Computer Graphics II – Cubemaps

Kai Lawonn

Introduction

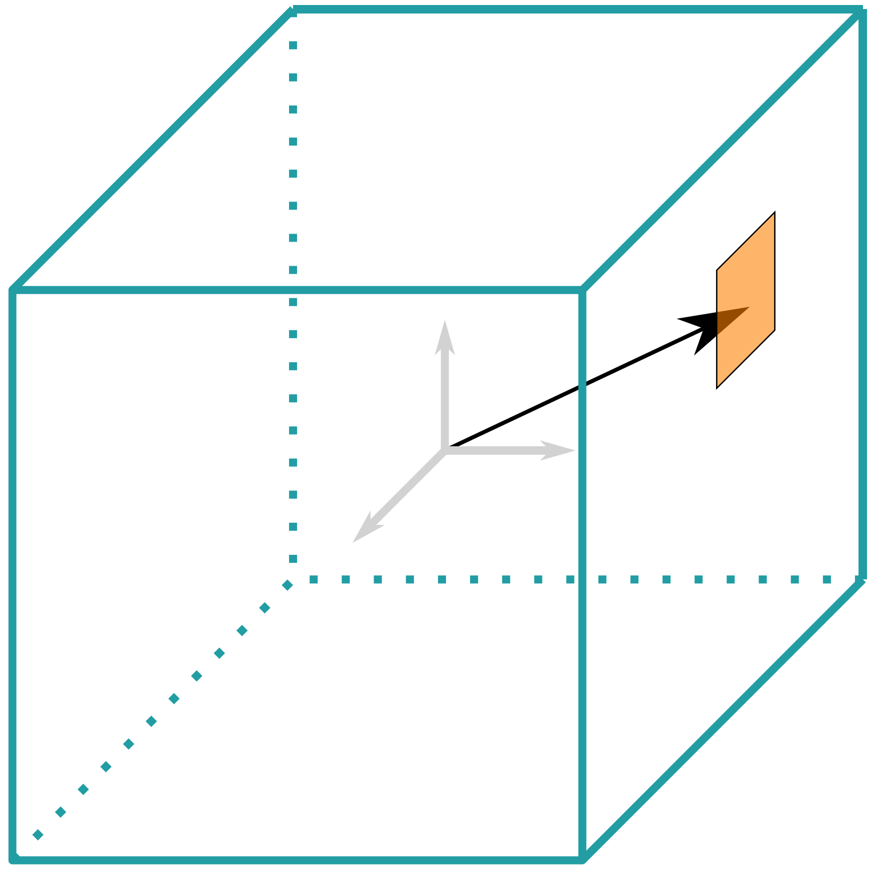
- We used 2D textures for a while now, but there are even more texture types
- Now, we will discuss a texture type that is actually a combination of multiple textures mapped into a single texture: a cube map

Introduction

- A cubemap is a texture containing 6 individual 2D textures forming a textured cube
- Why bother with combining 6 individual textures into a single entity instead of just using 6 individual textures?
- Advantage: they can be indexed/sampled using a direction vector

Introduction

- With a 1x1x1 unit cube with the origin of a direction vector residing at its center, we could sample a texture value from the cube map with a direction vector:



Magnitude?

The magnitude doesn’t matter

With the direction, OpenGL retrieves the corresponding hit texels and returns the properly sampled texture value

Introduction

- If we have a cube shape with a cubemap attached, the direction vector to sample the cubemap would be similar to the (interpolated) vertex position of the cube
- Then, we can sample the cubemap using the cube’s actual position vectors as long as the cube is centered on the origin
- Then, retrieve the texture coordinates of all vertices as the vertex positions of the cube
- The result is a texture coordinate that accesses the proper individual face texture of the cubemap

Creating a Cubemap

Creating a Cubemap

- To create a cubemap, generate a texture and bind it
- This time binding to GL_TEXTURE_CUBE_MAP:

```
unsigned int textureID;
glGenTextures(1, &textureID);
glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);
```

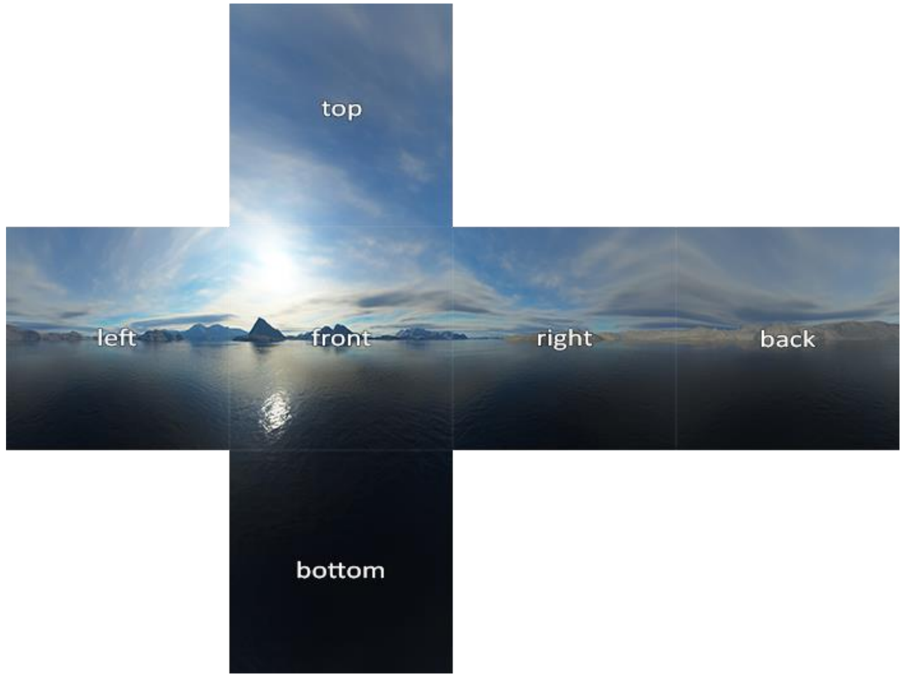
Creating a Cubemap

- Cubemap consists of 6 textures, one for each face
- Have to call glTexImage2D six times
- Additionally, set the texture target parameter to a specific face of the cubemap (tell OpenGL which texture belongs to which face)
- Thus, call glTexImage2D once for each face of the cubemap

Creating a Cubemap

- 6 faces corresponds to 6 special texture targets specifically for targeting a face of the cubemap:

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



Creating a Cubemap

- We can linearly increment the texture targets: loop over them by starting with GL_TEXTURE_CUBE_MAP_POSITIVE_X and incrementing it by 1 each iteration, looping through all the texture targets:

```
int width, height, nrChannels;
for (unsigned int i = 0; i < faces.size(); i++)
{
        unsigned char *data = stbi_load(faces[i].c_str(), &width, &height,
                &nrChannels, 0);
        glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB,
                width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
        stbi_image_free(data);
}
```

Creating a Cubemap

- Here, we have a vector called *faces* that contain the locations of all the textures required for the cubemap in the correct order
- This generates a texture for each face of the currently bound cubemap

Creating a Cubemap

- A cubemap is a texture: specify its wrapping and filtering methods:

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

- Call this only once for the entire cubemap, no need to do it for every face

Creating a Cubemap

- GL_TEXTURE_WRAP_R sets the wrapping method for the texture's R coordinate (texture's 3rd dimension, like the z for positions)
- Wrapping set to GL_CLAMP_TO_EDGE: texture coordinates exactly between two faces might not hit an exact face so by using GL_CLAMP_TO_EDGE OpenGL always return their edge values whenever we sample between faces

Creating a Cubemap

- Before drawing the objects that will use the cubemap, activate the texture unit and bind the cubemap before rendering
- In the fragment shader use a samplerCube (instead of sampler2D)
- Use a vec3 direction vector instead of a vec2 for sampling:

```
in vec3 TexDir;
uniform samplerCube skybox;
void main()
{
    FragColor = texture(skybox, TexDir);
}
```

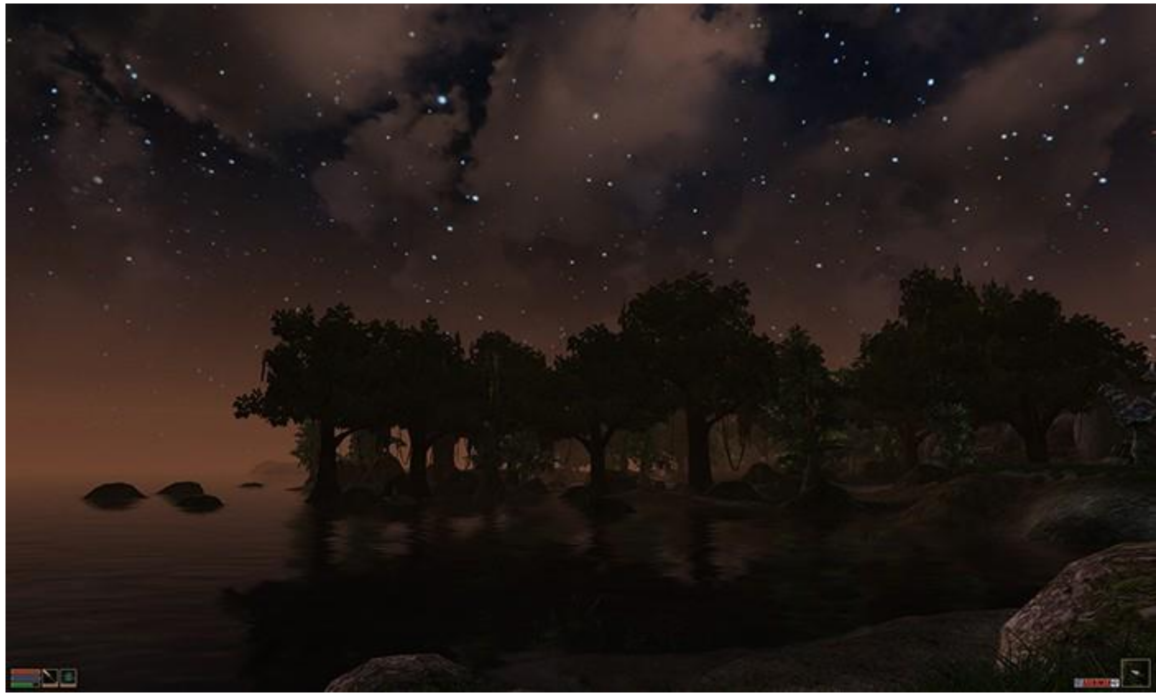
Creating a Cubemap

- That is great, but why bother?
- One interesting application of a cubemap is a skybox

Skybox

Skybox

- A skybox is a (large) cube that encompasses the entire scene containing 6 images of a surrounding environment
- Gives the illusion of an environment the player is actually in
- Examples of skyboxes used in videogames are images of mountains, of clouds or of a starry night sky:

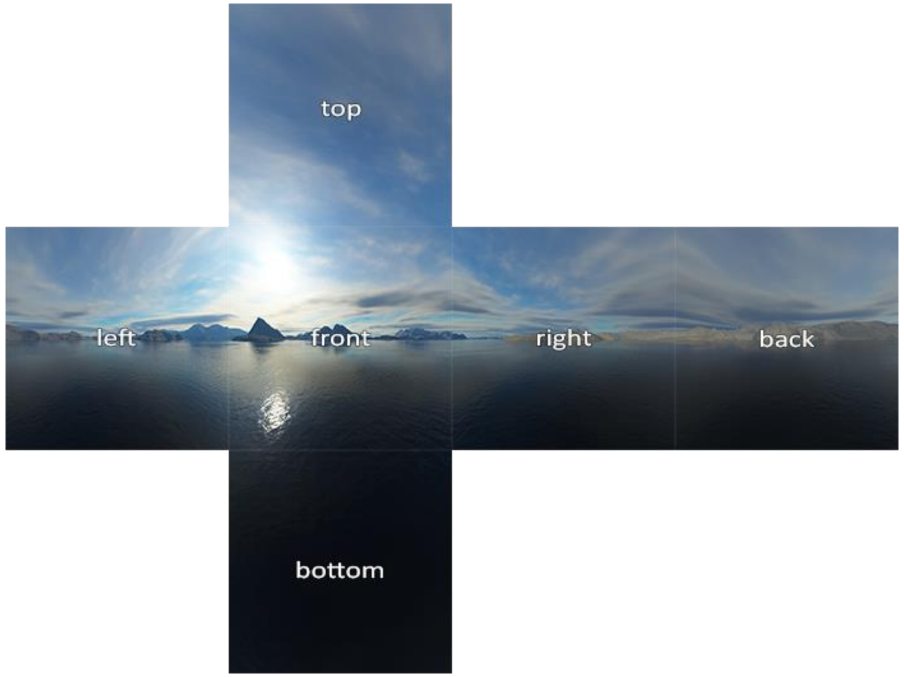


Skybox

- Skyboxes like this suit cubemaps perfectly: a cube that has 6 faces and needs to be textured per face
- In the previous image several images of a night sky give the illusion the player is in some large universe while s/he’s actually inside box

Skybox

- Folding those 6 sides gets us the completely textured cube that simulates a large landscape
- Some resources provide the skyboxes in a format like this (manually extract the 6 face images), but mostly provided as 6 single texture images



Loading a Skybox

- Skybox is just a cubemap, loading a skybox is not too different
- To load the skybox, we use the following function that accepts a vector of 6 texture locations:

```
unsigned int loadCubemap(vector<std::string> faces)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

    int width, height, nrChannels;
    …
```

Loading a Skybox

```
…
    for (unsigned int i = 0; i < faces.size(); i++)
    {
        unsigned char *data = stbi_load(faces[i].c_str(), &width, &height,
              &nrChannels, 0);
        if (data)
        {
            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB,
              width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
            stbi_image_free(data);
        }
        else
        {
            std::cout << "Cubemap texture failed to load at path: " <<
              faces[i] << std::endl;
            stbi_image_free(data);
        }
    }
```

Loading a Skybox

```
…
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_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 textureID;
}
```

Loading a Skybox

- The function is basically all the cubemap code from before
- Before calling this function, load the appropriate texture paths in a vector in the order as specified by the cubemap enums:

```
vector<std::string> faces
   {
       FileSystem::getPath("right.jpg"),
       FileSystem::getPath("left.jpg"),
       FileSystem::getPath("top.jpg"),
       FileSystem::getPath("bottom.jpg"),
       FileSystem::getPath("front.jpg"),
       FileSystem::getPath("back.jpg")
   };
```

Loading a Skybox

- Then load the skybox as a cubemap with cubemapTexture as its id
- In the next step, we can bind it to a cube to finally replace the default clear color as the background all this time

```
unsigned int cubemapTexture = loadCubemap(faces);
```

Displaying a Skybox

- A cubemap used to texture a 3D cube can be sampled using the positions of the cube as the texture coordinates
- When a cube is centered on the origin (0,0,0) each of its position vectors is also a direction vector from the origin
- This direction vector is exactly what we need to get the corresponding texture value at that specific cube's position
- Thus, we only need to supply position vectors and don't need texture coordinates

Displaying a Skybox

- For the skybox new shaders are needed
- We have only one vertex attribute → the vertex shader is quite simple:

```
#version 330 core
layout (location = 0) in vec3 aPos;

out vec3 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    TexCoords = aPos;
    gl_Position = projection * view * vec4(aPos, 1.0);
}
```

Displaying a Skybox

- We set the incoming local position vector as the outcoming texture coordinate for (interpolated) use in the fragment shader

```
#version 330 core
layout (location = 0) in vec3 aPos;

out vec3 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    TexCoords = aPos;
    gl_Position = projection * view * vec4(aPos, 1.0);
}
```

Displaying a Skybox

- The fragment shader then takes these as input to sample a samplerCube:

```
#version 330 core
out vec4 FragColor;

in vec3 TexCoords;

uniform samplerCube skybox;

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

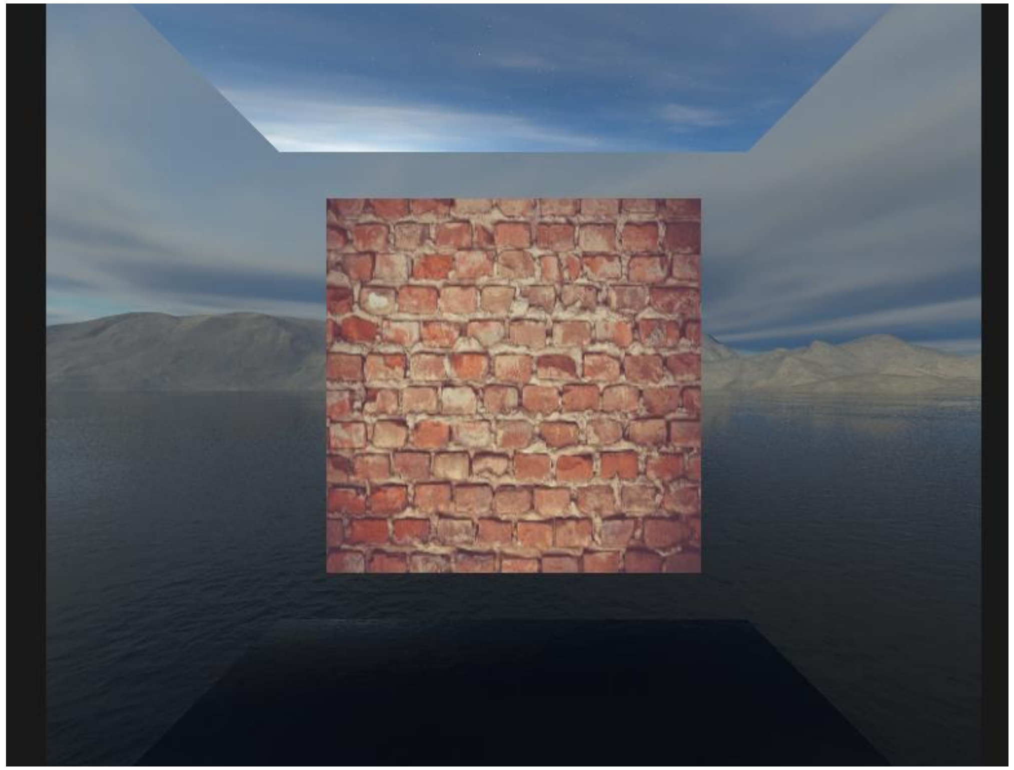
Displaying a Skybox

- Rendering the skybox is easy with the cubemap texture: bind the cubemap texture and the skybox sampler is automatically filled
- To draw the skybox: draw it first object, disable depth writing → skybox always be drawn at the background of all the other objects

```
glDepthMask(GL_FALSE);
skyboxShader.use();
skyboxShader.setMat4("view", view);
skyboxShader.setMat4("projection", projection);
// skybox cube
glBindVertexArray(skyboxVAO);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
glDrawArrays(GL_TRIANGLES, 0, 36);
glBindVertexArray(0);
glDepthMask(GL_TRUE);
```

F5…

- … something is wrong
- Want the skybox to be centered around the player with the impression the surrounding environment is extremely large
- This is currently not the case
- We want to remove the translation part of the view matrix to not affect the skybox’s position vectors



Displaying a Skybox

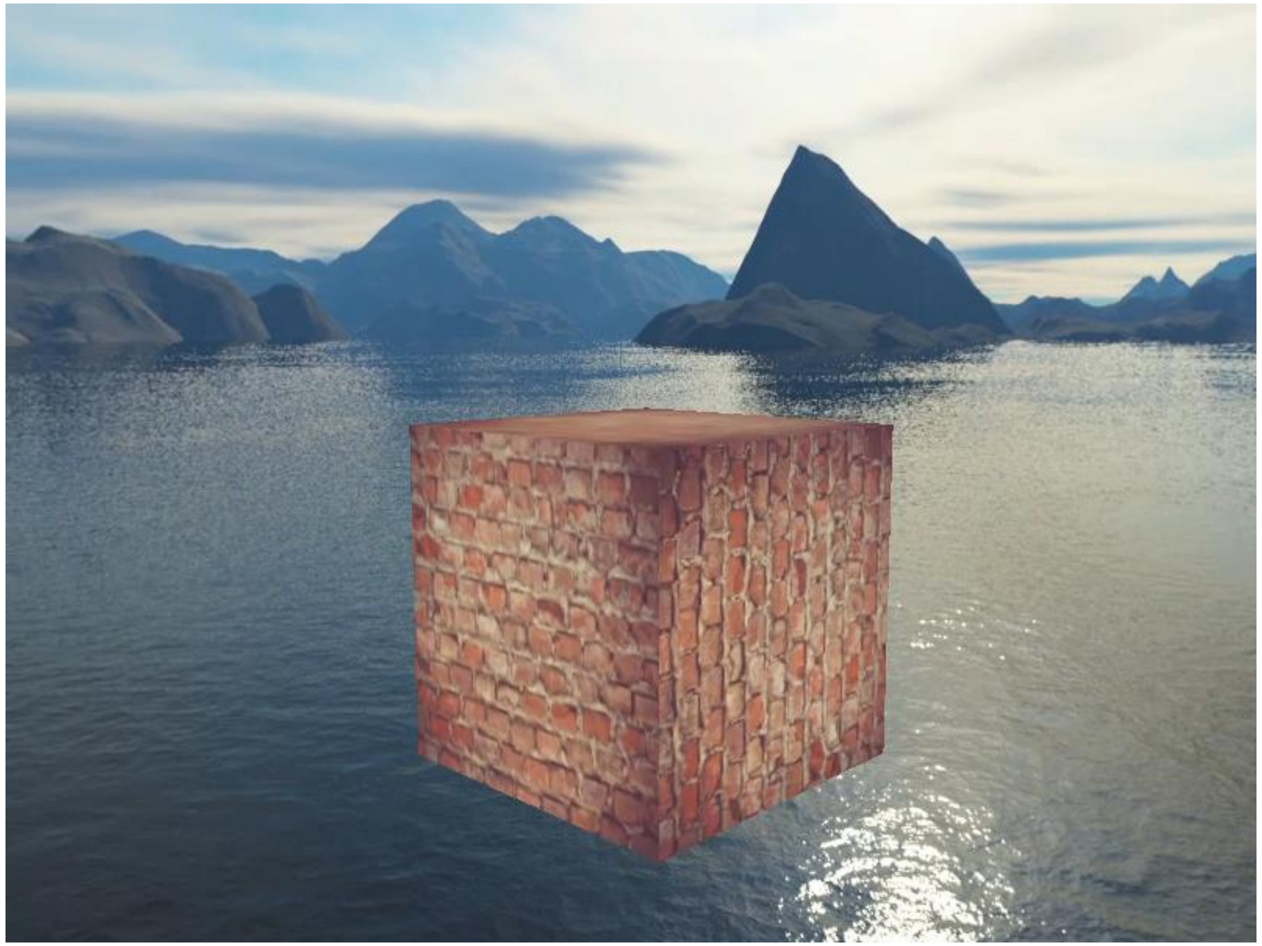
- In the lighting lecture we removed the translation part of transformation matrices (keep upper-left 3x3 matrix, which removes the translation components)
- Achieve this by converting the view matrix to a 3x3 matrix (removing translation) and converting it back to a 4x4 matrix:

```
view = glm::mat4(glm::mat3(camera.GetViewMatrix()));
```

- This removes any translation, but keeps all rotation transformations (user can still look around)

F5…

- … finally



Optimization

An Optimization

- We rendered the skybox first before all the other objects in the scene
- Not too efficient: running the fragment shader for each pixel even only a small part of the skybox will eventually be visible $\rightarrow$ discard fragments using early depth testing saving us valuable bandwidth

An Optimization

- Thus, render the skybox last → depth buffer is filled with objects' depth values, and the skybox's fragments rendered when the early depth test passes (reducing the calls to the fragment shader)
- Problem: skybox will most likely fail to render (it's only a 1x1x1 cube), failing most depth tests
- Rendering without depth testing is not a solution (skybox will overwrite all the other objects in the scene)
- Trick the depth buffer such that the skybox has the maximum depth value of 1.0 → fails the depth test wherever there's a different object in front of it

Displaying a Skybox

- Recall: perspective division is performed after the vertex shader, dividing the gl_Position's xyz coordinates by its w
- Also: z component of the division is equal to that vertex's depth value
- → Set $z = w$, result in a z component that is always 1.0, (perspective division is applied: $\frac{z}{w} = \frac{w}{w} = 1$)

```
void main()
{
    TexCoords = aPos;
    vec4 pos = projection * view * vec4(aPos, 1.0);
    gl_Position = pos.xyww;
}
```

An Optimization

- NDCs are always $z = 1.0$: the maximum depth value
- Skybox will only be rendered wherever there are no objects visible (then it pass the depth test, everything else is in front of the skybox)
- Change depth function by setting it to GL_LEQUAL instead of the default GL_LESS
- Depth buffer will be filled with 1.0 for the skybox → make sure the skybox passes the depth tests with values $\leq$ depth buffer

Displaying a Skybox

- Code:

```
// draw skybox as last
glDepthFunc(GL_LEQUAL);
skyboxShader.use();
view = glm::mat4(glm::mat3(camera.GetViewMatrix()));
skyboxShader.setMat4("view", view);
skyboxShader.setMat4("projection", projection);
// skybox cube
glBindVertexArray(skyboxVAO);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
glDrawArrays(GL_TRIANGLES, 0, 36);
glBindVertexArray(0);
glDepthFunc(GL_LESS); // set depth function back to default
```

Environment Mapping

Introduction

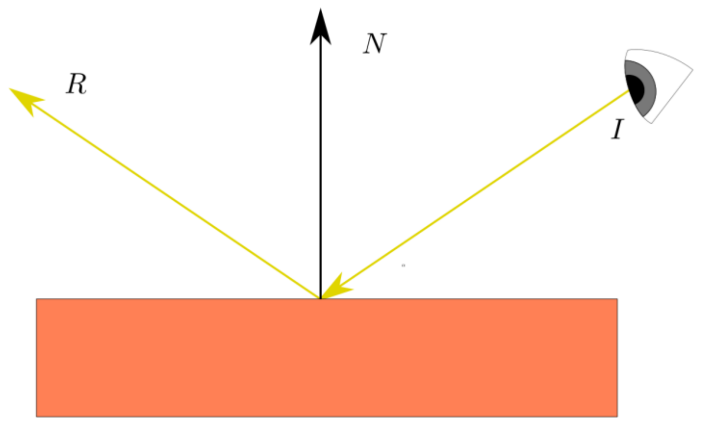
- With the surrounding environment mapped in a single texture object, we could use that information for more than just a skybox
- Using a cubemap with an environment, we could give objects reflective or refractive properties
- Techniques that use an environment cubemap like this are called environment mapping techniques and the two most popular ones are *reflection* and *refraction*

Reflection

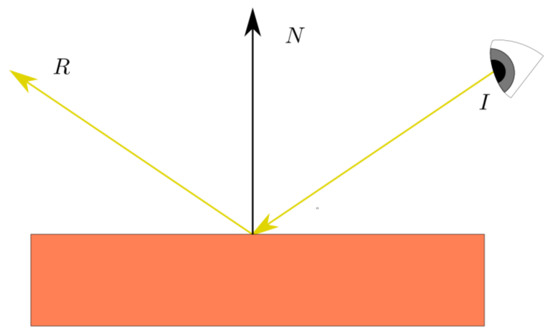
- Reflection is the property that an object (or part of an object) reflects its surrounding environment e.g. the object's colors are more or less equal to its environment based on the angle of the viewer
- A mirror for example is a reflective object: it reflects its surroundings based on the viewer's angle

Reflection

- The basics of reflection are not that difficult
- The following image shows how we can calculate a reflection vector and use that vector to sample from a cubemap:



Reflection



- We calculate a reflection vector R around the object's normal vector N based on the view direction vector I
- Calculate this reflection vector using GLSL's built-in reflect function:
- Resulting vector R is then used as a direction vector for the cubemap
- The resulting effect is that the object seems to reflect the skybox

Reflection

- Creating reflections isn’t too difficult (skybox scene)
- Change the fragment shader to give the container reflections

```
#version 330 core
out vec4 FragColor;

in vec3 Normal;
in vec3 Position;

uniform vec3 cameraPos;
uniform samplerCube skybox;

void main(){
    vec3 I = normalize(Position - cameraPos);
    vec3 R = reflect(I, normalize(Normal));
    FragColor = vec4(texture(skybox, R).rgb, 1.0);
}
```

Reflection

- Using normal vectors → transform them with a normal matrix
- *Position* output vector is a world-space position vector (used to calculate the view direction vector in the fragment shader)

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;

out vec3 Normal;
out vec3 Position;
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main(){
    Normal = mat3(transpose(inverse(model))) * aNormal;
    Position = vec3(model * vec4(aPos, 1.0));
    gl_Position = projection * view * model * vec4(aPos, 1.0);}
```

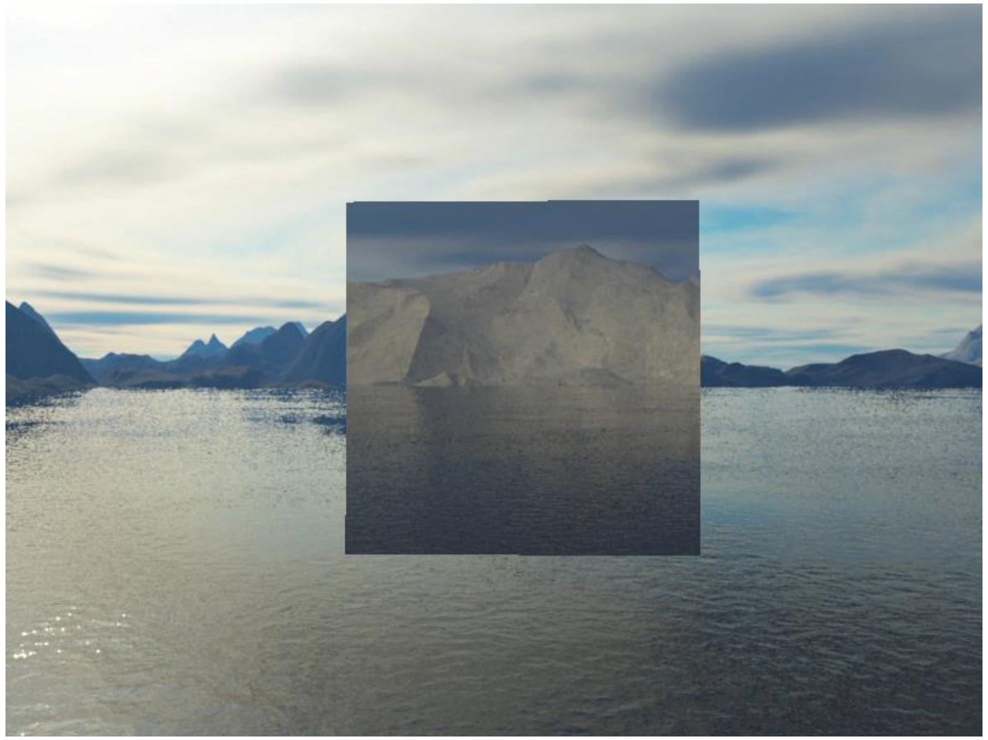
Reflection

- Then we also want to bind the cubemap texture before rendering the container:

```
glBindVertexArray(cubeVAO);
glBindTexture(GL_TEXTURE_CUBE_MAP, cubemapTexture);
glDrawArrays(GL_TRIANGLES, 0, 36);
```

F5…

- … a box that acts like a perfect mirror
- The surrounding skybox is perfectly reflected on the container:

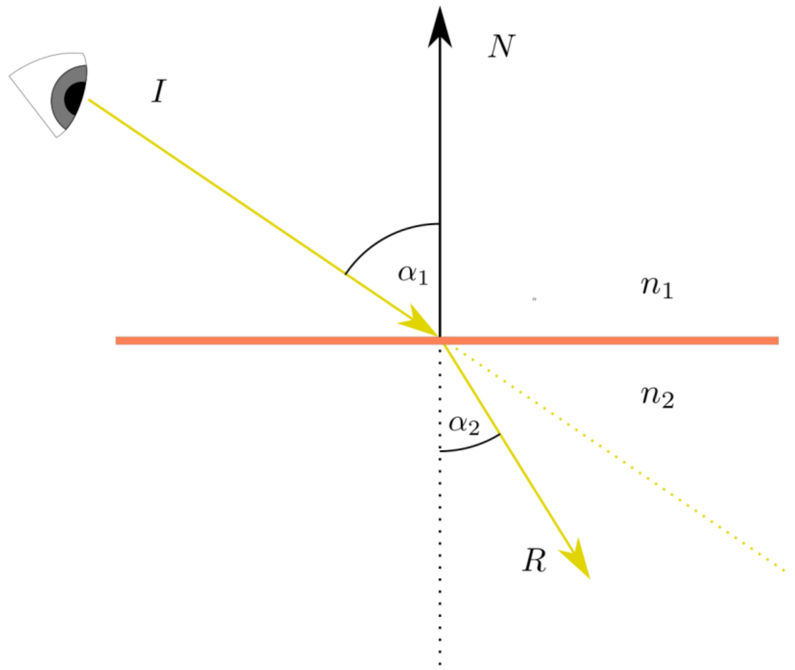


Refraction

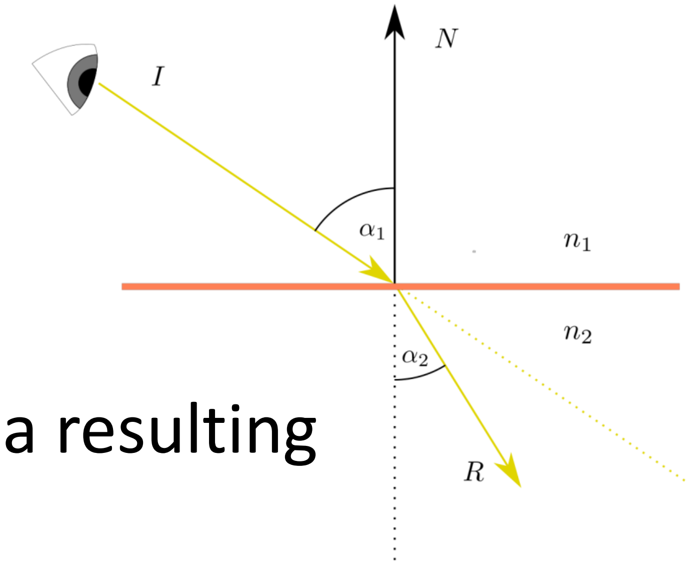
- Another form of environment mapping is called refraction
- Refraction is the change in direction of light due to the change of the material the light flows through
- Refraction is what we commonly see with water-like surfaces where the light doesn’t enter straight through, but bends a little (looking at a spoon in a glass full of tea)

Refraction

- Refraction is described by Snell's law: $$\frac{\sin \alpha_2}{\sin \alpha_1} = \frac{n_1}{n_2}$$
- n_1, n_2 are the refractive indices (unitless)
- Looks like this:



Refraction



- Again, given is a view vector I, a normal vector N and a resulting refraction vector R
- Direction of the view vector is slightly bend
- This bended vector R is then used to sample from the cubemap
- Refraction can be computed by using GLSL's built-in refract function (input a normal vector, a view direction and a ratio between both materials' refractive indices)

Refraction

- The refractive index determines the amount light distorts/bends of a material
- Most common refractive indices are given in the following table:

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

- Refractive indices used to calculate the ratio between both materials the light passes through (our case, the light/view ray goes from air to a glass container
- $Ratio = \frac{1}{1.52} \approx 0.658$

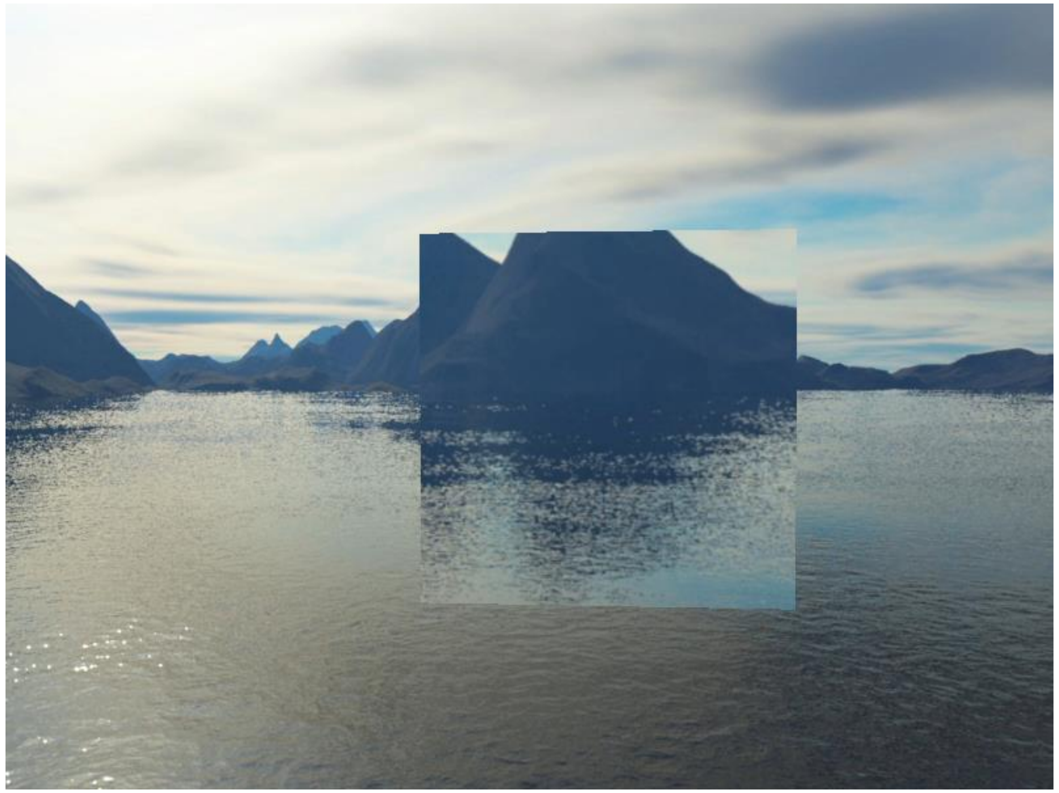
Refraction

- Everything is already set up, we only have to change the fragment shader

```
void main()
{
        float ratio = 1.00 / 1.52;
        vec3 I = normalize(Position - cameraPos);
        vec3 R = refract(I, normalize(Normal), ratio);
        FragColor = vec4(texture(skybox, R).rgb, 1.0);
}
```

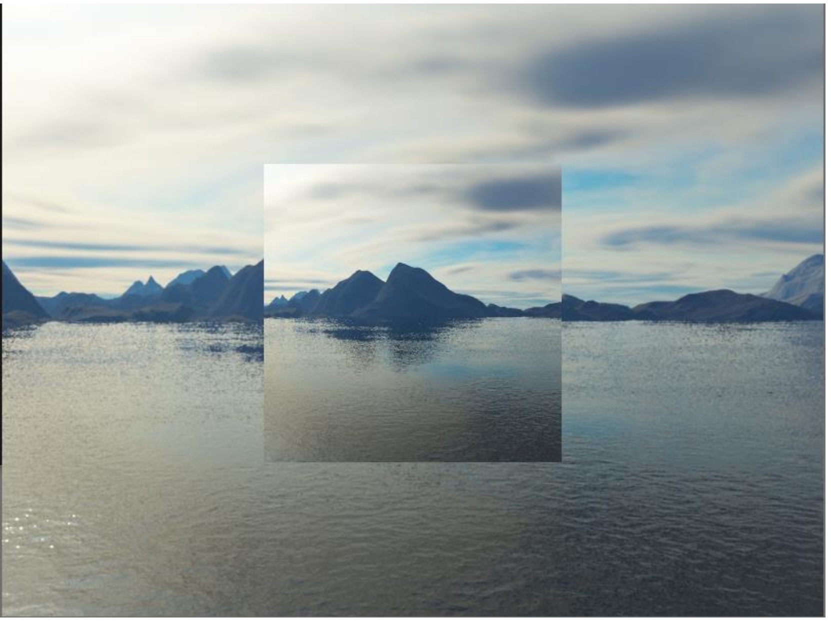
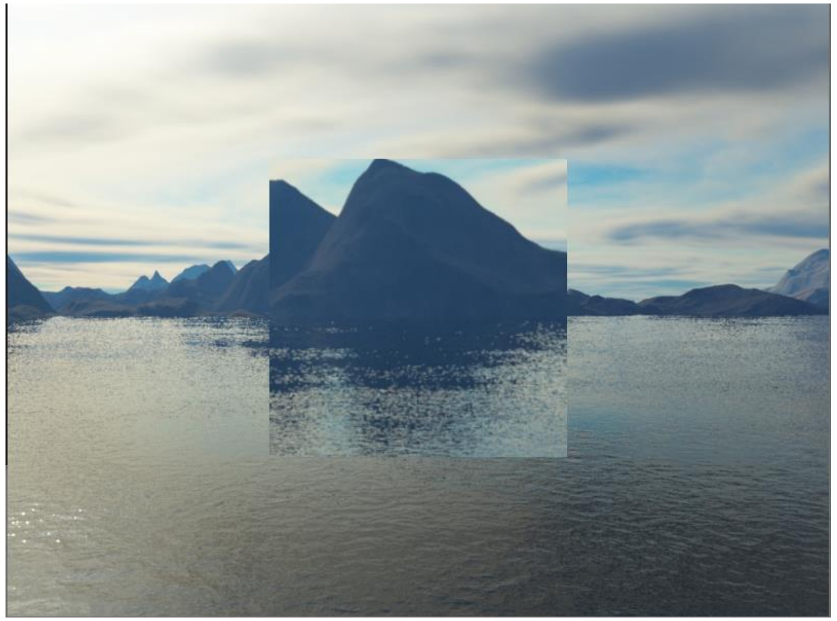
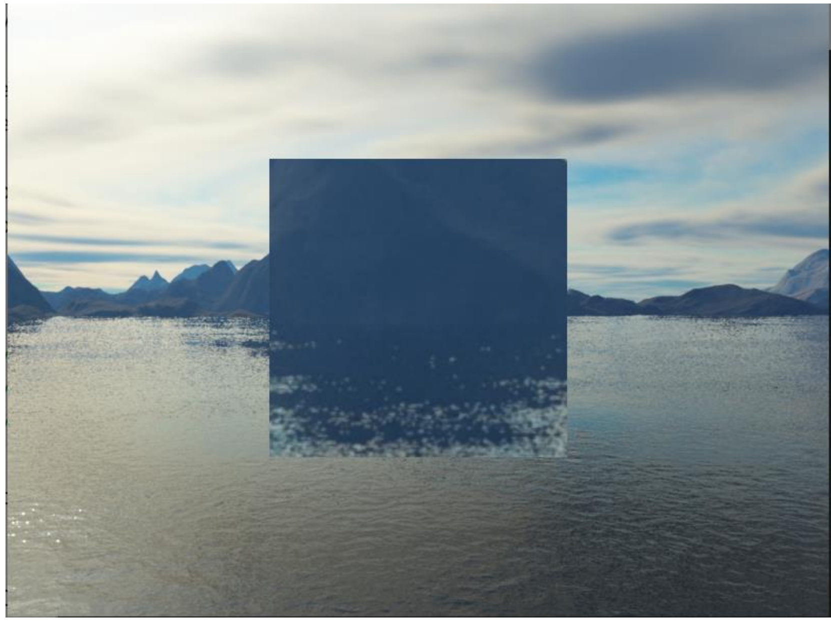
F5…

- … refraction acts slightly as a magnifying glass right now



Ratios

- (fltr): $1/0.52, 1/1.52, 1/2.52$

Remark

- With the right combination of lighting, reflection, refraction and vertex movement you can create pretty neat water graphics
- Do note that for physically accurate results we also should refract the light again when it leaves the object; now we simply used single-side refraction which is fine for most purposes

Dynamic Environment Maps

Dynamic Environment Maps

- Right now, using a static combination of images as the skybox, doesn’t include the actual scene with possibly moving objects
- If we had a mirror-like objects with multiple surrounding objects, only the skybox would be visible in the mirror as if it was the only object in the scene

Dynamic Environment Maps

- Using framebuffers it is possible to create a texture of the scene for all 6 different angles from the object → use for the cubemap
- This (dynamically generated) cubemap creates realistic reflection and refractive surfaces including 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

Dynamic Environment Maps

- While it looks great, it has one enormous disadvantage: render the scene 6x per object using an environment map (performance penalty)
- Modern applications try to use the skybox as much as possible and where possible pre-compile cubemaps
- 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

Questions???