the direction of your incident and normal vectors. Normalise them.
- My mesh reflections are not smooth like your monkey! - Remember to use interpolated surface normals (set all faces to smooth normals in Blender).

Refraction



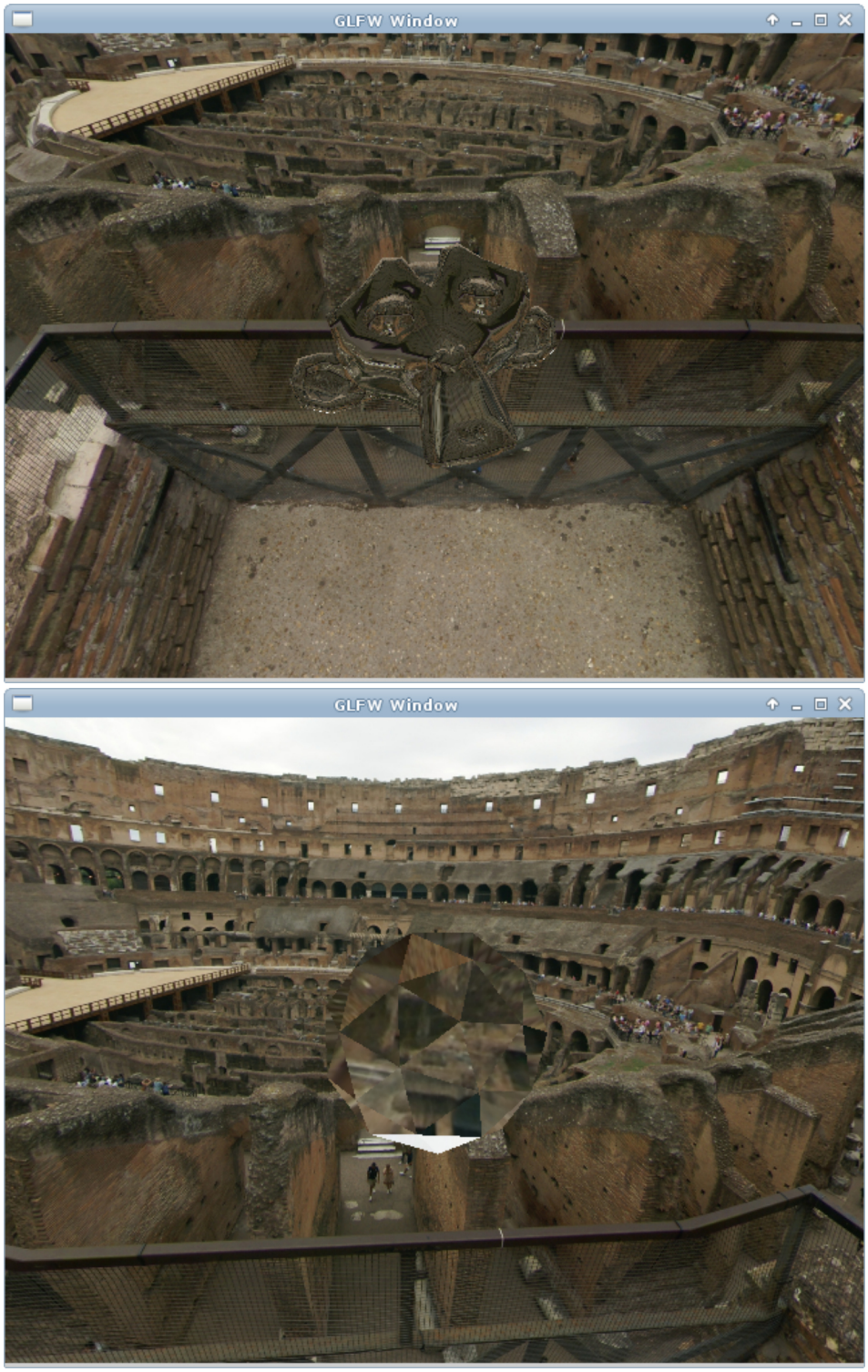
With the incident direction vector from the camera to the surface position, **I**, we calculate a refraction R which modifies the incident vector based on the surface normal, N, and the ratio of refractive indices, which is the first index (1.0 for air), divided by the second index (1.3333 for water). With the above example, you might use a cube map that displays the interior of a swimming pool, rather than the sky box.

Translucent objects should **refract** light. This means that it doesn't go straight through; the ray bends when it hits the surface and changes material. Looking into a swimming pool, for example. This works much the same way as reflection, except that we perturb the eye-to-surface vector about the surface normal, instead of reflecting it. The actual change in direction is governed by Snell's Law. GLSL also has a built-in `refract()` function with a vector-based version of Snell's Law. You can modify the previous fragment shader to use this instead of `vec3 reflected`:

```
float ratio = 1.0 /1.3333;
vec3 refracted = refract(incident_eye, normal, ratio);
refracted = vec3(inverse(V) * vec4(refracted, 0.0));
```

Where the `ratio` is the ratio between the **refractive index** of the first material (air, 1.0), and the second material (water, 1.3333). You can find some indices for common materials on Wikipedia. Note that the monkey head is single-sided; we are ignoring the effect of the light traveling through the back of the head, where it should refract a second time. So you can say that it's not very realistic, but fine for simple surfaces.


GLFW Window
GLFW Window



Refraction using indices for air (top), water (second), and cubic zirconia (bottom two).

You can imagine that with the right combination of Phong, vertex displacement, and refraction environment map, that you could make a pretty convincing water visualisation.

Common Mistakes

- My refracted image looks magnified or blocky! - Try using higher-resolution cube map textures.
- My image is upside-down! - This is probably correct. Look at photos of refractions of real, similar surfaces (i.e. sphere, pool). Check that your index ratio is correct (first / second).

And For Experts...Dynamic Environment Maps

It is possible to use the entire scene, including all the moving objects, in an environment map. If you are already comfortable with binding textures to framebuffers (see later tutorials) then you can bind 6 textures as target outputs from 6 framebuffers. You will render the scene 6 times from the point of view of the object that is going to be environment-mapped. Once pointing up, once down, once left, etc. The output textures will be combined into the cube map. The perspective matrix should have a field of view of 90 degrees so that the 6 frustums line up exactly into a cube.

1. Create 6 colour textures
2. Create 6 depth textures
3. Create 6 framebuffers
4. Attach a colour texture and depth map to each framebuffer
5. When rendering bind each framebuffer and draw in 1 of 6 fixed directions
6. Create a cube map from 6 textures

We looked at how to manually create a cube-map at the beginning of the article. It's also possible to attach all 6 depth and colour maps to a single framebuffer and render all 6 views in a single pass. To do this you will need to use a geometry shader that redirects the vertex positions to separate `gl_Layer`s. This might give you a small computational improvement if you are computing dynamic environment maps in real-time, but I find geometry shaders to be unstable across the range of drivers and would not recommend this option.