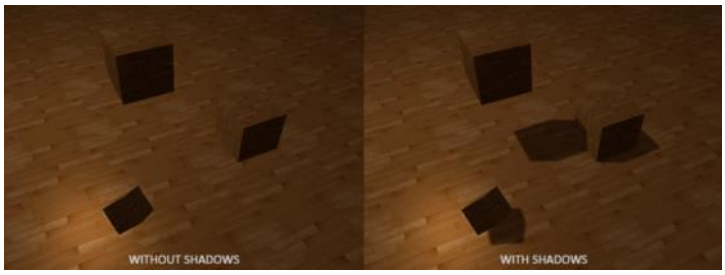
SHADOW MAPPING

- Since the cube vertices and plane vertices I've been using up to this point have not been structure with counter-clockwise winding order in mind, you should replace those vertices with these updated vertices and update your scene to follow along.



Shadows are the result of the absence of light due to **occlusion**. When a light source's rays do not hit an object because it gets occluded by some other object, the object is in shadow.

- In lit scenes, shadows add realism and make it easier for a viewer to observe spatial relationships between objects by yielding a greater sense of depth.
- An example of shadows can be seen in the image to the right.
  - Note how you can only really tell that one of the cubes is floating above the other because its shadow is so far away.

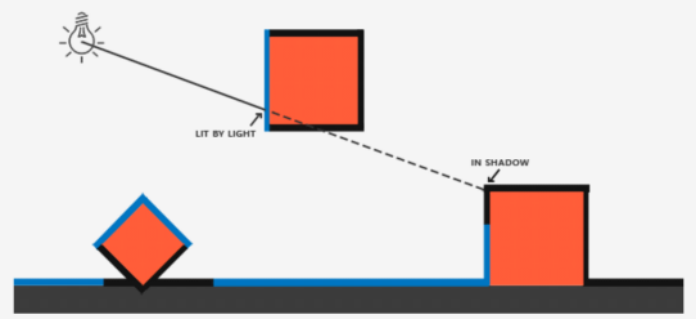
Shadows are tricky to implement because in current real-time (rasterized graphics) research, a perfect shadow algorithm hasn't been developed yet. There are several good approximation techniques, but they all have their ups and downs.

- The technique we'll be using is called **shadow mapping**, which is a technique that is relatively easy to implement and doesn't cost too much in performance.
  - Shadow mapping also easily extends into more advanced algorithms, like omnidirectional shadow maps (which we'll discuss in the next chapter) or cascaded shadow maps.

Shadow Mapping

The idea behind shadow mapping is that we render the scene from the light's point of view; everything we see from the light's perspective is lit and everything we can't see must be in shadow.

- Example: Imagine a floor section with a large box between itself and a light source. Since the light source will see this box and not the floor section when looking in its direction, that specific floor section should be in shadow.

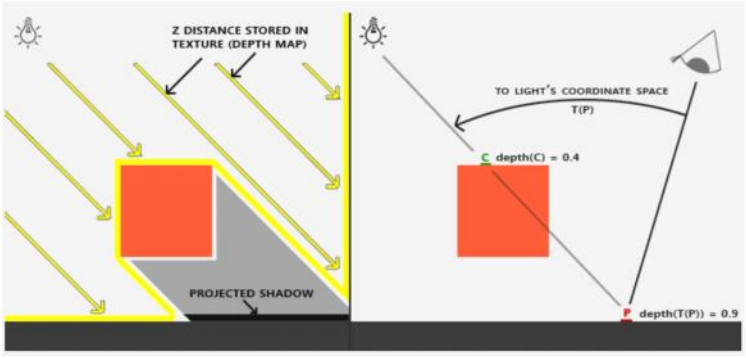


Take a look at the image to the right.

- The blue lines represent the fragments the light source can see and the black lines represent the fragments the light source cannot see (the occluded fragments).
- If we draw a ray from the light source to a fragment on the right-most box (as shown in the image), we can see the ray first hits the floating container before hitting the right-most container.
  - This is how we know that that specific fragment of the right-most box is in shadow, and should thus not be lit by the light source.
  - We want to get the point on the ray where it first hit an object and compare this closest point to other points on this ray. Then, we do a basic test to see if a test point's ray position is further down the ray than the closest point. If so, the test point must be in shadow.

Rather than casting and iterating through possibly thousands of individual light rays to determine if certain fragments are in shadow or not (which is extremely inefficient and thus not very useful for real-time rendering), we can use the depth buffer to achieve a similar result with less of a performance implication.

- You may remember from the depth buffer chapter, a value in the depth buffer corresponds to the depth of a fragment, clamped to `[0,1]`, from the camera's point of view.
- If we render the scene from the light's perspective and store the resulting depth values in a texture, we can sample the closest depth values as seen from the light's perspective.
  - We'll store all these depth values in a texture that we call a **depth map** or **shadow map**.



Take a look at the image to the right.

- The left part shows a directional light source (all light rays are parallel) casting a shadow on the surface below the cube.
  - Using the depth values stored in the depth map, we find the closest point and use that to determine whether fragments are in shadow.
  - We create the depth map by rendering the scene from the light's perspective using a view and projection matrix specific to that light source. This projection and view matrix together form a transformation ***T*** that transforms any 3D position to the light's (visible) coordinate space.
  - **NOTE:** A directional light doesn't have a position as it's modelled to be infinitely far away. However, for the sake of shadow mapping, we need to render the scene from a light's perspective and thus render the scene from a position somewhere along the lines of the light direction.
- The right part shows the same directional light and the viewer.
  - We render a fragment at point ***P*** for which we have to determine whether it is in shadow.
    - To do this, we first transform point ***P*** to the light's coordinate space using ***T***. Since point ***P*** is now as seen from the light's perspective, its z-coordinate corresponds to its depth. *In this example, the fragment's depth from the light's perspective is `0.9`.*
    - Using point ***P***, we can also index the depth/shadow map to obtain the closest visible depth from the light's perspective, which is at point ***C***. *In this example, the sampled depth of point **C** is `0.4`.*
    - Since indexing the depth map returns a depth smaller than the depth at point ***P***, we can conclude point ***P*** is occluded and, thus, in shadow.

Shadow mapping, therefore, consists of two passes:

1. When we render the depth map.
2. When we render the scene as normal, using the generated depth map to calculate whether fragments are in shadow.

The Depth Map

The depth map is the depth texture, as rendered from the light's perspective, that we'll be using for testing for shadows.

- Because we need to store the rendered result of a scene into a texture, we're going to have to use framebuffers again.

Create a framebuffer and attach a texture for storing depth values.

- Depth Map FBO Setup

With a properly configured framebuffer that renders depth values to a texture, we can start the first pass: generating the depth map.
- When combined with the second pass, the complete rendering stage will look a bit like this.

Light Space Transform

In the previous snippet of code, the ConfigureShaderAndMatrices function might be a little less obvious in the first pass than the second.
- In the second pass, we make sure the proper projection and view matrices are set and we set the relevant model matrices per object.
- However, in the first pass, we need to use a different projection and view matrix to render the scene from the light's point of view.
  - Because we're modelling a directional light source, all its light rays are parallel. For this reason, we use an orthographic projection matrix for the light source, where there is no perspective deform.
  - To create a view matrix to transform each object so they are visible from the light's point of view, we can use a look-at matrix, with the light source's position looking at the scene's center.

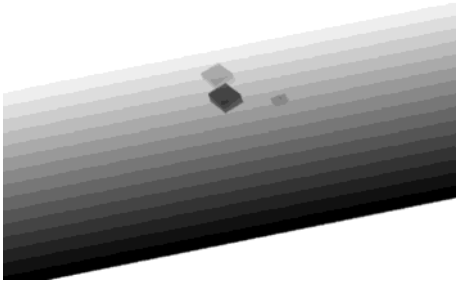
Create the light space transformation matrix by computing the light's projection and view matrices.
- Light Projection and View Matrices

The `lightSpaceMatrix` (as calculated above) is the transformation matrix that we earlier denoted as $\boldsymbol{T}$.
- While we could just inject our light space equivalents of the projection and view matrices into our current shaders, we don't need to worry about expensive fragment (lighting) calculations, since we only care about the depth values for the depth map.
- To save performance, we will instead use a different, simpler shader for rendering to the depth map.

Render to Depth Map

Write a shader (vertex and fragment) for generating the depth map.
- Vertex and Fragment Shaders for Depth Map

Rendering the depth/shadow map now effectively becomes this.



- The result of this is a filled depth buffer holding the closest depth of each visible fragment from the light's perspective. By rendering this texture onto a 2D quad that fills the screen (similar to what we did in the post-processing section at the end of the framebuffers chapter), we get something like the image to the right.
  - The white area at the bottom of the image is due to the wooden plane being outside the near clipping plane of the light.
- If you'd like to recreate this image, here is the fragment shader and render loop used to render the depth map onto a quad.

NOTE: There are some subtle changes when displaying depth using a perspective projection matrix instead of an orthographic projection matrix, as depth is non-linear when using perspective projection. We will discuss this more at the end of this chapter.

Rendering Shadows

With a properly generated depth map, we can start rendering the actual shadows.

To calculate whether a fragment is in shadow, we will do the following:
1. Convert the fragment's light space coordinates in clip space to normalized device coordinates.
   - Perspective division (division of the xyz components of the vector by the w component) only automatically occurs when clip space coordinates are output to `gl_Position`, so we otherwise have to do the calculation manually.
2. Convert those NDC coordinates to a range of [0, 1].
   - This puts the light space coordinates of the fragment in the same range as the values in the shadow map.
3. Get the depth value of the closest fragment by sampling the shadow map.
4. Compare the closest depth value with the current depth value to determine if the fragment has a shadow value of `1.0` or `0.0` (in shadow and not in shadow, respectively).
5. Multiply the diffuse and specular values in the fragment color calculation by the inverse of the shadow value.
   - We multiply by the inverse of the shadow value since we want to know how much a fragment is not in shadow, which is the opposite of what the shadow value means.
   - Since the diffuse and specular values effect the color of the fragment based on its exposure to the light source, we basically negate those if the fragment is in shadow.
   - We don't negate the ambient values as shadows can't hide things from ambient lighting.

Implement shadows using the generated depth/shadow map.
- **NOTE:** Your result will look messed up with many visual artifacts. This is normal.
- Rendering Shadows

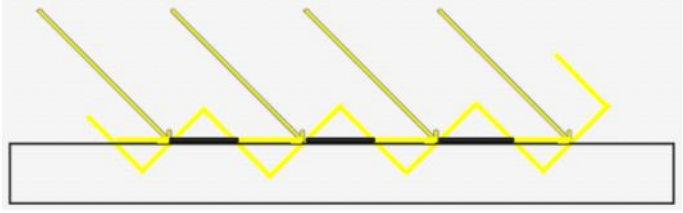
Improving Shadow Maps

We have shadows in our scene now, but we also have a few holes in our implementation (as displayed by the clear visual artifacts in the imaage to the right) that we must address.

Shadow Acne

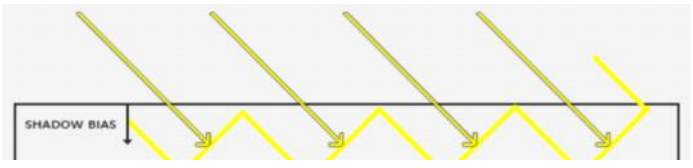
As you can see, there is a Moiré-like pattern that (albeit very interesting) is making our scene look bad. We can see a large part of the floor quad rendered with obvious black lines in an alternating fashion. This shadow mapping artifact is known as **shadow acne**.

Take a look at the image to the right.



- Because the shadow map is limited by resolution, multiple fragments can sample the same value from the depth map when they're relatively far away from the light source.
- The yellow tilted panels each represent a single texel of the depth map.
  - As you can see, several fragments sample the same depth sample.
  - This only becomes an issue when the light source looks at an angle towards the surface, as in that case, the depth map is also rendered from an angle. Several fragments then access the same tilted depth texel while some are above and some below the floor.
    - Because of this, some fragments are considered to be in shadow and some are not, giving the striped pattern from the image.

We can solve the above issue with a small hack called **shadow bias**.



- Shadow bias is where we simply offset the depth of the surface (or the shadow map) by a small bias amount such that the fragments are not incorrectly considered above the surface, as shown in

We can solve the above issue with a small hack called **shadow bias**.

- Shadow bias is where we simply offset the depth of the surface (or the shadow map) by a small bias amount such that the fragments are not incorrectly considered above the surface, as shown in the image to the right.
- With the bias applied, all the samples get a depth smaller than the surface's depth and, thus, the entire surface is correctly lit without any shadows.
- While we could implement a flat shadow bias for our scene, you can probably image that we would need a higher bias for situations where the angle between the light and the surface are steep to prevent shadow acne.

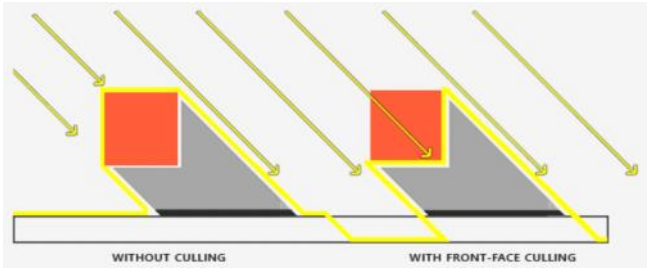
Implement a shadow bias, based on the angle between the light source and the surface, with a max of `0.05` and and min of `0.005` when determining if a fragment is in shadow.

- Shadow Bias

Choosing the bias value(s) requires tweaking and will typically be unique for each scene.

Peter Panning

A disadvantage of using a shadow bias is that you're applying an offset to the actual depth of objects. If the bias becomes large enough, you will
be able to see an offset of the shadows compared to the actual object locations, as you can see (with an exaggerated bias) in the image to the right.
This shadow artifact is called **peter panning**, since objects seem slightly detached from their shadows.



- We can eliminate most of the peter panning by using front face culling when rendering the depth map, as shown in the image to the right.
  - Because we only need depth values from the depth map, it shouldn't matter for solid objects whether we take the depth of their front or back faces.
  - Using back face depths doesn't give wrong results as it doesn't matter if we have shadows inside objects; we can't see there anyways.

Reduce the peter panning issue by culling front faces before drawing the depth map.

- Reducing Peter Panning

You will probably notice that, for you cube that is sitting directly on top of the plane, the bottom face is not contributing a shadow. This is because it has the same depth as the floor, so the shadow bias causes it to not generate a shadow, which will lead to a shadow artifact as seen in the image to the right.

- Because of this, the above method of reducing peter panning should be used situationally.
- Additionally, this method should only be used for solid objects because it relies on all back faces being completely solid.

Over Sampling

Another visual discrepancy is that regions outside the light's visible frustum are considered to be in shadow while they're (usually) not.

- This happens because projected coordinates outside the light's frustum are higher than `1.0` and, thus, sample the depth texture outside its default range of `[0, 1]`.
- Because of the depth texture's wrapping method, we will get incorrect depth results not based on the real depth values from the light source.
- You can see in the image to the right that a large area is in shadow (yellow circle) and that the shadows are being copied over to other fragments of the plane (red circle). This is happening because we used GL_REPEAT as the depth map's wrapping option.
  - We would rather have all coordinates outside the depth map's range have a depth of `1.0`, meaning they would never be in shadow. We can do this by configuring a texture border color and the depth map's texture wrap options to `GL_CLAMP_TO_BORDER`.
    - By doing this, whenever we sample outside the depth map's `[0, 1]` coordinate range, the texture function will always return a depth of `1.0`, producing a shadow value of `0.0`, which fixes the duplicated shadows.
  - The large dark area contains coordinates outside the far plane of the light's orthographic frustum. You can see that this dark region always occurs at the far end of the light source's frustum by looking at the shadow directions. A light-space projected fragment coordinate is further than the light's far plane when its z-coordinate is larger than `1.0`.
    - Because of this, the `GL_CLAMP_TO_BORDER` wrapping method doesn't work anymore since we compare the coordinate's z-component with the depth map values; this always returns true for z-components larger than `1.0`.
    - The solution is to force the shadow value to 0.0 whenever the projected vector's z-coordinate is larger than `1.0`.

Fix the over sampling of the depth map by using the above methodologies.

- Fixing Over Sampling

The result of this does mean that we only have shadows where the projected fragment coordinates sit inside the depth map range, so anything outside the light frustum will have no visible shadows.

- As games usually make sure this only occurs in the distance, it is a much more plausible effect than the obvious black regions and duplicated shadows we had before.

PCF

The shadows may look alright from a distance, but if we zoom in, the resolution dependency of shadow mapping quickly becomes apparent, as seen in the image to the right.

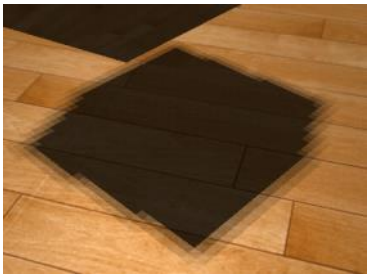
- Because the depth map has a fixed resolution, the depth frequently spans more than one fragment per texel. As a result, multiple fragments sample the same depth value from the depth map and come to the same shadow conclusions, which produces these jagged, blocky edges.
- You reduce the jagged edges by raising the depth map resolution, trying to fit the light frustum closer to the scene, or (partially) by using a method called **PCF**, or **percentage-closer filtering** (which is what we'll be using).

PCF is a term that hosts many different filtering functions that produce softer shadows, making them appear less blocky or hard.

- The idea is to sample more than once from the depth map, each time with slightly different texture coordinates. For each individual sample, we check whether it is in shadow or not. All the sub-results are then combined and averaged and we get a nice, soft-looking shadow.

PCF is a term that hosts many different filtering functions that produce softer shadows, making them appear less blocky or hard.

- The idea is to sample more than once from the depth map, each time with slightly different texture coordinates. For each individual sample, we check whether it is in shadow or not. All the sub-results are then combined and averaged and we get a nice, soft-looking shadow.



Write a simple PCF implementation that samples the surrounding texels of the depth map and averages the results to determine the strength of the shadow.

- Percentage-Closer Filtering

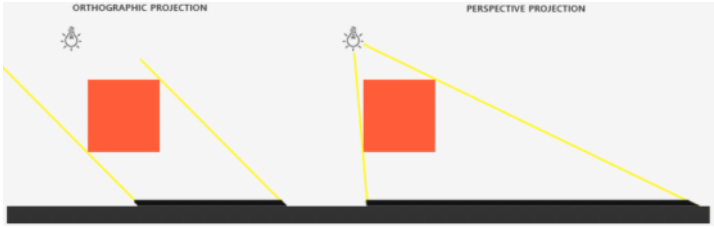
From a distance, the shadows look much softer. The resolution artifacts of the shadow map are still present, but the PCF makes it much less intense.

PCF is actually much more extensive than the implementation we've used, with quite a few techniques to considerably improve the quality of soft shadows, but that will not be covered in this chapter.

Orthographic vs Perspective

These is a difference between rendering the depth map with an orthographic or a perspective projection matrix.

- An orthographic projection matrix does not deform the scene with perspective, so all view/light rays are parallel. This makes it a great projection matrix for directional lights.
- A perspective projection matrix, however, does deform all vertices based on perspective, which gives different results.
- The image to the right shows the different shadow regions of both projection methods.



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

Another subtle difference with using a perspective matrix is that visualizing the depth buffer will often give an almost completely white result.

- This happens because, with perspective projection, the depth is transformed to non-linear depth values with most of its noticeable range close to the near plane.
- To be able to properly view the depth values, you first want to transform the non-linear depth values to linear, as we discussed in the depth testing chapter, like so.
  - The above code shows depth values similar to what we've seen with orthographic projection, however, it should be noted that this is only useful for debugging. The depth checks remain the same with orthographic and perspective projection matrices as the relative depths do not change.

CCW Winding Order Plane & Cube Vertices

Friday, May 27, 2022 7:55 PM

```
float planeVertices[] = {
   // Positions            // Normals         // Tex Coords
    25.0f, -0.5f,  25.0f,  0.0f, 1.0f, 0.0f,  25.0f,  0.0f,
   -25.0f, -0.5f, -25.0f,  0.0f, 1.0f, 0.0f,   0.0f, 25.0f,
   -25.0f, -0.5f,  25.0f,  0.0f, 1.0f, 0.0f,   0.0f,  0.0f,

    25.0f, -0.5f,  25.0f,  0.0f, 1.0f, 0.0f,  25.0f,  0.0f,
    25.0f, -0.5f, -25.0f,  0.0f, 1.0f, 0.0f,  25.0f, 25.0f,
   -25.0f, -0.5f, -25.0f,  0.0f, 1.0f, 0.0f,   0.0f, 25.0f
};
```

```
float vertices[] = {
   // Positions           // Normals      // Tex Coords
   // Back face
   -1.0f, -1.0f, -1.0f,    0.0f,  0.0f,   -1.0f, 0.0f, 0.0f, // bottom-left
    1.0f,  1.0f, -1.0f,    0.0f,  0.0f,   -1.0f, 1.0f, 1.0f, // top-right
    1.0f, -1.0f, -1.0f,    0.0f,  0.0f,   -1.0f, 1.0f, 0.0f, // bottom-right
    1.0f,  1.0f, -1.0f,    0.0f,  0.0f,   -1.0f, 1.0f, 1.0f, // top-right
   -1.0f, -1.0f, -1.0f,    0.0f,  0.0f,   -1.0f, 0.0f, 0.0f, // bottom-left
   -1.0f,  1.0f, -1.0f,    0.0f,  0.0f,   -1.0f, 0.0f, 1.0f, // top-left
   // Front face
   -1.0f, -1.0f,  1.0f,    0.0f,  0.0f,    1.0f, 0.0f, 0.0f, // bottom-left
    1.0f, -1.0f,  1.0f,    0.0f,  0.0f,    1.0f, 1.0f, 0.0f, // bottom-right
    1.0f,  1.0f,  1.0f,    0.0f,  0.0f,    1.0f, 1.0f, 1.0f, // top-right
    1.0f,  1.0f,  1.0f,    0.0f,  0.0f,    1.0f, 1.0f, 1.0f, // top-right
   -1.0f,  1.0f,  1.0f,    0.0f,  0.0f,    1.0f, 0.0f, 1.0f, // top-left
   -1.0f, -1.0f,  1.0f,    0.0f,  0.0f,    1.0f, 0.0f, 0.0f, // bottom-left
   // Left face
   -1.0f,  1.0f,  1.0f,   -1.0f,  0.0f,    0.0f, 1.0f, 0.0f, // top-right
   -1.0f,  1.0f, -1.0f,   -1.0f,  0.0f,    0.0f, 1.0f, 1.0f, // top-left
   -1.0f, -1.0f, -1.0f,   -1.0f,  0.0f,    0.0f, 0.0f, 1.0f, // bottom-left
   -1.0f, -1.0f, -1.0f,   -1.0f,  0.0f,    0.0f, 0.0f, 1.0f, // bottom-left
   -1.0f, -1.0f,  1.0f,   -1.0f,  0.0f,    0.0f, 0.0f, 0.0f, // bottom-right
   -1.0f,  1.0f,  1.0f,   -1.0f,  0.0f,    0.0f, 1.0f, 0.0f, // top-right
   // Right face
    1.0f,  1.0f,  1.0f,    1.0f,  0.0f,    0.0f, 1.0f, 0.0f, // top-left
    1.0f, -1.0f, -1.0f,    1.0f,  0.0f,    0.0f, 0.0f, 1.0f, // bottom-right
    1.0f,  1.0f, -1.0f,    1.0f,  0.0f,    0.0f, 1.0f, 1.0f, // top-right
    1.0f, -1.0f, -1.0f,    1.0f,  0.0f,    0.0f, 0.0f, 1.0f, // bottom-right
    1.0f,  1.0f,  1.0f,    1.0f,  0.0f,    0.0f, 1.0f, 0.0f, // top-left
    1.0f, -1.0f,  1.0f,    1.0f,  0.0f,    0.0f, 0.0f, 0.0f, // bottom-left
   // Bottom face
   -1.0f, -1.0f, -1.0f,    0.0f, -1.0f,    0.0f, 0.0f, 1.0f, // top-right
    1.0f, -1.0f, -1.0f,    0.0f, -1.0f,    0.0f, 1.0f, 1.0f, // top-left
    1.0f, -1.0f,  1.0f,    0.0f, -1.0f,    0.0f, 1.0f, 0.0f, // bottom-left
    1.0f, -1.0f,  1.0f,    0.0f, -1.0f,    0.0f, 1.0f, 0.0f, // bottom-left
   -1.0f, -1.0f,  1.0f,    0.0f, -1.0f,    0.0f, 0.0f, 0.0f, // bottom-right
   -1.0f, -1.0f, -1.0f,    0.0f, -1.0f,    0.0f, 0.0f, 1.0f, // top-right
   // Top face
   -1.0f,  1.0f, -1.0f,    0.0f,  1.0f,    0.0f, 0.0f, 1.0f, // top-left
    1.0f,  1.0f , 1.0f,    0.0f,  1.0f,    0.0f, 1.0f, 0.0f, // bottom-right
    1.0f,  1.0f, -1.0f,    0.0f,  1.0f,    0.0f, 1.0f, 1.0f, // top-right
    1.0f,  1.0f,  1.0f,    0.0f,  1.0f,    0.0f, 1.0f, 0.0f, // bottom-right
   -1.0f,  1.0f, -1.0f,    0.0f,  1.0f,    0.0f, 0.0f, 1.0f, // top-left
   -1.0f,  1.0f,  1.0f,    0.0f,  1.0f,    0.0f, 0.0f, 0.0f  // bottom-left
};
```

This is the superior rendition of the vertex data for a cube thus far. It contains position, normal, and UV data, as well as having a CCW winding order on the triangles.

Scene Updates

Sunday, May 29, 2022 9:29 PM

main()

```
// Cube VAO
unsigned int megaCubeVAO;
glGenVertexArrays(1, &megaCubeVAO);
glBindVertexArray(megaCubeVAO);

unsigned int megaCubeVBO;
glGenBuffers(1, &megaCubeVBO);
glBindBuffer(GL_ARRAY_BUFFER, megaCubeVBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), &vertices, GL_STATIC_DRAW);

// Positions
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// Normals
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);
// Texture Coords
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void*)(6 * sizeof(float)));
glEnableVertexAttribArray(2);

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

Render Loop

```
glm::vec3 lightPos{ -2.0f, 4.0f, -1.0f };
...
~~glBindVertexArray(planeVao);~~
~~glDrawArrays(GL_TRIANGLES, 0, 6);~~

renderScene(blinnPhongShader, planeVAO, megaCubeVAO);
```

renderScene function

```
void renderScene(const Shader& shader, const unsigned int& planeVAO, const unsigned int& cubeVAO) {
   // Plane
   glm::mat4 model = glm::mat4(1.0f);
   glUniformMatrix4fv(glGetUniformLocation(shader.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
   glBindVertexArray(planeVAO);
   glDrawArrays(GL_TRIANGLES, 0, 6);

   // Cubes
   model = glm::mat4(1.0f);
   model = glm::translate(model, glm::vec3(0.0f, 1.5f, 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, 0.0f, 1.0));
   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(-1.0f, 0.0f, 2.0));
    model = glm::rotate(model, glm::radians(60.0f), glm::normalize(glm::vec3(1.0, 0.0, 1.0)));
    model = glm::scale(model, glm::vec3(0.25));
    glUniformMatrix4fv(glGetUniformLocation(shader.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
    glBindVertexArray(cubeVAO);
    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

Depth Map FBO Setup

Thursday, May 26, 2022 9:02 PM

main()

```
const unsigned int SHADOW_WIDTH = 1024, SHADOW_HEIGHT = 1024; // Resolution of the depth map
unsigned int depthMapFBO, depthMap;

// Creates a 2D texture to use as the framebuffer's texture
glGenTextures(1, &depthMap);
glBindTexture(GL_TEXTURE_2D, depthMap);
// Only care about depth values, so we use the GL_DEPTH_COMPONENT format
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glBindTexture(GL_TEXTURE_2D, 0);

// Creates the framebuffer that stores the depth map texture
glGenFramebuffers(1, &depthMapFBO);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMap, 0);
// A framebuffer object is not complete without a color buffer, so we must explicitly specify that we're not
// rendering any color data.
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);

if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE) {
   std::cout << "ERROR::FRAMEBUFFER:: Framebuffer is not complete!" << std::endl;
}

glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Example Rendering using Depth Map

Thursday, May 26, 2022 9:23 PM

```
// 1. First, render to the depth map
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 the depth map)
glViewport(0, 0, SCR_WIDTH, SCR_HEIGHT);

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
ConfigureShaderAndMatrices();
glBindTexture(GL_TEXTURE_2D, depthMap);
RenderScene();
```

While this leave out a bit of the finer details, what's important here are the calls to `glViewport`.

- Because shadow maps often have a different resolution compared to what we originally render the scene in (usually the window resolution), we need to change the viewport parameters to accommodate for the size of the shadow map.
- If we forget to update the viewport parameters, the resulting depth map will be either incomplete or too small.

Light Projection and View Matrices

Thursday, May 26, 2022 9:47 PM

```
// Light projection matrix
float nearPlane = 1.0f, farPlane = 7.5f;
glm::mat4 lightProjection = glm::ortho(-10.0f, 10.0f, -10.0f, 10.0f, nearPlane, farPlane);
// Light view matrix
glm::vec3 lightPos { -2.0f, 4.0f, -1.0f };
glm::mat4 lightView = glm::lookAt(lightPos,                         // Light position
                                  glm::vec3(0.0f, 0.0f, 0.0f),      // Where the light is pointing
                                  glm::vec3(0.0f, 1.0f, 0.0f));     // World up
// Light transformation matrix
glm::mat4 lightSpaceMatrix = lightProjection * lightView;
```

Because a projection matrix indirectly determines the range of what is visible (e.g. what is not clipped), you want to make s ure the size of the projection frustum correctly contains the objects you want to be in the depth map. When objects or fragments are not in the depth map, they will not produce shadow s.

Vertex and Fragment Shaders for Depth Map

Thursday, May 26, 2022 10:09 PM

simple_depth.vert

```
#version 330 core

layout (location = 0) in vec3 aPos;

uniform mat4 lightSpaceMatrix;
uniform mat4 model;

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

simple_depth.frag

```
#version 330 core

void main() {
   // We leave the fragment shader empty since the depth map framebuffer has no color buffer and the read and draw
   // buffers are disabled. The depth buffer is updated at the end of this shader's run, regardless.

   // gl_FragDepth = gl_FragCoord.z; // We could include this, but this happens behind the scenes anyways
}
```

main()

```
Shader simpleDepthShader{ "Shaders/simple_depth.vert", "Shaders/simple_depth.frag" };
```

Example Depth Map Rendering

Thursday, May 26, 2022 10:12 PM

```
simpleDepthShader.use();
glUniformMatrix4fv(glGetUniformLocation(simpleDepthShader.id, "lightSpaceMatrix"), 1, GL_FALSE, glm::value_ptr(lightSpaceMatrix));

glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);

glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
RenderScene(simpleDepthShader);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Here, RenderScene takes a shader program, then calls all relevant drawing functions and sets the corresponding model matrices where necessary.

Example Orthographic Projection Depth Map Shader

Thursday, May 26, 2022 10:20 PM

depth_map.vert

```
#version 330 core

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

out vec2 texCoords;

void main() {
   texCoords = aTexCoords;

   gl_Position = vec4(aPos, 0.0, 1.0);
}
```

The vertex shader is nothing new.

depth_map.frag

```
#version 330 core
out vec4 FragColor;

in vec2 texCoords;

uniform sampler2D depthMap;

void main() {
    float depthValue = texture(depthMap, texCoords).r; // Depth maps are grayscale, so the color you choose for the depth is arbitrary
    FragColor = vec4(vec3(depthValue), 1.0);
}
```

main()

```
Shader depthMapShader{ "Shaders/depth_map.vert", "Shaders/depth_map.frag" };
```

You should have the `screenQuadVAO` and `screenQuadVertices` from the framebuffer chapter.

Render Loop

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

// Generating the depth map
simpleDepthShader.use();
glUniformMatrix4fv(glGetUniformLocation(simpleDepthShader.id, "lightSpaceMatrix"), 1, GL_FALSE, glm::value_ptr(lightSpaceMatrix));

glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);

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

renderScene(simpleDepthShader, planeVAO, megaCubeVAO);

glBindFramebuffer(GL_FRAMEBUFFER, 0);

// Drawing the depth map to the screen
depthMapShader.use();

glViewport(0, 0, WIDTH, HEIGHT);

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, depthMap);

glBindVertexArray(screenQuadVAO);
glDisable(GL_DEPTH_TEST);
glDrawArrays(GL_TRIANGLES, 0, 6);
glEnable(GL_DEPTH_TEST);
```

Rendering Shadows

Friday, May 27, 2022 4:39 PM

NOTE: We're using the blinn-phong lighting implementation but adding shadows on top of that, so these shaders will look very similar.

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;
   vec4 fragPosLightSpace;
} vs_out;

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

void main() {

   vs_out.fragPos = vec3(model * vec4(aPos, 1.0));
   vs_out.normal = normalize(mat3(transpose(inverse(model)))* aNormal);
   vs_out.texCoords = aTexCoords;
   vs_out.fragPosLightSpace = lightSpaceMatrix * vec4(vs_out.fragPos, 1.0);

   gl_Position = projection * view * vec4(vs_out.fragPos, 1.0);
}
```

NOTE: Before, we were using point lights. Now, since our shadow map is being generated based on the idea of using a singular directional light, the point light implementation has been swapped out for a directional light implementation. As with the point light implementation, the directional light implementation below was discussed in the light casters chapter, though you might notice a few differences here.

shadows.frag

```
// Replaced PointLight with DirLight as it more accurately represents what light we're using
struct DirLight {
   vec3 direction;

   vec3 ambient;
   vec3 diffuse;
   vec3 specular;
};
uniform DirLight dirLight;

in VS_OUT {
   vec3 fragPos;
   vec3 normal;
   vec2 texCoords;
   vec4 fragPosLightSpace;
} fs_in;

out vec4 FragColor;

uniform vec3 viewPos;
uniform vec3 lightPos;
uniform sampler2D shadowMap;

vec3 CalcDirLight(DirLight light, vec3 viewDir);

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

   vec3 color = CalcDirLight(dirLight, viewDir);

   FragColor = vec4(color, 1.0);

   // Gamma correction
   float gamma = 2.2;
   FragColor.rgb = pow(FragColor.rgb, vec3(1.0 / gamma));
}
```

```
// This function will handle most of the shadow calculations
float ShadowCalculation() {
   // This step is useless for orthographic projection but VERY important for perspective projection
   // Converts position to NDC using perspective division
   vec3 projCoords = fs_in.fragPosLightSpace.xyz / fs_in.fragPosLightSpace.w;

   // Transforms projCoords to a [0,1] range since the depths in the depth map are a [0,1] range.
   projCoords = projCoords * 0.5 + 0.5;

   // Gets the closest depth value from the light's perspective
   float closestDepth = texture(shadowMap, projCoords.xy).r;
   // Gets the depth of the current fragment from the light's perspective
   float currentDepth = projCoords.z;
   // Checks whether current fragment position is in shadow
   float shadow = currentDepth > closestDepth ? 1.0 : 0.0; // 1.0 = in shadow, 0.0 = in light

   return shadow;
}

// This function generates the fragment's color based on a directional light (including shadows)
vec3 CalcDirLight(DirLight light, vec3 viewDir) {
   // Ambient Lighting
   vec3 ambient = light.ambient;

   // Diffuse Lighting
   vec3 lightDir = normalize(lightPos - 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();

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

Render Loop

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

// 1. Generates the depth map
simpleDepthShader.use();
glUniformMatrix4fv(glGetUniformLocation(simpleDepthShader.id, "lightSpaceMatrix"), 1, GL_FALSE, glm::value_ptr(lightSpaceMatrix));

glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);

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

renderScene(simpleDepthShader, planeVAO, megaCubeVAO);

glBindFramebuffer(GL_FRAMEBUFFER, 0);

// 2. Draws the scene with shadows using the depth map
glViewport(0, 0, WIDTH, HEIGHT);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glActiveTexture(GL_TEXTURE2);
glBindTexture(GL_TEXTURE_2D, depthMap);

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

shadowShader.use();

planksDiffuseTexture.bind();
fullSpecularTexture.bind();
glActiveTexture(GL_TEXTURE2);
glBindTexture(GL_TEXTURE_2D, depthMap);

// Sets the vertex shader uniforms
glUniformMatrix4fv(glGetUniformLocation(shadowShader.id, "view"), 1, GL_FALSE, glm::value_ptr(view));
```

```
glUniformMatrix4fv(glGetUniformLocation(shadowShader.id, "projection"), 1, GL_FALSE, glm::value_ptr(projection));
glUniformMatrix4fv(glGetUniformLocation(shadowShader.id, "lightSpaceMatrix"), 1, GL_FALSE, glm::value_ptr(lightSpaceMatrix));
// Sets the fragment shader uniforms
// Material uniforms
glUniform1i(glGetUniformLocation(shadowShader.id, "material.diffuse"), planksDiffuseTexture.getTexUnit());
glUniform1i(glGetUniformLocation(shadowShader.id, "material.specular"), fullSpecularTexture.getTexUnit());
glUniform1f(glGetUniformLocation(shadowShader.id, "material.shininess"), 32.0f);
// Other fragment shader uniforms
glUniform3fv(glGetUniformLocation(shadowShader.id, "viewPos"), 1, glm::value_ptr(camera.position));
glUniform3fv(glGetUniformLocation(shadowShader.id, "lightPos"), 1, glm::value_ptr(lightPos));
glUniform1i(glGetUniformLocation(shadowShader.id, "shadowMap"), 2);

// Sets the directional light uniforms
glm::vec3 lightColor{ 1.0f, 1.0f, 1.0f };
glUniform3fv(glGetUniformLocation(shadowShader.id, "dirLight.direction"), 1,
    glm::value_ptr(-lightPos));
glUniform3fv(glGetUniformLocation(shadowShader.id, "dirLight.ambient"), 1,
    glm::value_ptr(glm::vec3(0.05f) * lightColor));
glUniform3fv(glGetUniformLocation(shadowShader.id, "dirLight.diffuse"), 1,
    glm::value_ptr(glm::vec3(0.8f) * lightColor));
glUniform3fv(glGetUniformLocation(shadowShader.id, "dirLight.specular"), 1,
    glm::value_ptr(glm::vec3(1.0f) * lightColor));

renderScene(shadowShader, planeVAO, megaCubeVAO);
```

renderScene function

```
// Renders a plane and 3 cubes in the current scene
void renderScene(const Shader& shader, const unsigned int& planeVao, const unsigned int& cubeVAO) {
   // Plane
   glm::mat4 model = glm::mat4(1.0f);
   glUniformMatrix4fv(glGetUniformLocation(shader.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
   glBindVertexArray(planeVao);
   glDrawArrays(GL_TRIANGLES, 0, 6);

   // Cubes
   model = glm::mat4(1.0f);
   model = glm::translate(model, glm::vec3(0.0f, 1.5f, 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, 0.0f, 1.0));
   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(-1.0f, 0.0f, 2.0));
   model = glm::rotate(model, glm::radians(60.0f), glm::normalize(glm::vec3(1.0, 0.0, 1.0)));
   model = glm::scale(model, glm::vec3(0.25));
   glUniformMatrix4fv(glGetUniformLocation(shader.id, "model"), 1, GL_FALSE, glm::value_ptr(model));
   glBindVertexArray(cubeVAO);
   glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

Shadow Bias

Friday, May 27, 2022 7:09 PM

```
// Shadow bias used to prevent shadow acne
float bias = max(0.05 * (1.0 - dot(fs_in.normal, lightDir)), 0.005);

float shadow = currentDepth - bias > closestDepth ? 1.0 : 0.0;
```

Reducing Peter Panning

Friday, May 27, 2022 7:26 PM

Render Loop

```
// Front face culling to reduce peter panning
glEnable(GL_CULL_FACE);
glCullFace(GL_FRONT);
renderScene(simpleDepthShader, planeVAO, megaCubeVAO);
glCullFace(GL_BACK);
glDisable(GL_CULL_FACE);
```

Fixing Over Sampling

Friday, May 27, 2022 9:04 PM

main()

```
// Clamping to a border ensures that coordinates outside the depth map's range have a depth of 1.0, fixing duplicated shadows
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
float borderColor[] = { 1.0f, 1.0f, 1.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
```

```
float ShadowCalculation(DirLight light, vec3 lightDir) {
   ...
   float shadow;
   if (projCoords.z > 1.0) {
      // Removes shadow from fragments outside view light frustum's range (z larger than 1.0)
      shadow = 0.0;
   }
   else {
      shadow = currentDepth - bias > closestDepth ? 1.0 : 0.0;
   }

   return shadow;
}
```

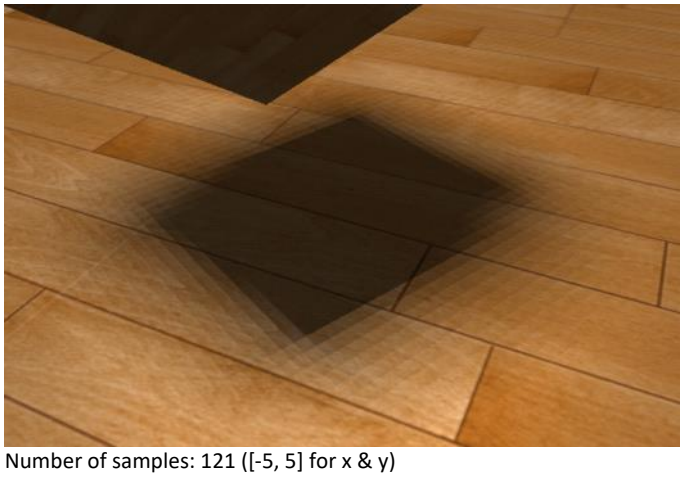
Percentage-Closer Filtering

Friday, May 27, 2022 9:59 PM

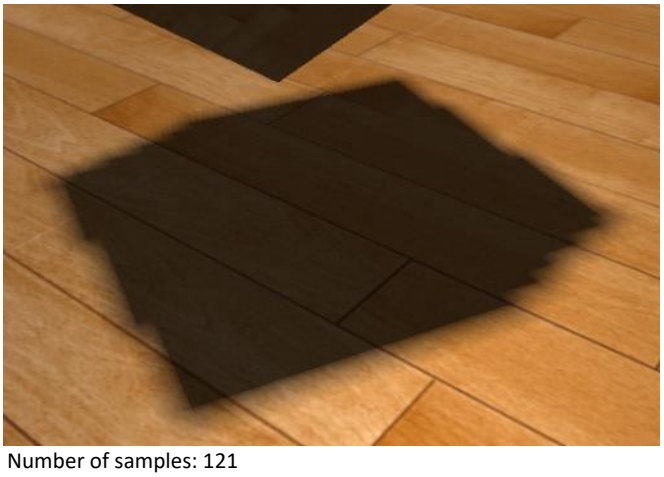
```
if (projCoords.z > 1.0) {
   ...
}
else {
   vec2 texelSize = 1.0 / textureSize(shadowMap, 0); // Size of a single texel, used to offset the texture coordinates
   // Samples 9 values around the projected coordinate's x and y value
   for (int x = -1; x <= 1; ++x) {
      for (int y = -1; y <= 1; ++y) {
         // Samples the depth value in shadowMap with offset vec2(x, y) * texelSize).r from the current projected coords
         float pcfDepth = texture(shadowMap, projCoords.xy + vec2(x, y) * texelSize).r;
         // Tests for shadow occlusion and possibly increments the shadow value
         shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
      }
   }
   shadow /= 9.0; // Averages the shadow value by the number of samples taken
}

return shadow;
```

If you want to configure the softness of the shadows:

- Increasing the number of samples provides a wider border for softening, which can make the shadow appear smaller.
- Decreasing the `texelSize` shrinks the border for softening, which can make the soft edges appear less aggressive.



Number of samples: 121 ([-5, 5] for x & y)
Size of texels sampled: 1.0 / textureSize(shadowMap, 0) = 0.0009765625



Number of samples: 121
Size of texels sampled: 0.2 / textureSize(shadowMap, 0) = 0.0001953125

Example Perspective Projection Depth Map Shader

Friday, May 27, 2022 10:35 PM

depth_map.frag

```
#version 330 core
out vec4 FragColor;

in vec2 texCoords;

uniform sampler2D depthMap;
uniform float nearPlane;
uniform float farPlane;

float LinearizeDepth(float depth) {
    float z = depth * 2.0 - 1.0; // Back to NDC
    return (2.0 * nearPlane * farPlane) / (farPlane + nearPlane - z * (farPlane - nearPlane));
}

void main() {
    float depthValue = texture(depthMap, texCoords).r;
    FragColor = vec4(vec3(LinearizeDepth(depthValue) / farPlane), 1.0); // Perspective
    // FragColor = vec4(vec3(depthValue), 1.0); // Orthographic
}
```

NOTE: This is the same shader as was shown here, but it now supports displaying perspective projection depth maps to a quad.