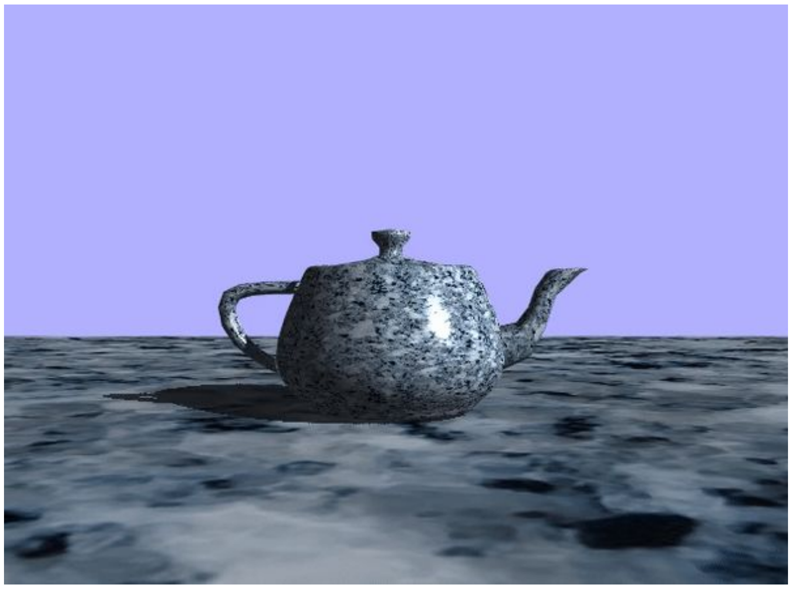
Shadow Mapping Guide

Overview

In this assignment, you will implement basic shadow mapping with a single directional light source. This assignment will incorporate many of the elements we've encountered through the course and focus on both the supporting code from the application as well as the shaders. You will need to use either Chrome or Firefox to complete this assignment. The starting code will be made available after the previous assignment is fully turned in or upon request via email.



It is assumed that you have covered the lecture on shadows and have a conceptual understanding of the process.

You may find the relevant section of the WebGL Programming Guide useful for implementation details: ch10 (pages 392-414). Alternatively, if you'd like a slower and more passive approach to soak it all in, you may find the series by *Indigo Code* useful: part 1, part 2, part 3.

Structure & Execution Order

Directory Structure

/Root *(contains main files .html and app.js)*

   /data *(contains images and models)*

   /math *(contains our math code)*

   /misc *(contains code to assist creating, loading, and rendering)*

   /shaders *(contains the GPU code for rendering)*

We will be implementing and using the following shaders.

root/shaders/depth-write.vs.glsl

root/shaders/depth-write.fs.glsl

root/shaders/phong..fs.glsl

root/shaders/phong.fs.glsl

Careful!

You will not be able to run your code simply by launching the index.html file. This is because the code now needs to access separate files like images and shaders from your hard drive. This is a security violation and all browsers will block it. The recommended way to get around this is to serve your files from a local web server and then start your application by going to your localhost url.

I like to use http-server from node.js. Instructions for how to set this up can be found here. If you prefer Python instead of using node, see here.

To start, if you run the code you will see:

This should be similar to the last assignment where we implemented a directional light.

Camera Setup

Todo #1 Aim the Depth Camera

For a surface point to be directly lit, it must have a clear line of sight to the light. To accomplish this we will render the scene from the point of view of the light, only we won't be storing colors in the frame buffer but rather distances in an offscreen texture. But first we need to place this depth camera somewhere that makes sense.

Unlike the idea of directional light which is infinite, rendering is restricted to a viewing volume so we want to position this camera such that is in the same direction as our directional light AND such that the scene lies in between the near and far planes.

At the top of app.js you can see that we have new separate camera for this purpose: ***lightCamera***.

```
var lightCamera = new Camera();
```

In *app.js:**updateAndRender()***, we will aim this camera before rendering. To do this, a new function has been added to matrix4.js: ***setLookAt***.

```
lightCamera.cameraWorldMatrix.setLookAt(eyePos, targetPos, worldUp);
```

This makes it easy to directly position and orient an object that faces another object. Set the correct arguments to setLookAt such that the light camera renders from **[5, 3, 0]** to the origin **[0, 0, 0]**. You can use **[0, 1, 0]** as the "up" value.

Todo #2 Create and Set Orthographic Projection Matrix

Now that we are aiming the depth camera in the scene, we need to focus on how it is projected. Previously, we always used perspective projection for this. In this case, perspective is not as useful as the light rays we are modeling are moving in parallel. Perspective projection will not preserve this. Luckily there is a projection that will: orthographic.

In your ***Matrix4***, implement a new function called ***setOrthographic***. This function should take the following parameters: left, right, top, bottom, near, far. These parameters represent the distance of each of the orthographic planes from the origin. See the lecture slides (7.1) for more detail.

If you are unsure as to whether you have coded this part correctly, please feel free to share your result for this todo and discuss in the class forum. **DO NOT** share any of your other code from other todos.

In app.js:***updateAndRender***, find the todo that calls your ***setOrthographic*** function use the following arguments:

Left and bottom = -10

Right and top = 10

Near = 1.0

Far = 20

Todo #3 Verify Results Using the Specter Debugger

Because we are now focused on rendering that will not appear directly on the screen, debugging becomes more difficult. Luckily instead of writing a bunch of special code to visualize what is happening, we can instead leverage a tool that will automatically do that for us: Specter.

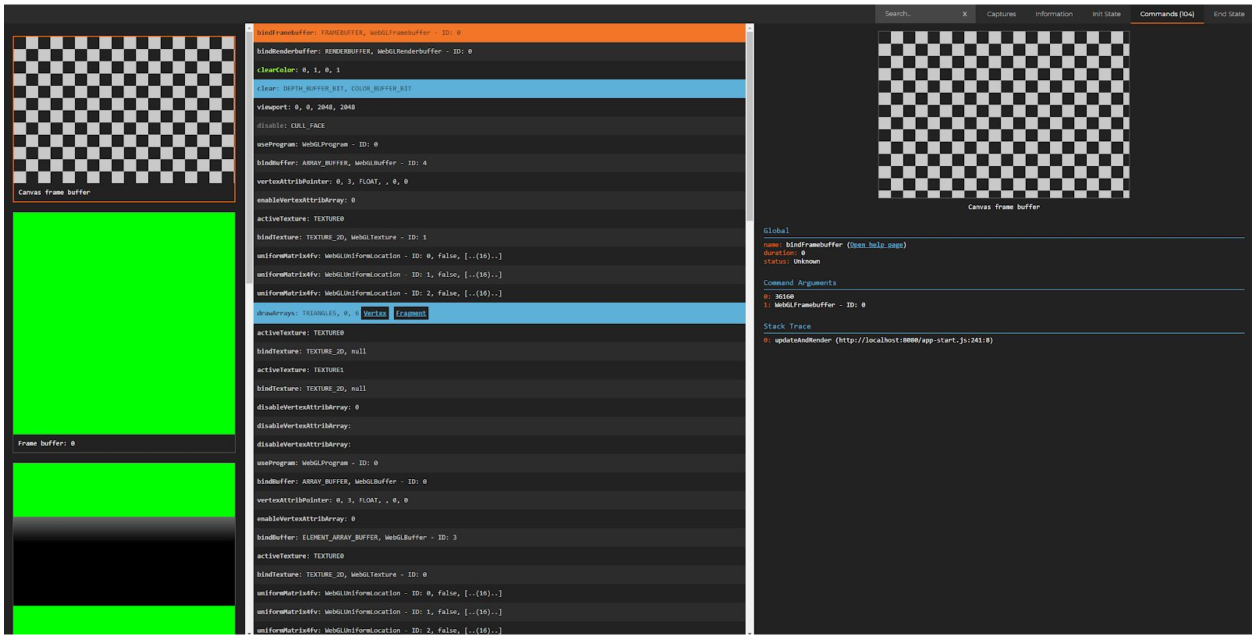
Install the Specter WebGL debugger. You will need to use Chrome or Firefox for this.

Install for Chrome

Install for Firefox

An introduction to using Specter can be found here.

Try capturing a frame in Specter and you should see something like this:



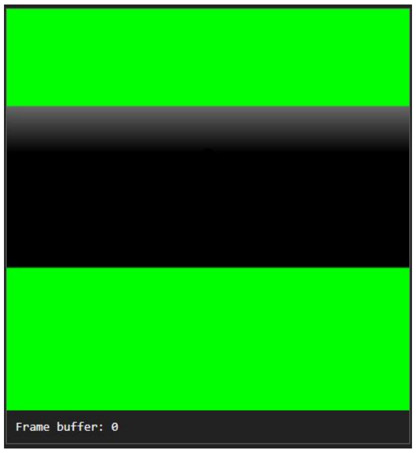
```
bindFramebuffer: FRAMEBUFFER, WebGLFramebuffer - ID: 0
bindRenderbuffer: RENDERBUFFER, WebGLRenderbuffer - ID: 0
clearColor: 0, 1, 0, 1
clear: DEPTH_BUFFER_BIT, COLOR_BUFFER_BIT
viewport: 0, 0, 2048, 2048
disable: CULL_FACE
useProgram: WebGLProgram - ID: 0
bindBuffer: ARRAY_BUFFER, WebGLBuffer - ID: 4
vertexAttribPointer: 0, 3, FLOAT, , 0, 0
enableVertexAttribArray: 0
activeTexture: TEXTURE0
bindTexture: TEXTURE_2D, WebGLTexture - ID: 1
uniformMatrix4fv: WebGLUniformLocation - ID: 0, false, [..(16)..]
uniformMatrix4fv: WebGLUniformLocation - ID: 1, false, [..(16)..]
uniformMatrix4fv: WebGLUniformLocation - ID: 2, false, [..(16)..]
drawArrays: TRIANGLES, 0, 6 Vertex Fragment
activeTexture: TEXTURE0
bindTexture: TEXTURE_2D, null
activeTexture: TEXTURE1
bindTexture: TEXTURE_2D, null
disableVertexAttribArray: 0
disableVertexAttribArray:
disableVertexAttribArray:
useProgram: WebGLProgram - ID: 0
bindBuffer: ARRAY_BUFFER, WebGLBuffer - ID: 0
vertexAttribPointer: 0, 3, FLOAT, , 0, 0
enableVertexAttribArray: 0
bindBuffer: ELEMENT_ARRAY_BUFFER, WebGLBuffer - ID: 3
activeTexture: TEXTURE0
bindTexture: TEXTURE_2D, WebGLTexture - ID: 0
uniformMatrix4fv: WebGLUniformLocation - ID: 0, false, [..(16)..]
uniformMatrix4fv: WebGLUniformLocation - ID: 1, false, [..(16)..]
uniformMatrix4fv: WebGLUniformLocation - ID: 2, false, [..(16)..]
```

```
Global
name: bindFramebuffer (Open help page)
duration: 0
status: Unknown

Command Arguments
0: 36160
1: WebGLFramebuffer - ID: 0

Stack Trace
0: updateAndRender (http://localhost:8080/app-start.js:241:8)
```

The important part of this to verify our current results is this:



This is a representation of the offscreen texture that we rendered into. Our depth-write shaders are currently only writing vertex position's z value to this texture which is not ultimately what we want. We will remedy this in the next "todo".

Depth & Texturing

Todo #4 Remap Depth into the 0 to 1 range

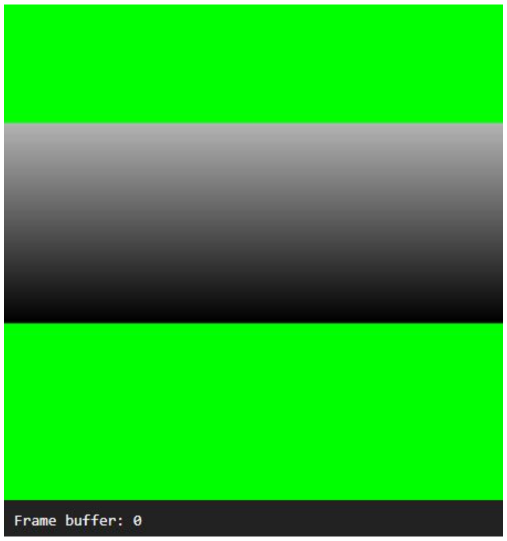
Our texture values are limited to this range so anything we output that is less than 0 or greater than 1 will get clamped which is not useful. If something is farther or closer than something else in depth, we want to be able to differentiate them. Inside of **depth-write.vs.glsl** (the vertex shader), notice that we are passing the floating point varying vDepth to be interpolated for the fragment shader and that we are just setting it to the z value computed by the projection. Recall that anything inside of the view volume will be remapped to the -1 to 1 range. Change the calculation for ***vDepth*** such that it leaves the vertex shader in the 0 to 1 range.

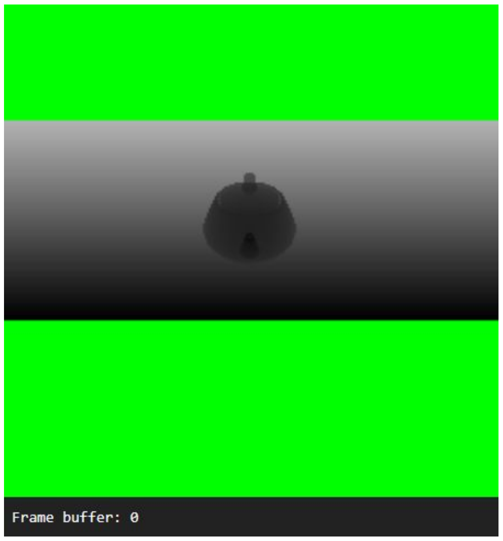
```
void main(void) {
    gl_Position = uProjectionMatrix * uViewMatrix * uWorldMatrix * vec4(aVertexPosition, 1.0);
    // todo #4 Rescale depth (z value) from normalized device coordinates ([-1, 1]) to [0, 1]
    //   Note: with orthographic projection, clip space and NDC are the same
    vDepth = ?; // gl_Position.z is in [-1, 1], remap it to [0, 1]
}
```

[Extra FYI]

With perspective projection, this remapped space was clip space. In order to get to the final -1 to 1 range the hardware would divide z by the homogeneous coordinate w after leaving the vertex shader. If we were using perspective projection and we wanted that value in the shader, we would have to manually do the division here.

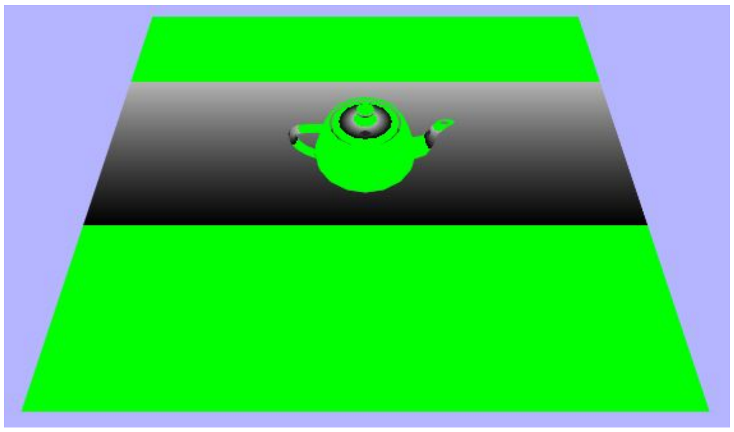
If you capture another frame from Specter, you should now see this:


Frame buffer: 0


Frame buffer: 0

Todo #5 Visualize the Depth Texture

As a sanity check, in **phong.fs.glsl**, sample the color from ***uShadowTexture*** using your normal ***vTexCoords***. These texture coordinates are meant for draping regular color textures on the surface, not some special texture we created by rendering from somewhere in the scene. However, we just want to verify that this piece is in place before moving on. If you set the output color to be equal to the color you sampled, you should see:

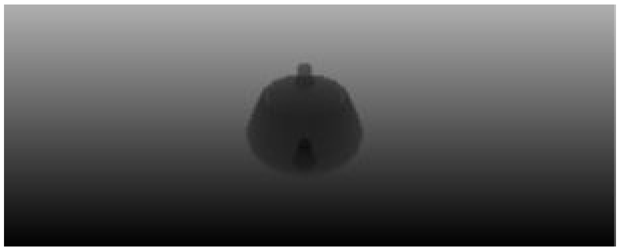


Todo #6 Projective Texturing

The last todo showed little more than the fact that we have some grayscale colors in our depth texture. It would be more useful if we showed those depth colors on our surface in a more meaningful way. The texture coordinates that the surfaces currently use are completely unrelated to our depth texture.

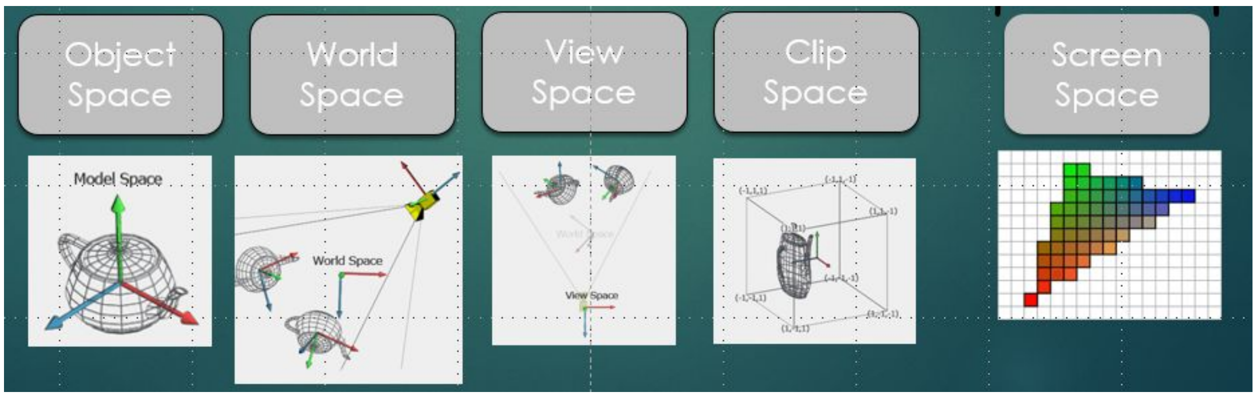
Ultimately, we want to be able to be able to get the distance to the light (along a direction) from any point on a surface. We already have information about depth from rendering the scene from the light direction (light camera). However, we currently have no way to access that information. For example, if we were rendering the spout of the teapot, what is its depth from the light camera? We have a texture with that information but we don't know which texture coordinate will get us the one that corresponds to the spout. Enter projective texture coordinates…

Remember how we set up the view matrix (***setLookAt***) and the orthographic projection for the light? That directly relates to the depth texture we generated.

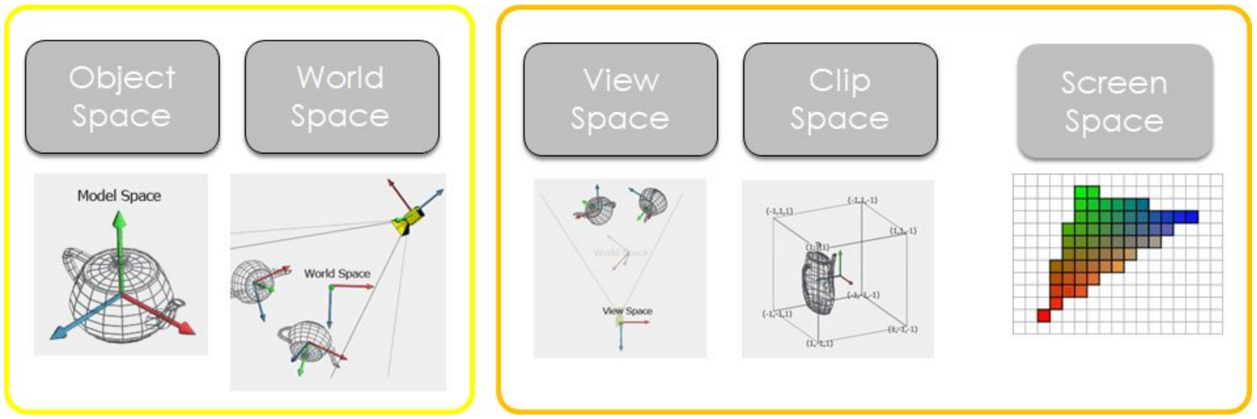


But we aren't rendering the real color scene from that perspective. We're looking at it from here:

The difference in each case are the final transformation stages.



Up until the view space transformation, they are the same.



In phong.vs.glsl, we output the world space position as ***vWorldPosition***. We can then take this and transform it into the light's clip space. This gives us the coordinates for the current position in the -1 to 1 range from the view of the light. This is pretty close to the range used for texture coordinates. It just needs to be remapped to 0 to 1. Before we do this, note the following:

In **app.js:updateAndRender**, we combine the light's view and projection into ***lightVPMatrix***.

```
var lightVPMatrix = shadowProjectionMatrix.clone().multiplyRightSide(lightCamera.getViewMatrix());
```

We then send that matrix to our shader before rendering.

```
gl.uniformMatrix4fv(uniforms.lightVPMatrixUniform, false, lightVPMatrix.transpose().elements);
```

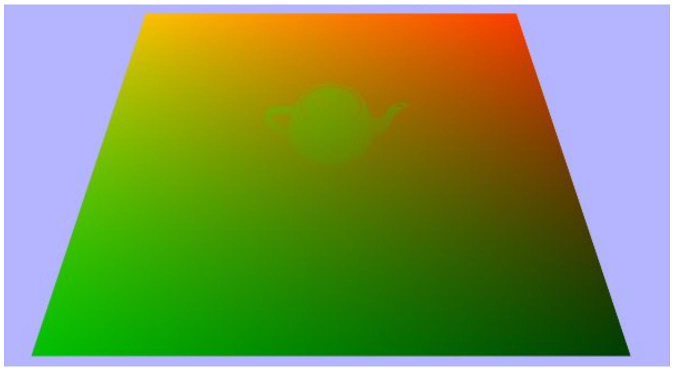
Now, in **phong.fs.glsl**, do the following:

1) Transform ***vWorldPosition*** into the light's clip space (same as NDC, -1 to 1) and store your result in a vec4 variable called ***lightSpaceNDC***.

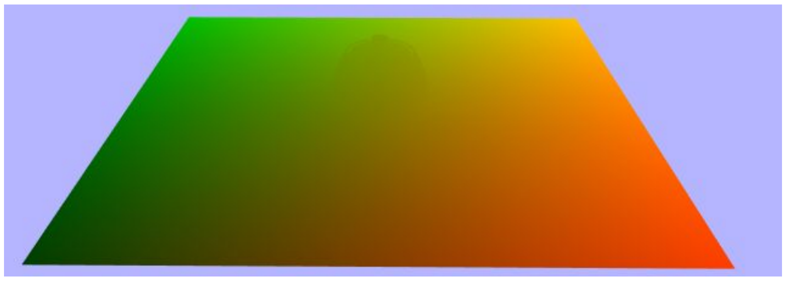
2) Next create a new vec2 variable called ***lightSpaceUV*** which has the same x and y components from ***lightSpaceNDC***, only remapped to 0 to 1.

3) Set gl_FragColor = vec4(***lightSpaceUV.x***, ***lightSpaceUV.y***, 0.0, 1.0)

You should see something like this:



If you rotate the main camera to line up in a place similar to where the light camera is you'll see the following:

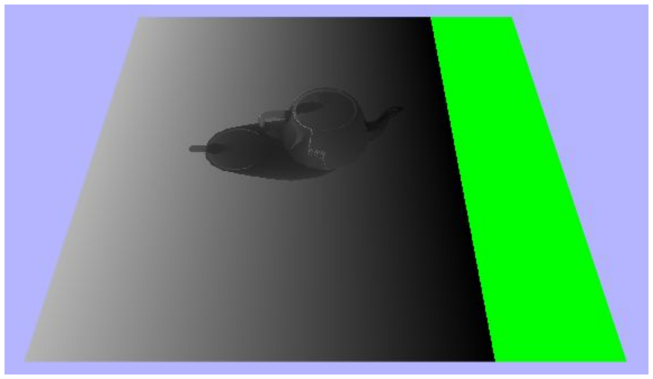


Notice that red goes to the right and green goes out (the up direction in light space is tilted).

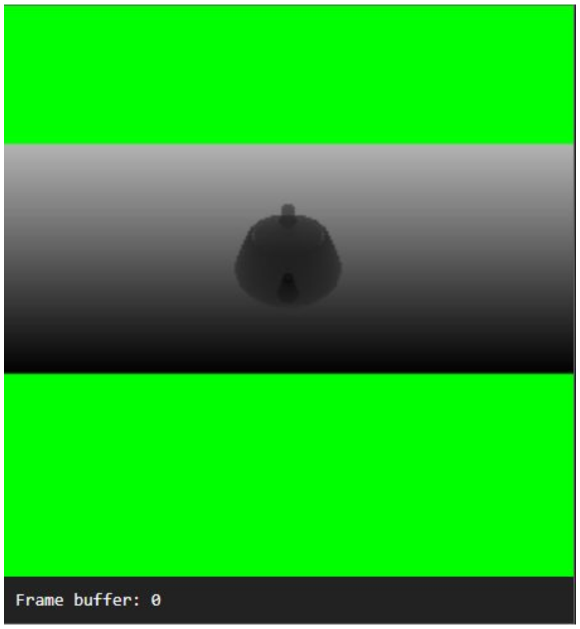
Shadows

Todo #7 Sample the Shadow texture

Now that you have light-space uv coordinates, use them to sample the depth color from the depth texture. Output this color and you will get:

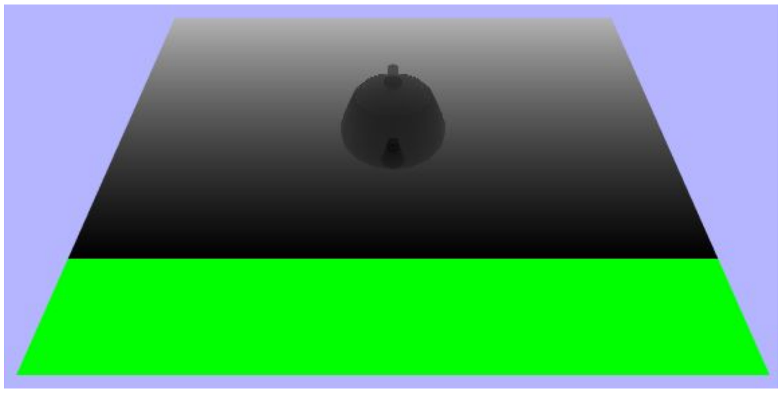


You may be surprised by this result, just remember that this is projected from the light camera's point of view which looked like this.



The green color is there because the light camera's near plane cut into it. It won't cause a problem but you could try adjusting your light camera's position and orthographic parameters if you're interested.

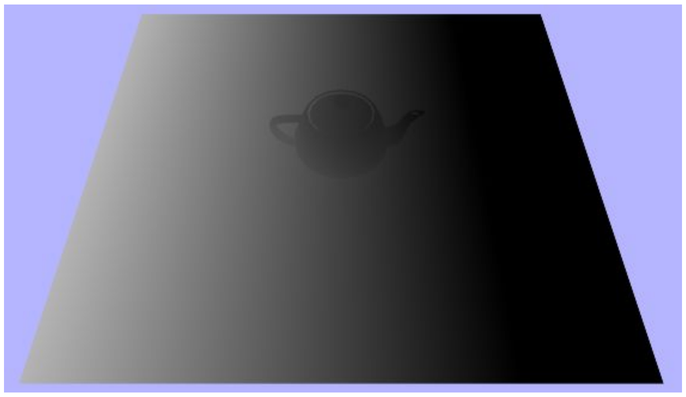
Again, if you rotate the main camera to roughly line up with the light camera you will see



This is what we're after, a measure of the distance from the camera's near plane to its closest object, not the one we're currently rendering.

Todo #8 Visualize Surface Depth in Light-Space

Remember how you calculated your projected uv coordinates? You did x and y but not z. Now Z gets to shine. Remap ***lightSpaceNDC.z*** into the 0 to 1 range and store it in a float variable called lightDepth. Then visualize this value: gl_FragColor = vec4(***lightDepth***, ***lightDepth***, ***lightDepth***, 1.0).



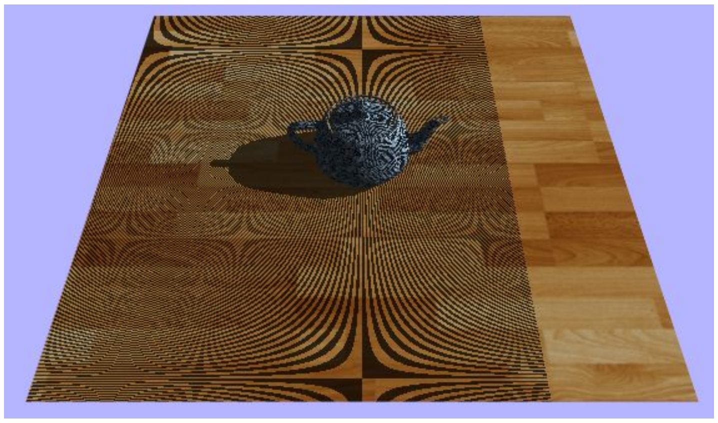
Todo #9 Shadows!

We now have depth values in light-space for both the current surface point and what the closest depth to the light is. Time to compare! Do something like this:

```
// Sample the depth of the closest position to the light (depth texture)
vec4 shadowColor = texture2D(?, ?);

if ((current surface position depth to light) > (closest depth to light))
    show only ambient light color
} else {
    show fully lit color
}
```

And you will get some glorious surface acne shadows.



Todo #10 Shadow Bias

You will need to add a tiny value to your "closest depth to the light value" to eliminate this. 0.004 worked well for me.

Try rotating the light with the keyboard to see the shadow change accordingly.

Bonus

Todo #1 Implement Percentage Closer Filtering (PCF) (15 pts)

This is a technique to help minimize aliasing artifacts. A quick example of this can be seen here. Use this or any other online reference for your implementation.