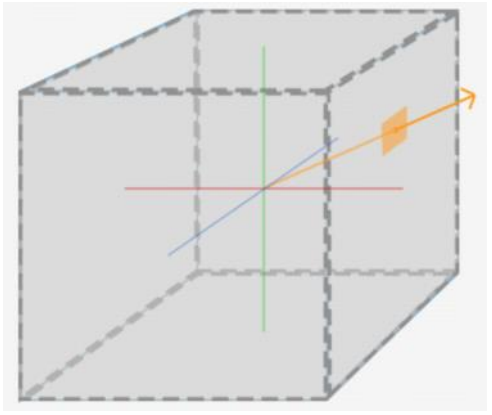
CUBEMAPS



In this chapter, we'll discuss a texture type that is a combination of multiple textures mapped into one: a **cubemap**.

A cubemap is a texture that contains 6 individual 2D textures that each form one side of a cube (a textured cube).
- Cubemaps are useful because they can be indexed/sampled using a direction vector.
  - Imagine we have a 1x1x1 unit cube with the origin of a direction vector residing at its center. Sampling a texture value from the cubemap with an orange direction looks like the image to the right.
  - **NOTE:** The magnitude of the direction vector does not matter. OpenGL retrieves corresponding texels that the direction (eventually) hits and returns the properly sampled texture value.
  - If we imagine we have a cube shape that we attach such a cubemap to, this direction vector would be similar to the (interpolated) local vertex position of the cube.
    - This way, we can sample the cubemap using the cube's actual position vectors as long as the cube is centered on the origin. We thus consider all vertex positions of the cube to be its texture coordinates when sampling a cubemap. The result is a texture coordinate that accesses the proper individual face texture of the cubemap.

Creating a Cubemap

A cubemap is like any other texture, so you can generate and bind it the same way you would a texture, except you'll bind to `GL_TEXTURE_CUBE_MAP`.

Because a cubemap contains 6 textures, one for each face, we have to call `glTexImage2D` six times with their parameters set similarly to the previous chapters. However, this time, we have to set the texture target parameter to match a specific face of the cubemap so OpenGL knows which side of the cubemap we're creating the texture for.

Since we have six faces, OpenGL gives us six special texture targets for targeting a face of the cubemap, as shown in the image to the right.

Texture target	Orientation
`GL_TEXTURE_CUBE_MAP_POSITIVE_X`	Right
`GL_TEXTURE_CUBE_MAP_NEGATIVE_X`	Left
`GL_TEXTURE_CUBE_MAP_POSITIVE_Y`	Top
`GL_TEXTURE_CUBE_MAP_NEGATIVE_Y`	Bottom
`GL_TEXTURE_CUBE_MAP_POSITIVE_Z`	Back
`GL_TEXTURE_CUBE_MAP_NEGATIVE_Z`	Front

- **NOTE:** As with many of OpenGL's enums, these enums are linearly incremented, meaning you could easily loop through all texture targets by adding to `GL_TEXTURE_CUBE_MAP_POSITIVE_X`.
  - Example: `GL_TEXTURE_CUBE_MAP_POSITIVE_X + 1 = GL_TEXTURE_CUBE_MAP_NEGATIVE_X`

Generate and bind a cubemap. Then, use `stbi_load` to load a texture for each face of the cubemap and configure the relevant texture properties. Finally, write a fragment shader to render the cubemap.
- Cubemap Setup

A **skybox** is a common application of a cubemap.

Skybox

A skybox is a (large) cube that encompasses the entire scene and contains 6 images of a surrounding environment, giving the player the illusion that the environment they're in is much larger than it actually is.
- Skyboxes can consist of varying things, such as mountains, clouds, or a night sky.

You can download the necessary images for a skybox online, or you can use the textures in the zipped folder provided below.

Loading a Skybox

Load the skybox textures of your choice into a cubemap.
- Skybox Texture Loading

Displaying a Skybox

For this next part, using this set of vertex data for the skybox: Sample Skybox Vertices

Create the VAO and VBO for the skybox object. Then, draw the skybox first in the scene and make sure that it does not get translated when the player moves.
- Skybox Drawing

An Optimization

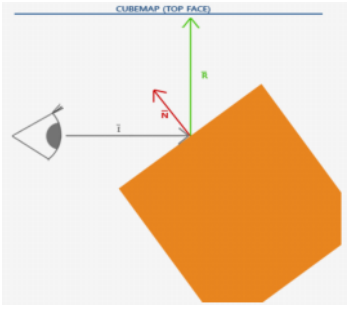
Right now, the entire skybox is being drawn, no matter if something in the scene is in front of it. We're running the fragment shader for each pixel on the screen even though only a small part of the skybox will be visible. The hidden fragments could have easily been discarded using early depth testing, saving us valuable bandwidth.

Instead of rendering the skybox first, we're going to render it last so that we only have to render the skybox's fragments when early depth testing passes, greatly reducing the number of fragment shader calls.
- This is tricky, though. Because the skybox is just a `1x1x1` cube centered on the camera, its fragments will almost always succeed depth tests. Disabling depth testing when drawing the skybox won't work because the skybox is drawn last, so it will hide everything else in the scene.
- We need to trick the depth buffer into believing that the skybox has the maximum depth value of `1.0` so that it fails the depth test whenever there's another object in front of it.

We can solve the depth issue by setting `gl_Position`'s `z` component to equal its `w` component.
- We know that perspective division is performed after the vertex shader has run, dividing `gl_Position`'s xyz coordinates by its `w` component.
- We also know that the `z` component of the resulting perspective division is equal to that vertex's depth value.
- So, if we set the `z` component to equal the `w` component, when perspective division occurs, the `z` component will always be `1.0`.
  - Example: $\frac{w}{w} = 1.0$

Optimize the skybox drawing by drawing it last and using depth testing to omit hidden fragments.
- Skybox Optimization

Environment Mapping

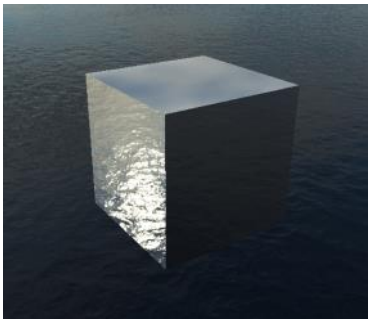


We now have the entire surrounding environment mapped to a single texture object.
- With this information, we could give objects reflective and refractive properties.
- Techniques that use an environment cubemap are called **environment mapping** techniques. The two most popular ones are **reflection** and **refraction**.

Reflection

Reflection is the property that an object (or part of an object) reflects its surrounding environment; the object's colors are more or less equal to its environment based on the angle of the viewer (e.g. a mirror).

The image to the right shows how we can calculate a reflection vector and use that vector to sample from a cubemap.
- We calculate a reflection direction vector $\vec{R}$ around the objects normal direction vector $\vec{N}$ based on the view direction vector $\vec{I}$.
  - We can calculate $\vec{R}$ using GLSL's built-in `reflect` function. The resulting $\vec{R}$ is then used as a direction vector to index/sample the cubemap, returning a color value of the environment. This has the effect of the object reflecting the skybox.



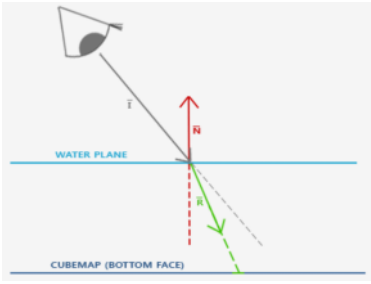
For this next part, use this vertex data for the position and normal attributes of the reflective cube: Reflective Cube Vertices

Create a cube object the reflects the skybox.
- Reflective Cube

In reality, most models aren't completely reflective.
- We could introduce reflection maps, which are texture images that we can sample to determine the reflectivity of a fragment. Using reflection maps can determine which parts of the model show reflection and by what intensity.

Refraction

Refraction is similar to reflection. It is the change in direction of light due to the change of the material the light flows through.
- Refraction is what we commonly see with water-like surfaces, where the light doesn't enter straight through, but bends a little.
  - Example: Think about looking at your arm when it's halfway under water. It sometimes appears that the underwater part of your arm is slightly shifted from the above water part.



Refraction is described by Snell's law. With environment maps, Snell's law looks like the image to the right.
- Similar to reflection, we calculate the refraction direction vector $\vec{R}$ using the view direction vector $\vec{I}$ and normal direction vector $\vec{N}$.
  - It is easy to implement this refraction using GLSL's `refract` function here. We pass $\vec{I}$ and $\vec{N}$ like we did with the `reflect` function, but now we also pass a ratio between both materials' refractive indices.

Material	Refractive index
Air	1.00
Water	1.33
Ice	1.309
Glass	1.52
Diamond	2.42

The refractive index determines the amount the light distorts/bends in a material where each material has its own refractive index.
- To the right are some common refractive indices.
- We use these refractive indices to calculate the ratio between both materials the light passes through.
  - In our case, the light/view ray goes from air to glass (if we assume the object is made of glass), so the ratio becomes $\frac{1.00}{1.52} = 0.658$.

Modify the fragment shader of the cube to implement refraction as if the cube was made of glass.
- Glass Refraction

With the right combination of lighting, reflection, refraction, and vertex movement, you can create very nice water graphics.
- **NOTE:** For physically accurate results, we should refract the light a second time when it leaves the object. For now, we're simply using single-sided refraction, which is fine for most purposes.

Dynamic Environment Maps

Right now, we've been using a static combination of images as the skybox, which looks great, but it doesn't include the actual 3D scene with possibly moving objects.
- Currently, if we add more objects to the scene, they are not reflected nor refracted by the cube.

Using framebuffers, it is possible to create a texture of the scene for all 6 different angles from the object in question and store those in a cubemap each frame. We can then use this (dynamically generated) cubemap to create realistic reflection and refraction surfaces that include all other objects.
- This is called **dynamic environment mapping**, because we dynamically create a cubemap of an object's surroundings and use that as its environment map.
- While it looks great, we have to render the scene 6 times per object using an environment map, which is an enormous performance penalty on your application.
  - Modern applications try to use the skybox and pre-render cubemaps wherever they can to still sort-of create dynamic environment maps.

While dynamic environment mapping is a great technique, it requires a lot of clever tricks and hacks to get it working in an actual rendering application without too many performance drops.

Cubemap Setup

Tuesday, May 3, 2022 8:33 PM

main()

```
// Generates and binds the cubemap texture object
unsigned int cubemap;
glGenTextures(1, &cubemap);
glBindTexture(GL_TEXTURE_CUBE_MAP, cubemap);

// Stores the directories for each face of the cubemap
std::vector<const char*> cubemapFacesDirs = {
   "Textures/container.jpg",    // Right face
   "Textures/container.jpg",    // Left face
   "Textures/container.jpg",    // Top face
   "Textures/container.jpg",    // Bottom face
   "Textures/container.jpg",    // Front face
   "Textures/container.jpg",    // Back face
};

stbi_set_flip_vertically_on_load(false);

// Loads the texture for each face of the cubemap
int width, height, numChannels;
unsigned char* data;
for (int i = 0; i < cubemapFacesDirs.size(); ++i) {
   data = stbi_load(cubemapFacesDirs[i], &width, &height, &numChannels, 0);
   glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
}

stbi_set_flip_vertically_on_load(true);

// Configures the properties of the cubemap.
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
// Use edge clamping to reduce artifacts where face edges meet
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);
// Since a cubemap is a 3D texture, you must specify the r (z) wrapping technique
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
```

GL_CLAMP_TO_EDGE prevents accidental texture coordinate misses caused by being exactly between two faces, which would could gaps or cause unwanted texture repeating between cubemap faces.

NOTE: The order of the faces in the cubemapFacesDirs vector is not the same as the cubemap texture targets table (the front and back faces are swapped in the order). This is because OpenGL uses RenderMan specifications for cubemaps, which assumes the image's origin is in the upper left (rather than the lower left with normal images). This is also why you have to set the `stbi_set_flip_vertically_on_load` flag to `false` before loading the cubemap face textures.

Skybox Texture Loading

Tuesday, May 3, 2022 9:04 PM

main()

```
std::vector<const char*> cubemapFacesDirs = {
   "Textures/skybox/right.jpg", // Right face
   "Textures/skybox/left.jpg",      // Left face
   "Textures/skybox/top.jpg",       // Top face
   "Textures/skybox/bottom.jpg",// Bottom face
   "Textures/skybox/front.jpg", // Front face
   "Textures/skybox/back.jpg",      // Back face
};

unsigned int cubemap = loadCubemap(cubemapFacesDirs);
```

loadCubemap function

```
// Generates a cubemap from a vector of texture directories
unsigned int loadCubemap(std::vector<const char*> cubemapFacesDirs) {
   // A cubemap can only have 6 faces, so we want to check to make sure we pass at least 6.
   int numDirs = cubemapFacesDirs.size();
   if (numDirs < 6) {
      std::cout << "Argument must be a vector of at least 6 elements. Returning 0..." << std::endl;
      return 0;
   }

   // Standard cubemap setup (from the previous subpage)
   unsigned int cubemap;
   glGenTextures(1, &cubemap);
   glBindTexture(GL_TEXTURE_CUBE_MAP, cubemap);

   stbi_set_flip_vertically_on_load(false);

   int width, height, numChannels;
   for (int i = 0; i < numDirs; ++i) {
      unsigned char* data = stbi_load(cubemapFacesDirs[i], &width, &height, &numChannels, 0);
      if (data) {
         glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
      }
      else {
         std::cout << "Cubemap texture failed to load at path: " << cubemapFacesDirs[i] << std::endl;
      }
      // Make sure you free the image data to release the memory
      stbi_image_free(data);
   }

   stbi_set_flip_vertically_on_load(true);

   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);

   return cubemap;
}
```

Sample Skybox Vertices

Wednesday, May 4, 2022 1:19 PM

```
float skyboxVertices[] = {
    // Positions
    -1.0f,  1.0f, -1.0f,
    -1.0f, -1.0f, -1.0f,
     1.0f, -1.0f, -1.0f,
     1.0f, -1.0f, -1.0f,
     1.0f,  1.0f, -1.0f,
    -1.0f,  1.0f, -1.0f,

    -1.0f, -1.0f,  1.0f,
    -1.0f, -1.0f, -1.0f,
    -1.0f,  1.0f, -1.0f,
    -1.0f,  1.0f, -1.0f,
    -1.0f,  1.0f,  1.0f,
    -1.0f, -1.0f,  1.0f,

     1.0f, -1.0f, -1.0f,
     1.0f, -1.0f,  1.0f,
     1.0f,  1.0f,  1.0f,
     1.0f,  1.0f,  1.0f,
     1.0f,  1.0f, -1.0f,
     1.0f, -1.0f, -1.0f,

    -1.0f, -1.0f,  1.0f,
    -1.0f,  1.0f,  1.0f,
     1.0f,  1.0f,  1.0f,
     1.0f,  1.0f,  1.0f,
     1.0f, -1.0f,  1.0f,
    -1.0f, -1.0f,  1.0f,

    -1.0f,  1.0f, -1.0f,
```

```
     1.0f,  1.0f, -1.0f,
     1.0f,  1.0f,  1.0f,
     1.0f,  1.0f,  1.0f,
    -1.0f,  1.0f,  1.0f,
    -1.0f,  1.0f, -1.0f,

    -1.0f, -1.0f, -1.0f,
    -1.0f, -1.0f,  1.0f,
     1.0f, -1.0f, -1.0f,
     1.0f, -1.0f, -1.0f,
    -1.0f, -1.0f,  1.0f,
     1.0f, -1.0f,  1.0f
};
```

Skybox Drawing

Wednesday, May 4, 2022 1:23 PM

skybox.vert

```
#version 330 core

layout (location = 0) in vec3 aPos;

out vec3 texCoords;

uniform mat4 projection;
uniform mat4 view;

void main() {
   // Since the origin is at (0,0,0), the (interpolated) position is the same as the texture direction vector
   texCoords = aPos;
   // Don't need model matrix since vertex position data is in NDC and world space isn't a concern for skyboxes
   gl_Position = projection * view * vec4(aPos, 1.0);
}
```

skybox.frag

```
#version 330 core

out vec4 FragColor;

uniform samplerCube skybox;

in vec3 texCoords;

void main() {
   FragColor = texture(skybox, texCoords);
}
```

If continuing from last chapter, insert the following code after binding your custom fbo.
Make sure the skybox is the first thing drawn in the scene and that depth testing is disable when drawing it so that it alway s appears behind all other objects in the scene.

Render Loop

```
// Creates the view matrix
glm::mat4 view = camera.GetViewMatrix();
// Creates the projection matrix using perspective projection
glm::mat4 projection = glm::perspective(glm::radians(camera.fov), (float)WIDTH / (float)HEIGHT, 0.1f, 100.0f);

// Disable depth testing so the skybox is always drawn behind other objects
glDepthMask(GL_FALSE);
skyboxShader.use();
glBindTexture(GL_TEXTURE_CUBE_MAP, cubemap);

// Prevents the skybox from being translated when the player moves by resetting the translation portion (last column) of the  matrix
glm::mat4 skyboxView = glm::mat4(glm::mat3(camera.GetViewMatrix()));

glUniform1i(glGetUniformLocation(skyboxShader.id, "skybox"), 0); // This line is optional
glUniformMatrix4fv(glGetUniformLocation(skyboxShader.id, "view"), 1, GL_FALSE, glm::value_ptr(skyboxView));
glUniformMatrix4fv(glGetUniformLocation(skyboxShader.id, "projection"), 1, GL_FALSE, glm::value_ptr(projection));

glBindVertexArray(skyboxVAO);
glDrawArrays(GL_TRIANGLES, 0, 36);

glDepthMask(GL_TRUE);
```

Skybox Optimization

Wednesday, May 4, 2022 2:54 PM

skybox.vert

```
void main() {
   texCoords = aPos;
   vec4 pos = projection * view * vec4(aPos, 1.0);
   gl_Position = pos.xyww; // Forces z-component to always equal 1.0, only allowing the skybox to be rendered when nothing is in front
}
```

Render Loop

```
// Draw the scene
[...]

glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LEQUAL);
// Draw the skybox
[...]
```

Since the skybox is drawn last and we want to render it behind everything, we want the depth test to pass if the skybox fragment's depth value is less than or equal to (`GL_LEQUAL`) the depth buffer (`GL_LESS` is the default depth test function). This is because the depth buffer will be filled with values of `1.0` for the skybox. Using `GL_LEQUAL` thus means that all objects in the scene will render in front of the skybox, omitting those fragment shader calls for the skybox.

NOTE: If you're following along from the framebuffers chapter, you'll actually need to draw the skybox before the transparent windows so the transparent values of the window show the skybox and not the background color.

Reflective Cube Vertices

Wednesday, May 4, 2022 3:57 PM

```
float reflectiveCubeVertices[] = {
   // Positions          // Normals
   -0.5f, -0.5f, -0.5f,   0.0f,  0.0f, -1.0f,
    0.5f, -0.5f, -0.5f,   0.0f,  0.0f, -1.0f,
    0.5f,  0.5f, -0.5f,   0.0f,  0.0f, -1.0f,
    0.5f,  0.5f, -0.5f,   0.0f,  0.0f, -1.0f,
   -0.5f,  0.5f, -0.5f,   0.0f,  0.0f, -1.0f,
   -0.5f, -0.5f, -0.5f,   0.0f,  0.0f, -1.0f,

   -0.5f, -0.5f,  0.5f,   0.0f,  0.0f, 1.0f,
    0.5f, -0.5f,  0.5f,   0.0f,  0.0f, 1.0f,
    0.5f,  0.5f,  0.5f,   0.0f,  0.0f, 1.0f,
    0.5f,  0.5f,  0.5f,   0.0f,  0.0f, 1.0f,
   -0.5f,  0.5f,  0.5f,   0.0f,  0.0f, 1.0f,
   -0.5f, -0.5f,  0.5f,   0.0f,  0.0f, 1.0f,

   -0.5f,  0.5f,  0.5f,  -1.0f,  0.0f,  0.0f,
   -0.5f,  0.5f, -0.5f,  -1.0f,  0.0f,  0.0f,
   -0.5f, -0.5f, -0.5f,  -1.0f,  0.0f,  0.0f,
   -0.5f, -0.5f, -0.5f,  -1.0f,  0.0f,  0.0f,
   -0.5f, -0.5f,  0.5f,  -1.0f,  0.0f,  0.0f,
   -0.5f,  0.5f,  0.5f,  -1.0f,  0.0f,  0.0f,

    0.5f,  0.5f,  0.5f,   1.0f,  0.0f,  0.0f,
    0.5f,  0.5f, -0.5f,   1.0f,  0.0f,  0.0f,
    0.5f, -0.5f, -0.5f,   1.0f,  0.0f,  0.0f,
    0.5f, -0.5f, -0.5f,   1.0f,  0.0f,  0.0f,
    0.5f, -0.5f,  0.5f,   1.0f,  0.0f,  0.0f,
    0.5f,  0.5f,  0.5f,   1.0f,  0.0f,  0.0f,

   -0.5f, -0.5f, -0.5f,   0.0f, -1.0f,  0.0f,
```

```
    0.5f, -0.5f, -0.5f,   0.0f, -1.0f,  0.0f,
    0.5f, -0.5f,  0.5f,   0.0f, -1.0f,  0.0f,
    0.5f, -0.5f,  0.5f,   0.0f, -1.0f,  0.0f,
   -0.5f, -0.5f,  0.5f,   0.0f, -1.0f,  0.0f,
   -0.5f, -0.5f, -0.5f,   0.0f, -1.0f,  0.0f,

   -0.5f,  0.5f, -0.5f,   0.0f,  1.0f,  0.0f,
    0.5f,  0.5f, -0.5f,   0.0f,  1.0f,  0.0f,
    0.5f,  0.5f,  0.5f,   0.0f,  1.0f,  0.0f,
    0.5f,  0.5f,  0.5f,   0.0f,  1.0f,  0.0f,
   -0.5f,  0.5f,  0.5f,   0.0f,  1.0f,  0.0f,
   -0.5f,  0.5f, -0.5f,   0.0f,  1.0f,  0.0f
};
```

Reflective Cube

Wednesday, May 4, 2022 3:57 PM

reflect.vert

```
#version 330 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;

out vec3 position;
out vec3 normal;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
   position = vec3(model * vec4(aPos, 1.0)); // fragment position
   // We use the normal matrix to transform the normal vector to prevent incorrect normals if the object is scaled
   normal = mat3(transpose(inverse(model))) * aNormal; gl_Position = projection * view * vec4(position, 1.0);
}
```

reflect.frag

```
#version 330 core

out vec4 FragColor;

in vec3 normal;
in vec3 position;

uniform samplerCube skybox;
uniform vec3 cameraPos;

void main() {

   vec3 I = normalize(position - cameraPos); // View direction vector (from fragment to camera)
   vec3 R = reflect(I, normalize(normal)); // Reflection vector for sampling the skybox
   FragColor = texture(skybox, R);
}
```

main()

```
Shader reflectShader{ "Shaders/reflect.vert", "Shaders/reflect.frag" };

...

unsigned int reflectCubeVAO, reflectCubeVBO;
glGenVertexArrays(1, &reflectCubeVAO);
glGenBuffers(1, &reflectCubeVBO);

glBindVertexArray(reflectCubeVAO);
glBindBuffer(GL_ARRAY_BUFFER, reflectCubeVBO);

glBufferData(GL_ARRAY_BUFFER, sizeof(reflectiveCubeVertices), reflectiveCubeVertices, GL_STATIC_DRAW);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(sizeof(float) * 3));

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);

glBindVertexArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Render Loop

```
reflectShader.use();
```

```
glBindTexture(GL_TEXTURE_CUBE_MAP, cubemap);

glUniform3fv(glGetUniformLocation(reflectShader.id, "cameraPos"), 1, glm::value_ptr(camera.position));
glUniform1i(glGetUniformLocation(reflectShader.id, "skybox"), 0);
glUniformMatrix4fv(glGetUniformLocation(reflectShader.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
glUniformMatrix4fv(glGetUniformLocation(reflectShader.id, "view"), 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(glGetUniformLocation(reflectShader.id, "projection"), 1, GL_FALSE, glm::value_ptr(projection));

glBindVertexArray(reflectCubeVAO);
glDrawArrays(GL_TRIANGLES, 0, 36);

...

// Draw skybox
[...]
```

Glass Refraction

Wednesday, May 4, 2022 4:42 PM

reflect.frag

```
void main() {
   float ratio = 1.00 / 1.52; // Refraction ratio between air and glass
   vec3 I = normalize(position - cameraPos);
   vec3 R = refract(I, normalize(normal), ratio);
   FragColor = texture(skybox, R);
}
```