POINT SHADOWS

Perspective projections make the most sense for light sources that have actual locations, such as point lights, while orthographic projections are used for directional lights and spotlights.

- Directional shadow maps are used for directional lights and spotlights.
  - The depth map is generated from only the direction the light is looking at.
- Omnidirectional shadow maps are used for point lights.
  - The depth map is generated from all directions around the light source.

The technique for omnidirectional shadow mapping is similar to directional shadow mapping.

- We generate a depth map from the light's perspective(s), sample the depth map based on the current fragment position, and compare each fragment with the stored depth value to see whether it is in shadow.
- The main difference is that the depth map we use.
  - The depth map we need requires rendering a scene from all surrounding directions of a point light.
    - As such a normal 2D depth map won't work.
    - However, a cubemap can store full environment data with only 6 faces, so it is possible to render the entire scene to each of the faces of a cubemap and sample these as the point light's surrounding depth values.
    - The generated depth cubemap is then passed to the lighting fragment shader that samples the cubemap with a direction vector to obtain the closest depth (from the light's perspective) at that fragment.

Generating the Depth Cubemap

To create a cubemap of a light's surrounding depth values, we have to render the scene 6 times: once for each face.

- One obvious way to do this is to render the scene 6 times with 6 different view matrices, each attaching a different cubemap face to the framebuffer object.
  - This idea would look something like this.
  - This is super expensive, though, so we'll be discussing a different technique that uses the geometry shader to build the depth cubemap with a single render pass.

Generate and configure a depth cubemap for storing depth data and attach it to the `depthMapFBO` framebuffer.

- Depth Cubemap

With omnidirectional shadow maps, we have two render passes (exactly like with directional shadow maps):

1. First, we generate the depth cubemap.
2. Then, we use the depth cubemap in the normal render pass to add shadows to the scene.

The above process looks somewhat like this.

- **NOTE:** The process is exactly the smae as with directional shadow mapping, except, this time, we render to and use a cubemap depth texture instead of a 2D depth texture.

Light Space Transform

With the framebuffer and cubemap set, we need to transform all the scene's geometry to the relevant light spaces in all 6 directions of light. Just like in the last chapter, we're going to need a light space transformation matrix ***T***, but this time, one for each face.

- Each light space transformation matrix contains both a projection and view matrix.
  - For the projection matrix, we're going to use a perspective projection matrix, since the light source represents a point in space.
  - Each light space transformation matrix uses the same projection matrix.
    - The perspective projection matrix must have a FOV of 90 degrees. This ensures the viewing field is exactly large enough to fill a single face of the cubemap such that all faces align correctly to each other at the edges.
  - However, we do need a different view matrix per direction, with each looking at one face direction of the cubemap.
    - The order of these is important, since we'll be iterating using `GL_TEXTURE_CUBE_MAP_POSITIVE_X + i`.
    - The order is right, left, top, bottom, near, and far.
- The light space transformation matrix is then sent to the shaders that render the depth into the cubemap.

Create the light space matrices (from the perspective matrix and view matrices) for each face of the depth cubemap.

- Depth Cubemap Projection/View Matrices

Depth Shaders

To render depth values to a depth cubemap, we're going to need a geometry shader, which will be responsible for transforming all world-space vertices to the 6 different light spaces.

- Therefore, the vertex shader simply transforms vertices to world-space and directs them to the geometry shader.

Write the vertex shader for the omnidirectional depth map shader.

- Omnidirectional Depth Map Vertex Shader

The geometry shader will transform the vertices of each triangle of the cubemap to the light spaces.

- The geometry shader has a built-in variable, `gl_Layer`, that specifies which cubemap face to emit a primitive to.
  - When left alone, the geometry shader just sends its primitives further down the pipelines (as usual), but when we update this variable, we can control to which cubemap face we render to for each primitive.
    - **NOTE:** This only works when we have a cubemap texture attached to the active framebuffer.

Write the geometry shader for the omnidirectional depth map shader.

- Omnidirectional Depth Map Geometry Shader

In the last chapter, we used an empty fragment shader and let OpenGL figure out the depth values of the depth map. This time, however, we're going to calculate our own (linear) depth as the linear distance between each closest fragment position and the light source's position.

- Calculating our own depth values makes the later shadow calculations a bit more intuitive.

Write the fragment shader for the omnidirectional depth map shader.

- Omnidirectional Depth Map Fragment Shader

Rendering the scene with the above shaders and the cubemap-attached framebuffer object active should give you a completely filled depth cubemap for the second pass's shadow calculations.

Omnidirectional Shadow Maps

With everything set up, it is time to render the actual omnidirectional shadows. The procedure is similar to the previous chapter, although this time we bind a cubemap texture instead of a 2D texture, and also pass the light projection's far plane variable to the shaders, as seen here.

For the scene, we're going to render a few cubes in a large cube scattered around a light source at the center of the scene. The code for this scene setup can be found here.

In regards to rendering the shadows, the vertex and fragment shader are mostly similar to the original shadow mapping shaders. The difference is that the fragment shader no longer requires a fragment position in light space as we can now sample the depth values with a direction vector.

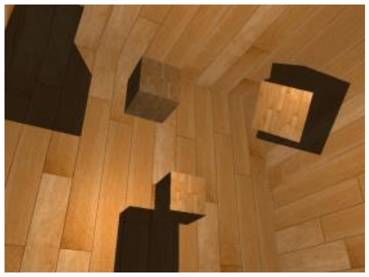
- Because of this, the vertex shader doesn't need to transform its position vectors to light space.
- The `sampler2D` uniform variable for the shadow map in the fragment shader becomes a `samplerCube` uniform and the `ShadowCalculation` function takes the current fragment's position as its argument instead of the fragment position in light space.
- We also include the light frustum's far plane, as that will be needed later for closest depth calculations.

Write the vertex and fragment shader for displaying shadows (without implementing the ShadowCalculations function just yet) using the depth cubemap texture.

- Omni-Shadows Vertex and Fragment Shaders

Define the ShadowCalculations function, which determines whether a fragment is in shadow (similar to the previous chapter). Then, draw the shadows in the scene.

- Don't forget to use the depth map debugging line in the fragment shader to check if your depth cubemap is working properly.
- ShadowCalculation Function + Rendering



With these shaders, we already get pretty good shadows, and this time, in all surrounding directions from a point light. With a point light positioned at the center of our scene, it should look like the image to the right.

We can also see the to-be shadowed regions on the inside wall, if we populate `FragColor` using our depth map debugging line.

PCF

Since omnidirectional shadow maps are based on the same principles of directional shadow mapping, it also has the same resolution-dependent artifacts (the jagged shadow edges). Percentage-closer filtering (PCF) allows us to smooth out these jagged edges by filtering multiple samples around the fragment position and average the results.

Implement PCF in the fragment shader as part of our shadow calculations.

- Omnidirectional PCF

As most of the samples we get using the PCF method above are redundant (in that they sample close to the original direction vector), it may make more sense to only sample in perpendicular direction of the sample direction vector.

- However, as there is no (easy) way to figure out which sub-directions are redundant, this becomes difficult.
  - One trick we can use is to take an array of offset directions that are all roughly seperable (e.g. each of them points in completely different directions). This will significantly reduce the number of sub-directions that are close together.
  - Here is an array containing 20 offset directions.

Adapt the PCF algorithm to take a fixed amount of samples from the above sampleOffsetDirections array and use them to sample the cubemap.

- PCF Sampling Cubemap with Offset Directions Array

Note how our new PCF implementation makes the shadows look very similar to our original PCF implementation, but requires less than a third of the samples.

Remember that the bias will require tweaking depending on the scene.

It is worth mentioning that using geometry shaders to generate a depth map isn't necessarily faster than rendering the scene 6 times for each face.

- Using a geometry shader like this has its own performance penalties that may outweigh the performance gain of using one in the first place.
  - This depends on the type of environment, the specific video card drivers, and other things.
- If you really care about performance, make sure to profile both methods and select the more efficient one for your scene.

Example 6-Times Rendering for Depth Cubemap

Tuesday, May 31, 2022 6:49 PM

```
for(unsigned int i = 0; i < 6; i++) {
    GLenum face = GL_TEXTURE_CUBE_MAP_POSITIVE_X + i;
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, face, depthCubemap, 0);
    BindViewMatrix(lightViewMatrices[i]);
    RenderScene();
}
```

Depth Cubemap

Tuesday, May 31, 2022 7:22 PM

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

// Allocates memory for the 6 faces of the cubemap
for (int i = 0; i < 6; ++i) {
   glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_DEPTH_COMPONENT, SHADOW_WIDTH, SHADOW_HEIGHT, 0,
       GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
}
// Configures the cubemap's texture parameter
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
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);

// Attaches the whole cubemap to the framebuffer, since its being used in one render pass by the geometry shader
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, depthCubemap, 0);
// We only care about the depth values, so we don't want to render to a color buffer
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Example Omnidirectional Shadow Map Rendering

Tuesday, May 31, 2022 7:50 PM

```
// 1. First, render to the depth cubemap
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
ConfigureShaderAndMatrices();
RenderScene();
glBindFramebuffer(GL_FRAMEBUFFER, 0);

// 2. Then, render the scene as normal with shadow mapping (using depth cubemap)
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
ConfigureShaderAndMatrices();
glBindTexture(GL_TEXTURE_CUBE_MAP, depthCubemap);
RenderScene();
```

Depth Cubemap Projection/View Matrices

Tuesday, May 31, 2022 8:03 PM

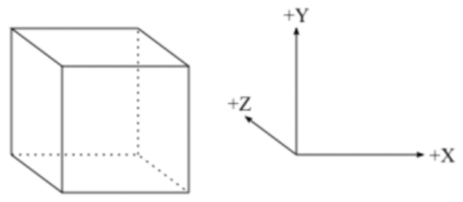
```
// Perspective Projection Matrix for Omnidirectional Shadow Mapping
float omniNearPlane = 1.0f;
float omniFarPlane = 25.0f;
glm::mat4 omniProjection = glm::perspective(glm::radians(90.0f), (float)SHADOW_WIDTH / (float)SHADOW_HEIGHT, omniNearPlane, o mniFarPlane);

// Light Space Matrices for Omnidirectional Shadow Mapping
std::vector<glm::mat4> shadowTransforms;
shadowTransforms.push_back(omniProjection * glm::lookAt(lightPos, lightPos + glm::vec3(1.0, 0.0, 0.0),
   glm::vec3(0.0, -1.0, 0.0)));    // right
shadowTransforms.push_back(omniProjection * glm::lookAt(lightPos, lightPos + glm::vec3( -1.0, 0.0, 0.0),
   glm::vec3(0.0, -1.0, 0.0)));    // left
shadowTransforms.push_back(omniProjection * glm::lookAt(lightPos, lightPos + glm::vec3(0.0, 1.0, 0.0),
   glm::vec3(0.0, 0.0, 1.0)));     // top
shadowTransforms.push_back(omniProjection * glm::lookAt(lightPos, lightPos + glm::vec3(0.0, -1.0, 0.0),
   glm::vec3(0.0, 0.0, -1.0)));    // bottom
shadowTransforms.push_back(omniProjection * glm::lookAt(lightPos, lightPos + glm::vec3(0.0, 0.0, 1.0),
   glm::vec3(0.0, -1.0, 0.0)));    // near
shadowTransforms.push_back(omniProjection * glm::lookAt(lightPos, lightPos + glm::vec3(0.0, 0.0, -1.0),
   glm::vec3(0.0, -1.0, 0.0)));    // far
```

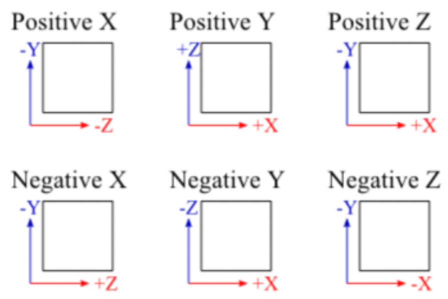
NOTE: As explained in the Cubemaps chapter, cubemaps are specified to follow the RenderMan specification, which assumes the image's origin to be in the upper left, contrary to the usual OpenGL behavior of having the image origin in the lower left. This is why things get swapped in the Y direction.

Use the image to the right to better understand the Up vectors used.

- Imagine looking at a face from inside the cube, and rotating it around its central axis. This is how you can determine what is "up" relative to the face.
- **NOTE:** Positive Z is away, as opposed to typical OpenGL orientation. You can thank RenderMan for this. Keep this in mind when thinking about cubemaps.

It is completely understandable if this is confusing. I wouldn't be terribly concerned with remembering exactly how it works. Just don't forget that cubemaps are oriented differently and reference this when necessary.





Omnidirectional Depth Map Vertex Shader

Tuesday, May 31, 2022 9:12 PM

omni_depth.vert

```
#version 330 core

layout (location = 0) in vec3 aPos;

uniform mat4 model;

void main() {
   gl_Position = model * vec4(aPos, 1.0);
}
```

Omnidirectional Depth Map Geometry Shader

Tuesday, May 31, 2022 9:26 PM

omni_depth.geom

```
#version 330 core

layout(triangles) in;
layout(triangle_strip, max_vertices = 18) out; // Outputting 6 triangles (6 * 3 = 18)

uniform mat4 shadowMatrices[6];

out vec4 fragPos; // Needed in fragment shader to calculate depth value (outputted after each EmitVertex call)

void main() {
   for (int face = 0; face < 6; ++face) { // For each face of the cubemap,
      gl_Layer = face; // Built-in variable that specifies to which face we render.
      for (int i = 0; i < 3; ++i) { // For each triangle vertex,
         fragPos = gl_in[i].gl_Position;
         gl_Position = shadowMatrices[face] * fragPos; // Transforms the vertex's position to light space.
         EmitVertex();
      }
      EndPrimitive();
   }
}
```

Omnidirectional Depth Map Fragment Shader

Tuesday, May 31, 2022 9:33 PM

omni_depth.frag

```
#version 330 core

in vec4 fragPos;

uniform vec3 lightPos;
uniform float farPlane;

void main() {
   // Gets the distance between fragment and light source
   float lightDistance = length(fragPos.xyz - lightPos);

   // Maps to [0,1] range by dividing by farPlane (if your near plane is not 1.0, you will need to modify this)
   lightDistance = lightDistance / farPlane;

   // Writes lightDistance as the modified depth
   gl_FragDepth = lightDistance;
}
```

Example Depth Cubemap Rendering

Wednesday, June 1, 2022 8:47 PM

```
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
shader.use();
// ... send uniforms to shader (including light's far_plane value)
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, depthCubemap);
// ... bind other textures
RenderScene();
```

renderScene Function

Thursday, June 2, 2022 8:34 PM

renderScene function

```
void renderScene(const Shader& shader, const unsigned int& cubeVAO) {
   // Room cube
   glm::mat4 model = glm::mat4(1.0f);
   model = glm::scale(model, glm::vec3(5.0f));
   glUniformMatrix4fv(glGetUniformLocation(shader.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
   // We disable culling here since we render 'inside' the cube instead of the usual 'outside', which throws off the normal culling
   // methods.
   glDisable(GL_CULL_FACE);
   // A small little hack to invert normals when drawing cube from the inside so lighting still works.
   glUniform1i(glGetUniformLocation(shader.id, "reverseNormals"), 1);
   glBindVertexArray(cubeVAO);
   glDrawArrays(GL_TRIANGLES, 0, 36);
   glUniform1i(glGetUniformLocation(shader.id, "reverseNormals"), 0); // and of course disable it
   glEnable(GL_CULL_FACE);

   // Cubes
   model = glm::mat4(1.0f);
   model = glm::translate(model, glm::vec3(4.0f, -3.5f, 0.0));
   model = glm::scale(model, glm::vec3(0.5f));
   glUniformMatrix4fv(glGetUniformLocation(shader.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
   glBindVertexArray(cubeVAO);
   glDrawArrays(GL_TRIANGLES, 0, 36);

   model = glm::mat4(1.0f);
   model = glm::translate(model, glm::vec3(2.0f, 3.0f, 1.0));
   model = glm::scale(model, glm::vec3(0.75f));
   glUniformMatrix4fv(glGetUniformLocation(shader.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
   glBindVertexArray(cubeVAO);
   glDrawArrays(GL_TRIANGLES, 0, 36);

   model = glm::mat4(1.0f);
   model = glm::translate(model, glm::vec3(-3.0f, -1.0f, 0.0));
   model = glm::scale(model, glm::vec3(0.5f));
   glUniformMatrix4fv(glGetUniformLocation(shader.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
   glBindVertexArray(cubeVAO);
   glDrawArrays(GL_TRIANGLES, 0, 36);

   model = glm::mat4(1.0f);
   model = glm::translate(model, glm::vec3(-1.5f, 1.0f, 1.5));
   model = glm::scale(model, glm::vec3(0.5f));
   glUniformMatrix4fv(glGetUniformLocation(shader.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
   glBindVertexArray(cubeVAO);
   glDrawArrays(GL_TRIANGLES, 0, 36);

   model = glm::mat4(1.0f);
   model = glm::translate(model, glm::vec3(-1.5f, 2.0f, -3.0));
   model = glm::rotate(model, glm::radians(60.0f), glm::normalize(glm::vec3(1.0, 0.0, 1.0)));
   model = glm::scale(model, glm::vec3(0.75f));
   glUniformMatrix4fv(glGetUniformLocation(shader.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
   glBindVertexArray(cubeVAO);
   glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

Omni-Shadows Vertex and Fragment Shaders

Thursday, June 2, 2022 3:55 PM

NOTE: These shaders are largely the same as the directional shadow mapping ones, with a few modifications to fit omnidirectional shadow mapping's needs, as previously discussed.

omni_shadows.vert

```
#version 330 core

layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;

out VS_OUT {
   vec3 fragPos;
   vec3 normal;
   vec2 texCoords;
} vs_out;

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

uniform bool reverseNormals;

void main() {
   vs_out.fragPos = vec3(model * vec4(aPos, 1.0));
   if (reverseNormals) { // A slight hack to ensure the outer large cube displays light from the inside instead of the outside
      vs_out.normal = normalize(transpose((mat3(model))) * (-1.0 * aNormal));
   }
   else {
      vs_out.normal = normalize(transpose(inverse(mat3(model))) * aNormal);
   }
   vs_out.texCoords = aTexCoords;
   gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

omni_shadows.frag

```
#version 330 core

struct Material {
   sampler2D diffuse;
   sampler2D specular;

   float shininess;
};
uniform Material material;

struct PointLight {
   vec3 position;

   vec3 ambient;
   vec3 diffuse;
   vec3 specular;
};
uniform PointLight pointLight;

in VS_OUT {
   vec3 fragPos;
   vec3 normal;
   vec2 texCoords;
} fs_in;

out vec4 FragColor;

uniform vec3 viewPos;
uniform samplerCube shadowMap;
uniform float farPlane;

float closestDepth; // Used for viewing the depth map to debug

vec3 CalcPointLight(PointLight light, vec3 viewDir);

void main() {
   vec3 viewDir = normalize(viewPos - fs_in.fragPos);

   vec3 color = CalcPointLight(pointLight, viewDir);
   FragColor = vec4(color, 1.0);

   // Gamma correction
   float gamma = 2.2;
   FragColor.rgb = pow(FragColor.rgb, vec3(1.0 / gamma));
   //FragColor = vec4(vec3(closestDepth / farPlane), 1.0); // This is for viewing the depth map (for debugging)
}

// Calculates whether or not the current fragment is in shadow
float ShadowCalculation(PointLight light) {
   // To be discussed
}

vec3 CalcPointLight(PointLight light, vec3 viewDir) {
   // Ambient Lighting
   vec3 ambient = light.ambient;

   // Diffuse Lighting
   vec3 lightDir = normalize(light.position - fs_in.fragPos);
   float diff = max(dot(lightDir, fs_in.normal), 0.0);
   vec3 diffuse = light.diffuse * diff;

   // Blinn-Phong Specular Lighting
   vec3 halfwayDir = normalize(viewDir + lightDir);
   float spec = pow(max(dot(fs_in.normal, halfwayDir), 0.0), material.shininess);
   vec3 specular = light.specular * spec * vec3(texture(material.specular, fs_in.texCoords));

   // Shadow Calculations
   float shadow = ShadowCalculation(light);

   return (ambient + (1.0 - shadow) * (diffuse + specular)) * vec3(texture(material.diffuse, fs_in.texCoords));
}
```

ShadowCalculation Function + Rendering

Thursday, June 2, 2022 4:14 PM

ShadowCalculation function

```
// Calculates whether or not the current fragment is in shadow
float ShadowCalculation(PointLight light) {
   // Gets the difference vector of the fragment and light position
   vec3 fragToLight = fs_in.fragPos - light.position;
   // Uses fragToLight as a direction vector for sampling the cubemap
   closestDepth = texture(shadowMap, fragToLight).r; // Currently in the range [0,1]
   // Transforms closestDepth to the range [0, farPlane]
   closestDepth *= farPlane;
   // Retrieves the linear depth as the length between the current fragment and the light source
   float currentDepth = length(fragToLight);

   // Reduces shadow acne
   float bias = 0.05;
   // Compares both depth values to determine if the fragment is in shadow or not
   float shadow = currentDepth - bias > closestDepth ? 1.0 : 0.0;

   return shadow;
}
```

main()

```
Shader omniDepthShader{ "Shaders/omni_depth.vert", "Shaders/omni_depth.frag", "Shaders/omni_depth.geom" };
Shader omniShadowShader{ "Shaders/omni_shadows.vert", "Shaders/omni_shadows.frag" };
```

Render Loop

```
glClearColor(0.2f, 0.2f, 0.2f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Generating the depth cubemap
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);

glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);

omniDepthShader.use();
for (int i = 0; i < 6; ++i) {
   glUniformMatrix4fv(glGetUniformLocation(omniDepthShader.id, ("shadowMatrices[" + std::to_string(i) + "]").c_str()),
         1, GL_FALSE, glm::value_ptr(shadowTransforms[i]));
}
glUniform3fv(glGetUniformLocation(omniDepthShader.id, "lightPos"), 1, glm::value_ptr(lightPos));
glUniform1f(glGetUniformLocation(omniDepthShader.id, "farPlane"), omniFarPlane);

renderScene(omniDepthShader, megaCubeVAO);

glBindFramebuffer(GL_FRAMEBUFFER, 0);

// Drawing the scene with shadows using the depth map
glViewport(0, 0, WIDTH, HEIGHT);

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Creates the model matrix for the light cube
glm::mat4 objModel{ 1.0f };
// Creates the view matrix (camera)
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);

// Draw the scene normally using the new omnidirectional shadow shader
omniShadowShader.use();

planksDiffuseTexture.bind();
fullSpecularTexture.bind();
glActiveTexture(GL_TEXTURE2);
glBindTexture(GL_TEXTURE_CUBE_MAP, depthCubemap);
```

```
// Sets the vertex shader uniforms
glUniformMatrix4fv(glGetUniformLocation(omniShadowShader.id, "view"), 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(glGetUniformLocation(omniShadowShader.id, "projection"), 1, GL_FALSE, glm::value_ptr(projection));
// Sets the fragment shader uniforms
// Material uniforms
glUniform1i(glGetUniformLocation(omniShadowShader.id, "material.diffuse"), planksDiffuseTexture.getTexUnit());
glUniform1i(glGetUniformLocation(omniShadowShader.id, "material.specular"), fullSpecularTexture.getTexUnit());
glUniform1f(glGetUniformLocation(omniShadowShader.id, "material.shininess"), 32.0f);
// Other fragment shader uniforms
glUniform3fv(glGetUniformLocation(omniShadowShader.id, "viewPos"), 1, glm::value_ptr(camera.position));
glUniform1i(glGetUniformLocation(omniShadowShader.id, "shadowMap"), 2);
glUniform1f(glGetUniformLocation(omniShadowShader.id, "farPlane"), omniFarPlane);
// Sets the point light properties
glUniform3fv(glGetUniformLocation(omniShadowShader.id, "pointLight.position"), 1,
    glm::value_ptr(lightPos));
glUniform3fv(glGetUniformLocation(omniShadowShader.id, "pointLight.ambient"), 1,
    glm::value_ptr(glm::vec3(0.05f) * lightColor));
glUniform3fv(glGetUniformLocation(omniShadowShader.id, "pointLight.diffuse"), 1,
    glm::value_ptr(glm::vec3(0.8f) * lightColor));
glUniform3fv(glGetUniformLocation(omniShadowShader.id, "pointLight.specular"), 1,
    glm::value_ptr(glm::vec3(1.0f) * lightColor));

renderScene(omniShadowShader, megaCubeVAO);
```

Omnidirectional PCF

Thursday, June 2, 2022 9:13 PM

ShadowCalculation function

```
// Calculates whether or not the current fragment is in shadow
float ShadowCalculation(PointLight light) {
   // Gets the difference vector of the fragment and light position
   vec3 fragToLight = fs_in.fragPos - light.position;
   // Retrieves the linear depth as the length between the current fragment and the light source
   float currentDepth = length(fragToLight);

   float shadow = 0.0;
   float bias = 0.5;
   float samples = 4.0f;
   float offset = 0.1f;

   // Takes samples^3 samples for each fragment and averges the resulting shadow value to smooth shadows
   for (float x = -offset; x < offset; x += offset / (samples * 0.5)) {
      for (float y = -offset; y < offset; y += offset / (samples * 0.5)) {
         for (float z = -offset; z < offset; z += offset / (samples * 0.5)) {
            // Gets the depth value of the fragment with offset xyz
            closestDepth = texture(shadowMap, fragToLight + vec3(x, y, z)).r;
            // Maps closestDepth from [0,1] to [0, farPlane]
            closestDepth *= farPlane;
            // Sums up the shadow value of each sample
            shadow += currentDepth - bias > closestDepth ? 1.0 : 0.0;
         }
      }
   }
   // Averages the shadow value based on the number of samples to determine the shadow intensity
   shadow /= (samples * samples * samples);

   return shadow;
}
```

Sample Offset Directions Array

Thursday, June 2, 2022 9:22 PM

```
vec3 sampleOffsetDirections[20] = vec3[] (
   vec3( 1,  1,  1), vec3( 1, -1,  1), vec3(-1, -1,  1), vec3(-1,  1,  1),
   vec3( 1,  1, -1), vec3( 1, -1, -1), vec3(-1, -1, -1), vec3(-1,  1, -1),
   vec3( 1,  1,  0), vec3( 1, -1,  0), vec3(-1, -1,  0), vec3(-1,  1,  0),
   vec3( 1,  0,  1), vec3(-1,  0,  1), vec3( 1,  0, -1), vec3(-1,  0, -1),
   vec3( 0,  1,  1), vec3( 0, -1,  1), vec3( 0, -1, -1), vec3( 0,  1, -1)
);
```

PCF Sampling Cubemap with Offset Directions Array

Thursday, June 2, 2022 9:28 PM

ShadowCalculation function

```
// Calculates whether or not the current fragment is in shadow
float ShadowCalculation(PointLight light) {
   // Gets the difference vector of the fragment and light position
   vec3 fragToLight = fs_in.fragPos - light.position;
   // Retrieves the linear depth as the length between the current fragment and the light source
   float currentDepth = length(fragToLight);

   float shadow = 0.0;
   float bias = 0.15;
   int samples = 20;
   float viewDistance = length(viewPos - fs_in.fragPos);
   //float diskRadius = 0.05; // Scales the offsets
   float diskRadius = (1.0 + (viewDistance / farPlane)) / 25.0; // Makes shadows sharper when the viewer is closer, softer when the
                                                                // viewer is further away

   // Samples offsets (scaled by diskRadius) around the original fragToLight vector to sample from the cubemap
   for (int i = 0; i < samples; ++i) {
      float closestDepth = texture(shadowMap, fragToLight + sampleOffsetDirections[i] * diskRadius).r;
      closestDepth *= farPlane;
      shadow += currentDepth - bias > closestDepth ? 1.0 : 0.0;
   }
   shadow /= float(samples);

   return shadow;
}
```