02561: Computer Graphics

Project report

Fall 2022

Gabriel Luo – s221400

December 18, 2022

Table of Contents

1.0 Introduction

The purpose of the project is to implement planar reflections using WebGL. The problem will be divided into 4 different steps as outlined in the "Project Initiator 1: Planar Reflector" document uploaded on DTU Learn [1]. The project is aimed to gain a better understanding of how reflections work in Computer Graphics, and understand that implementing reflections using rasterization, as is done with WebGL, is harder than when done using ray tracing, but is still possible. For simplicity, the project will only consider planar reflectors as the result can be achieved simply by drawing the reflection as a separate object and rendering.

2.0 Method

The problem to implement planar reflections is split into four parts as outlined in the project initiator document. Each of the steps will be explained in detail in the sections that follow.

2.1 Draw the reflected object

The purpose of this first part is to create the reflection of the teapot in the proper position, ensuring that the lighting is also reflected appropriately.

Reflecting the teapot involved drawing another teapot object and applying a specific transformation matrix to put the second teapot in the position where the reflection of the initial teapot would stand based on the reflection plane. The general idea for the transformation matrix needed to translate the reflection plane up to the origin, rotate the plane to match one of the major planes (such as the x-y plane), apply a -1 scale, and use the inverse of the rotation and translation previously used to get back to the mirror's original location.

The reflected object can be drawn using the following transformation matrix [2]:

$$R = \begin{pmatrix} 1 - 2V_x^2 & -2V_xV_y & -2V_xV_z & 2(P \cdot V)V_x \\ -2V_xV_y & 1 - 2V_y^2 & -2V_yV_z & 2(P \cdot V)V_y \\ -2V_xV_y & -2V_yV_z & 1 - 2V_z^2 & 2(P \cdot V)V_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (1)$$

In the transformation matrix are the variables V and P, which are a vector perpendicular to the mirror plane (ie. the normal), and a point on the mirror's surface. With the table being defined as the reflective surface, its coordinates, which are (-2, -1, -1), (-2, -1, -5), (2, -1, -1), and (2, -1, -5), can be used to determine P and V.

For V, this is just the x-z plane shifted in the -y direction by 1 unit, so the normal vector is (0, 1, 0). This vector is chosen instead of (0, -1, 0) as although they are both normal to the plane reflector, the reflection will appear on the opposite side of the reflector. In other words, since the reflection should appear on the -y side of the plane as the real teapot is on the positive side, the normal vector should be a scalar multiplication of (0, 1, 0). The unit vector is chosen for simplicity.

For consistency, when objects are on the side that will generate a reflection on the table or the reflective side, will be considered as the positive side of the planar reflector due to the mostly positive values of y which determine the position coordinates of objects on that side. The negative side will

consider objects that are on the non-reflective side of the planar reflector. Objects on this side should not be reflected.

For P, the point in the center of the plane was chosen arbitrarily, but any of the vertex points would have satisfied. Point P was found to be (0, -1, -3).

By multiplying R after the model matrix but after the view matrix, the teapot drawn should be shown in the reflected position. The ground is also not rendered for this step to see the reflection better.

2.2 Add the reflective ground and blend

Adding the ground should be as simple as uncommenting the part that was previously commented out to make the ground appear. To add the blending, the alpha parameter when setting up WebGL should be set to false to ensure that blending is independent of the colors in the browser window. Then, the ground must be drawn on top of the teapot reflection. Before drawing the ground, alpha blending should be enabled, and the most common blend function can be used as shown below [3].

$$\boldsymbol{d}' = s_a \boldsymbol{s} + (1 - s_a)\boldsymbol{d} \tag{2}$$

After the ground is drawn, alpha blending should be disabled to avoid errors with drawing in the next render call. In the table's fragment shader the alpha value for the table color should then be set to an arbitrary value determined by trial and error to see which value gives the best visual representation of a reflective marble surface.

2.3 Implement stencil buffer for clipping

When considering the implementation of Section 2.2, there isn't any evident issue until the movement of the teapot is enabled. When the real teapot is high enough that the reflection begins to move past the bounds of the table, the reflection will still be shown, which ruins the illusion.

To clip the geometry of the reflection past the table, the depth buffer cannot be used as was done for the projection shadows because it is needed for the reflection object to be rendered properly. Hence, a stencil buffer will be used instead.

The basic methodology is that the table, the reflector, will be drawn in the stencil buffer, and the stencil buffer will assign a "1" to the points where the table is drawn. This notes the points where the table would be drawn. The depth and color buffers are disabled for this part. Then, when the teapot reflection is drawn, if the stencil buffer does not contain a "1" at a specific pixel, it means that the table surface does not cover that. So the pixel of the reflected teapot is rejected and not drawn. The table is then drawn afterwards with the depth and color buffers enabled as usual [4].

If done correctly, the reflection of the teapot should only be visible in the area where the table is rendered.

2.4 Oblique near-plane clipping

The implementation in Section 2.3 looks complete as the reflection of the teapot matches throughout its motion from the surface of the table and up. However, consider the edge case where the teapot was to go underneath the table, ie. on the other side of the planar reflector. Based on the reflection matrix from Section 2.1, the reflection of the teapot would show up on the positive side of the planar reflector, or the side where the real teapot was. The reflection would still be clipped to the dimensions of the table due to the stencil buffer, but there are still positions where the parts of the teapot under the table

are still reflected. In fact, without the stencil buffer, the two teapots would essentially just have switched positions and rotated 180°.

To avoid this problem, another form of clipping needs to be introduced so that the reflection of object geometry on the negative side of the planar reflector (ie. the non-reflective side) isn't shown. To do so, a technique called oblique near-plane clipping will be used.

The implementation of oblique near-plane clipping is described in [5]. The procedure involves taking the projection matrix and modifying the near plane parameter so that parts drawn past a certain plane (in this case the planar reflector) will not be shown. This needs to be done in a manner so that the view angle, far plane, and aspect ratio are all kept the same.

The modified projection matrix can be constructed as follows. First, vector $\boldsymbol{C}$ will be used to represent the plane used to clip the geometry, in this case, the table. A plane can be mathematically defined using the following equation:

$$ax + by + cz + d = 0 \tag{3}$$

The values for a, b, c, and d are the values that will be put into $\boldsymbol{C}$. The values for a, b, and c are the x, y, and z components of a normal vector to the plane, while d can be defined using the following equation [5]:

$$d = -\boldsymbol{N} \cdot \boldsymbol{P} \tag{4}$$

Where **N** is the normal vector and **P** is a point on the plane. It is assumed that the same normal vector and point used in section 2.1 when finding the reflection matrix can also be used here. However, the plane must be represented in camera space, so transforming the vector simply involves multiplying it by the inverse transpose of the view matrix, which will give the final value for C [5]. In other words,

$$\boldsymbol{C} = (\boldsymbol{V}^{-1})^T \langle N_x, N_y, N_z, d \rangle \tag{5}$$

Where **V** is the view matrix.

Next, $\boldsymbol{C}'$ is defined by transforming C into clip space using the projection matrix M as shown below [5].

$$\boldsymbol{C}' = (\boldsymbol{M}^{-1})^T \boldsymbol{C} \tag{6}$$

From $\boldsymbol{C}'$, the value of $\boldsymbol{Q}$ can be defined as the point which is the corner of the view frustum lying on the other side of $\boldsymbol{C}'$.

$$\boldsymbol{Q}' = \langle sign(C'_x), sign(C'_y), 1, 1 \rangle \tag{7}$$

With these variables defined, the modified projection can now be found with $\boldsymbol{M}$ being the original projection matrix, $\boldsymbol{M}'$ being the modified projection matrix, $\boldsymbol{C}$ being the clipping plane in camera space, and $\boldsymbol{Q} = \boldsymbol{M}^{-1}\boldsymbol{Q}'$, defined in Equation (7) above.

$$\boldsymbol{M}' = \begin{bmatrix} \boldsymbol{M_1} \\ \boldsymbol{M_2} \\ \dfrac{-2Q_z}{\boldsymbol{C} \cdot \boldsymbol{Q}} \boldsymbol{C} + \langle 0,0,1,0 \rangle \\ \boldsymbol{M_4} \end{bmatrix} \tag{8}$$

By applying this modified projection matrix when drawing the reflection of the teapot, it should now be clipped when the reflection is trying to show the positive side of the reflector.

Another issue that comes up is that when the real teapot moves under the table, it is still visible since although it is behind the table object, the table was made to be slightly transparent from Section 2.2 and the real teapot now looks like another reflection. To fix this issue, the parts of the real teapot under the table should not be drawn. This can be accomplished by simply using the depth buffer and drawing the real teapot after the table instead of drawing it first as is done in Section 2.3.

2.5 Percentage-closer filtering

To help improve the quality of the shadows produced using shadow mapping, a technique known as shadow percentage-closer filtering can be used to reduce the aliasing (staircase) effect that appears at the edge of the shadows. The methodology is based on a function from the Lecture 10 slides and a source from NVIDIA [6] [7].

The method involves looking at the pixels around the pixel in question on the shadow map and taking the average of the depth values obtained. This results in less drastic changes in the visibility between pixels which are closer together, thus replacing the staircase effect with a blurred effect. The values around the pixel in question are accumulated in a loop and each one is referenced to the shadow map to determine the appropriate value.

Implementing this also requires the resolution of the shadow map as a uniform variable to the fragment shader. Specifically, the size of each step in the shadow map is needed, which is based on the frame buffer object's height and width as shown below [7].

$$texmapscale = \left\langle \frac{1}{width}, \frac{1}{height} \right\rangle \tag{9}$$

3.0 Implementation

The code from Worksheet 9 Part 2 will be used as a base, which contains a jumping teapot and rotating light source. Effects rendered in the scene are shadow mapping and lighting effects on the teapot and ground.

3.1 Part 1

The first step was to change the view and perspective matrices from Worksheet 9 part 2 to better see the reflection.

```
var P = perspective(65, 1, 0.1, 10);
var eyePos = vec3(0.0, 0.0, 1.0);
var eyeUp = vec3(0.0, 1.0, 0.0);
var eyeAt = vec3(0.0, 0.0, -3.0);
var V = lookAt(eyePos, eyeAt, eyeUp);
```

Next, the algorithm indicated in Section 2.1 needs to be implemented. First, the draw call for the table is commented out so that the table and shadows on the table are no longer shown. Two temporary vector3's, `V_r` and `P_r`, are created which represent V and P in Equation (1). The rest of matrix R is calculated accordingly using the following code, where R is first initiated as a 4x4 identity matrix.

```
var R = mat4();
```

```
var V_r = vec3(0.0, 1.0, 0.0);
var P_r = vec3(0.0, -1.0, -3.0);
R[0][0] = 1.0-2.0*Math.pow(V_r[0],2.0);
R[0][1] = -2.0*V_r[0]*V_r[1];
R[0][2] = -2.0*V_r[0]*V_r[2];
R[0][3] = 2.0*(dot(P_r, V_r))*V_r[0];
R[1][0] = -2.0*V_r[0]*V_r[1];
R[1][1] = 1.0-2.0*Math.pow(V_r[1],2.0);
R[1][2] = -2.0*V_r[1]*V_r[2];
R[1][3] = 2.0*(dot(P_r, V_r))*V_r[1];
R[2][0] = -2.0*V_r[0]*V_r[2];
R[2][1] = -2.0*V_r[1]*V_r[2];
R[2][2] = 1.0-2.0*Math.pow(V_r[2],2.0);
R[2][3] = 2.0*(dot(P_r, V_r))*V_r[2];
```

One tricky aspect was ensuring that the lighting on the teapot was also reflected appropriately. To make things easier, the reflection matrix was initially multiplied with the model matrix in the JavaScript file before being sent to the vertex shader. However, doing so meant that the model matrix when calculating the lighting also contained the reflection, meaning that the lighting was calculated based on the position of the reflected teapot in the scene, rather than using the lighting that was based on the position of the real teapot in the scene. This produced a result that resembled two teapots stacked on top of each other rather than being a reflection.

To resolve this issue, the reflection and model matrix were sent to the vertex shader separately, and that way only the original model matrix could be applied to the lighting calculations and thus the lighting on the teapot was reflected as well.

The code below shows the reflection matrix, R, being sent as a separate uniform variable in the teapot program's vertex shader.

```
var teapotReflectionMatrixLoc =
gl.getUniformLocation(teapotProgram,"u_Reflection");

//…

gl.uniformMatrix4fv(teapotReflectionMatrixLoc, false, flatten(R));
```

In the vertex shader, the model matrix is multiplied by the reflection matrix before being multiplied by the view matrix, followed by the perspective matrix.

```
gl_Position = u_Perspective * u_View * u_Reflection * u_Model * a_Position;
```

The code below shows that the position value sent to the fragment shader to calculate the lighting only uses the model matrix.

```
v_Position = (u_Model * a_Position).xyz;
```

Since the real teapot and the teapot reflection are created using the same vertex shaders, it was important to remember to send an identity matrix as the reflection matrix for the real teapot so that it wouldn't be reflected.

```
gl.uniformMatrix4fv(teapotReflectionMatrixLoc, false, flatten(mat4()));
```

3.2 Part 2

The step was simply to make the ground appear again by uncommenting the draw function call for the table.

TO execute alpha blending, the alpha parameter when setting up WebGL was first set to false.

```
var gl = WebGLUtils.setupWebGL(canvas, { alpha: false });
```

Then, the draw call for the table was moved to the very end after both teapots were drawn. Alpha blending was enabled and the blending function was set.

```
gl.enable(gl.BLEND);
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

The transparency of the table was changed by simply changing the alpha value inside the table's fragment shader. A value of 0.5 was chosen arbitrarily using trial and error based on how the rendering looked.

```
gl_FragColor.a = 0.5;
```

The blending function was then disabled right after the table was drawn.

```
gl.drawArrays(gl.TRIANGLES, 0, 6);
gl.disable(gl.BLEND);
```

3.3 Part 3

The stencil buffer was implemented using the methodology described in Section 2.3. Note that the implementation from [4]n uses the depth test to enable logic for more complex scenes but given that there is only one mirror and one object being reflected, the additional settings of the depth test done aren't strictly necessary. For simplicity, the depth buffer is completely disabled when the stencil buffer is generated and reenabled when the objects are drawn in the scene.

To improve code organization, separate functions for drawing the teapot and table were created.

To use the stencil buffer, it needs to be set to be available which needs to be initiated when WebGL is set up at the start of the file.

```
var gl = WebGLUtils.setupWebGL(canvas, { alpha: false, stencil: true });
```

At the start of the render function, the stencil, depth, and color buffers are cleared, and the stencil test is disabled.

```
gl.enable(gl.DEPTH_TEST);
gl.disable(gl.STENCIL_TEST);

//…

gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT | gl.STENCIL_BUFFER_BIT);
gl.clearStencil(0);
```

Everything that's not a reflection on the mirror or the mirror itself is then rendered, in this case being just the real teapot.

```
drawTeapot(false);
```

The stencil test is then enabled, while the color and depth buffers are disabled. The stencil buffer's function is set to set a value of 1 for everything that's drawn. Then the table, being the mirrored surface, is then drawn.

```
// create stencil buffer
gl.enable(gl.STENCIL_TEST);
gl.stencilOp(gl.KEEP, gl.KEEP, gl.REPLACE);
gl.stencilFunc(gl.ALWAYS, 1, ~0);
gl.colorMask(0,0,0,0);
gl.disable(gl.DEPTH_TEST);

// draw mirror surface, the table to stencil buffer
drawTable(false);
```

Now that the stencil buffer has the correct values, the function is set to match the table's visible pixels, the color and depth buffer are reenabled, and the reflection is drawn. Since the stencil test is still active, only the pixels of the reflected teapot that are on the table's surface will be drawn.

```
gl.stencilFunc(gl.EQUAL, 1, ~0); // match mirror's visible pixels
gl.stencilOp(gl.KEEP, gl.KEEP, gl.KEEP); // do not change stencil values
gl.colorMask(1,1,1,1); // restore color mask
gl.enable(gl.DEPTH_TEST); // enable depth test

// reflected teapot
drawTeapot(true);
```

Finally, the stencil buffer is disabled, and the visible rendering of the table is drawn.

```
gl.disable(gl.STENCIL_TEST)

// draw table
drawTable(true);
```

3.4 Part 4

The oblique near-plane clipping was implemented by first adding the function provided in [1], which will perform the calculation in Equation (8).

```
function modifyProjectionMatrix(clipplane, projection) {
    // MV.js has no copy constructor for matrices
    var oblique = mult(mat4(), projection);
    var q = vec4((Math.sign(clipplane[0]) + projection[0][2])/projection[0][0],
    (Math.sign(clipplane[1]) + projection[1][2])/projection[1][1],
    -1.0,
    (1.0 + projection[2][2])/projection[2][3]);
    var s = 2.0/dot(clipplane, q);
    oblique[2] = vec4(clipplane[0]*s, clipplane[1]*s,
    clipplane[2]*s + 1.0, clipplane[3]*s);
    return oblique;
}
```

A function called `findClipPlane` was also added to find the value of **C**, the values representing the planar reflector, as shown in Equation (5).

```
function findClipPlane(viewMatrix) {
    var N = vec3(0.0, -1.0, 0.0);
    var P = vec3(0.0, -1.0, -3.0);

    var plane = vec4(N, -1.0*dot(N, P));
    var viewMatrix = transpose(inverse(viewMatrix));

    return mult(viewMatrix, plane);
}
```

The plane's values were stored at the start of the program since the plane's position and orientation wasn't going to change for this project.

```
var reflectionClipPlane = findClipPlane(V);
```

The teapot's motion was also modified to go into the table first to a value of $y = -1.8$ so that the result could be verified

```
var movingUp = false;

//…

if (teapotHeight <= -1.8)
   movingUp = true;
```

The drawTeapot function was also modified to take in the perspective matrix used when drawing so that the oblique-plane clipping would only apply when drawing the reflection.

In the render function, the drawing of the teapot reflection was modified by using the modified projection matrix found using the modifyProjectionMatrix function.

```
// reflected teapot
reflectionP = modifyProjectionMatrix(reflectionClipPlane, P);
drawTeapot(true, reflectionP);
```

The stencil test was then disabled as before, but the depth buffer also needed to be cleared since oblique near-plane clipping made the depth buffer values inconsistent. Since the depth buffer was used right after this inconsistency can cause issues. The drawing of the real teapot was moved to the end so that it was drawn after the table. This facilitated using the depth buffer so that the teapot geometry under the table wouldn't be shown. Note that the regular projection matrix, P, was used for the real teapot.

```
gl.disable(gl.STENCIL_TEST);
gl.clear(gl.DEPTH_BUFFER_BIT);

// draw table
drawTable(true);

// draw real teapot
drawTeapot(false, P);
```

The final functions for the drawTable and drawTeapot functions can be found in the Appendix.

3.5 Additional feature

Implementing the percentage-closer filtering was done by first sending the scale of the shadow map based on Equation (9) as a uniform variable to the fragment shader for both the table and the teapot.

```
var texMapScale = vec2(1.0/fbo.width, 1.0/fbo.height);

//…
var teapottexMapScaleLoc = gl.getUniformLocation(teapotProgram,"texmapscale");
var tabletexMapScaleLoc = gl.getUniformLocation(tableProgram,"texmapscale");

//…
gl.uniform2fv(teapottexMapScaleLoc, texMapScale);
gl.uniform2fv(tabletexMapScaleLoc, texMapScale);
```

With the resolution of the shadow map in the fragment shader, the following code was used to loop through the pixels around the current pixels, find the corresponding shadow map value, and average the collected values to produce a finalVisibility value.

```
vec4 offset_lookup(sampler2D map, vec3 shadowCoord, vec2 offset) {
```

```
    return texture2D(map, shadowCoord.xy + offset*texmapscale);
}

void main()
{
    vec3 shadowCoord = (v_PositionFromLight.xyz/v_PositionFromLight.w)/2.0 + 0.5;
    float finalVisibility = 0.0;
    float x, y;
    for (float y = -5.5; y <= 5.5; y += 1.0) {
        for (float x = -5.5; x <= 5.5; x += 1.0){
            vec4 rgbaDepth = offset_lookup(shadowMap, shadowCoord, vec2(x, y));
            float depth = unpackDepth(rgbaDepth);
            float visibility = (shadowCoord.z > depth + 0.0015) ? 0.0 : 1.0;
            finalVisibility += visibility;
        }
    }
    finalVisibility = finalVisibility / 144.0;

    //…

}
```

The `finalVisibility` value simply replaced the visibility value used with the shadow mapping, shown below for the teapot program as an example.

```
gl_FragColor.rgb = L_o * finalVisibility + L_a * u_diffuseCoefficient;
```

Note that the percentage-closer filtering was applied to all parts of the project.

4.0 Results

4.1 Part 1

The result from Part 1 can be shown in Figure 1 below.

Figure 1: Result of Part 1

The reflection is shown in the correct position and changing the real teapot's position moves the reflected teapot in the expected direction (opposite that of the real teapot). When the light rotation is toggled, the lighting effects on both teapot objects are stopped as expected. The lighting effects are reflected as well thanks to the separation of the reflection and model matrices.

4.2 Part 2

The result from Part 2 can be shown in Figure 2 below.

Figure 2: Result of Part 2

The ground is now visible, and the reflection of the teapot can be seen underneath the table. The alpha blending is implemented correctly as the color of the teapot's reflection now has a blueish tint from the table's marble surface. With the chosen value of 0.5 the author has determined that it is similar to what a real marble reflection would look like.

Although difficult to see, the lighting and shadow effects on the table and the teapot reflection are still rendered properly.

4.3 Part 3

The issue from Part 2 is shown in Figure 3, while the resolution from Part 3 is shown in Figure 4 below.

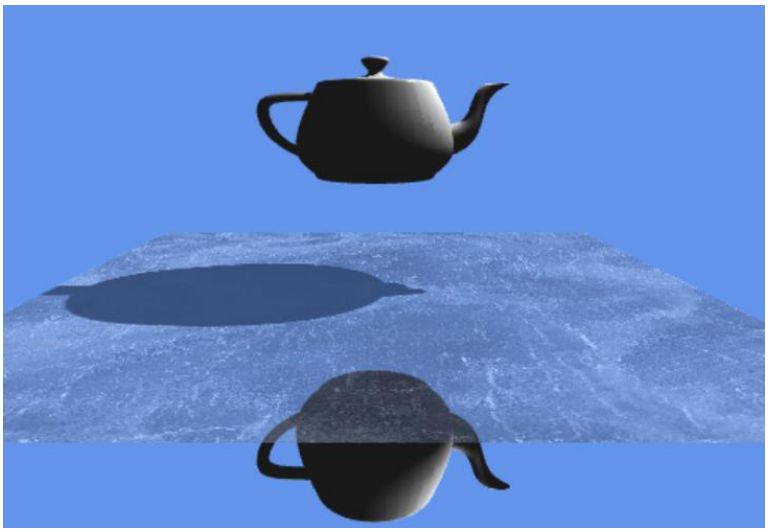

Figure 3: Teapot reflection being seen past table edge

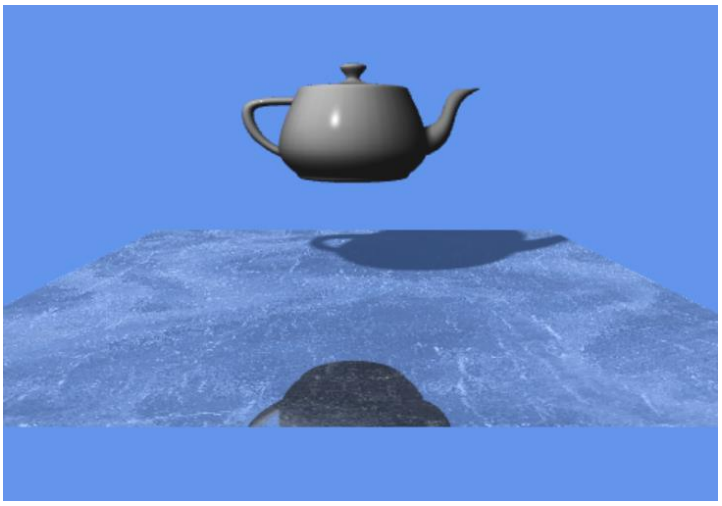

Figure 4: Result of Part 3

When the teapot is raised, the reflection can no longer be seen past the edge of the table. With the rotating light, all the shadows are still rendered as expected. Hence, the clipping of the reflection worked as expected.

4.4 Part 4

The result from Part 4 is shown in Figure 5 below.

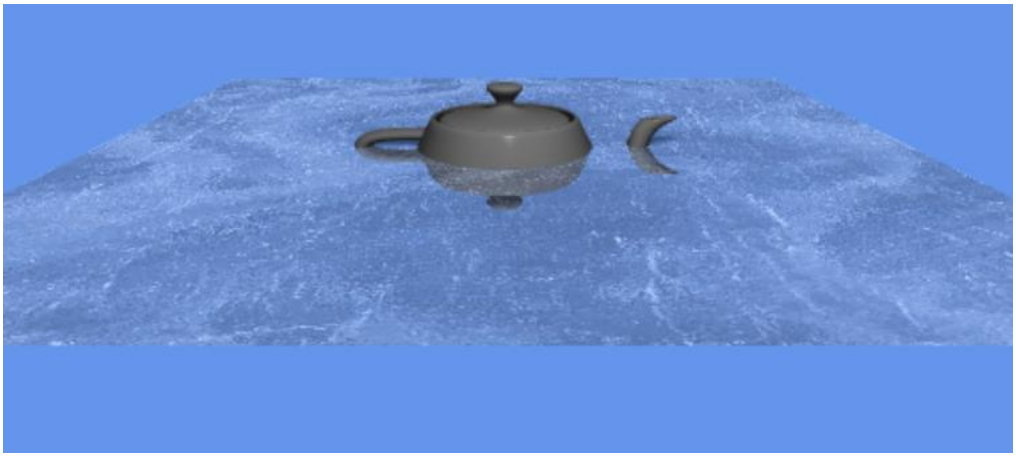

Figure 5: Result of Part 4

All features from previous parts are retained as expected. With the translation of the teapot changed to go down first, it is easy to see first that as the real teapot descends into the table, the reflection for the geometry under the table no longer visible. The teapot can be lowered to a point where only the top nub from the lid is visible. Since only the nub is still reflected while the rest of the teapot isn't the desired behavior has been achieved. There are also no remanent shadow effects on the table, further confirming that the results are as expected.

4.5 Additional feature

Figure 6 and Figure 7 below show the difference in if percentage-closer filtering is included.

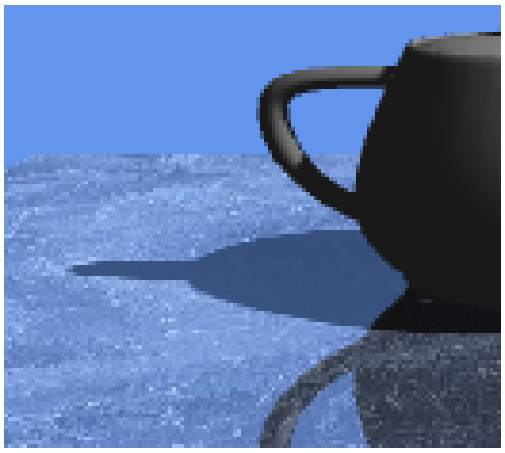

Figure 6: Aliasing effects present

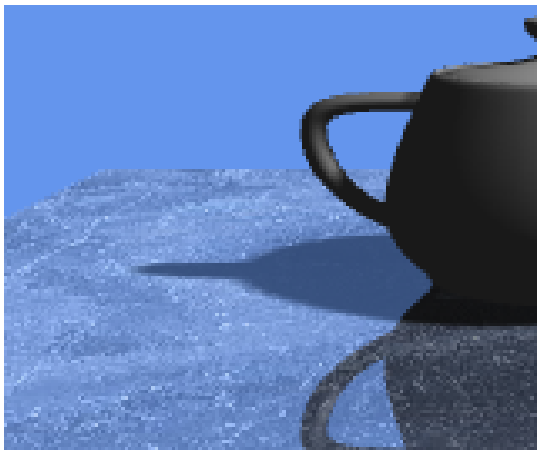

Figure 7: Percentage-closer filtering applied

Although not very apparent, the staircasing or aliasing effect can be seen in Figure 6. This effect is removed and instead replaced with a blurring effect, which is expected. The number of nearby pixels used to calculate the proper visibility value was adjusted based on trial and error until the value of 144 was chosen to give a good balance of masking the aliasing effect while also not distorting the edges of the shadow too much.

Discussion

For part 4, note that initially the value of N in the `findClipPlane` function was set to (0.0, 1.0, 0.0) as described in Section 2.4. However once implemented the clipping effect ended up being applied on the negative side, meaning that only the positive portions of the reflections (ie. the portions that should not be shown) were visible. The normal vector should be set to point toward the direction where objects should be visible, away from the direction where the clipping would occur. Since the negative side is shown, the normal vector implemented in the code was flipped which produced the desired result. Hence, the initial assumption made in Section 2.4 was wrong.

There are some mild lighting artifacts on the teapot once the percentage-closer filtering was applied in the form of a grid-like texture. To avoid this, it could have been beneficial to use a model of the teapot that was generated using subdivisions rather than an obj file as the resolution of the obj file cannot be adjusted.

Other than that, the results of the implementation match the expected results, and the planar reflection of the teapot was done successfully.

Another potential idea for future work is that instead of considering the table being made of marble and including the step of making the real teapot disappear under the table, the material of the “table” can be considered as water. In that case, the real teapot should still be visible once under the table, but additional refraction and possibly water physics should be applied to render a more realistic display. This wouldn’t necessarily be “better” than a marble implementation, but it would certainly add additional difficulty level to the project in considering the extra lighting effects.

References

[1] J. Frisvad, "Project Initiator: Planar Reflector," [Online]. Available: https://learn.inside.dtu.dk/d2l/le/content/125764/viewContent/469931/View. [Accessed 1 December 2022].

[2] D. Blythe and T. McReynolds, "Reflections," in *Advanced Graphics Programming Using OpenGL*, San Francisco, Morgan Kaufmann, 2005, pp. 404-411.

[3] J. Frisvad, "Projection Shadows, blending, and depth sorting," November 2019. [Online]. Available: https://learn.inside.dtu.dk/d2l/le/content/125764/viewContent/469899/View. [Accessed 15 December 2022].

[4] M. J. Kilgard, "Improving Shadows and Reflections via the Stencil Buffer," [Online]. Available: https://www.cs.rpi.edu/~cutler/classes/advancedgraphics/S19/papers/kilgard_stencil_buffer.pdf. [Accessed 1 December 2022].

[5] E. Lengyel, "Reflections and Oblique Clipping," in *Mathematics for 3D Game Programming*, Boston, Cengage Learning, 2011, pp. 120-128.

[6] J. Frisvad, "Shadow mapping and off-screen buffers," November 2019. [Online]. Available: https://learn.inside.dtu.dk/d2l/le/content/125764/viewContent/469901/View. [Accessed 1 December 2022].

[7] M. Bunnell and F. Pellacini, "Chapter 11. Shadow Map Antialiasing," NVIDIA, 2004. [Online]. Available: https://developer.nvidia.com/gpugems/gpugems/part-ii-lighting-and-shadows/chapter-11-shadow-map-antialiasing. [Accessed 1 December 2022].

Appendix

Below is the code for the drawTable and drawTeapot functions.

```
function drawTable(alphaBlend){
    gl.useProgram(tableProgram);
    initAttributeVariable(gl, tableProgram.a_Position, positionbuffer);
    initAttributeVariable(gl, tableProgram.a_TexCoord, tBuffer);
    if (alphaBlend) {
        gl.enable(gl.BLEND);
        gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
    }
    gl.uniformMatrix4fv(tableViewMatrixLoc, false, flatten(V));
    gl.uniformMatrix4fv(tableLightViewMatrixLoc, false, flatten(lightV));
    gl.uniformMatrix4fv(tableLightPerspectiveMatrixLoc, false, flatten(lightP));
    gl.uniformMatrix4fv(tablePerspectiveMatrixLoc, false, flatten(P));
    gl.uniform1f(tableAmbientIntensityLoc, ambientIntensity);
    gl.uniform2fv(tabletexMapScaleLoc, texMapScale);
    gl.uniform1i(gl.getUniformLocation(tableProgram, "shadowMap"), 0);
    gl.uniform1i(gl.getUniformLocation(tableProgram, "marbletexMap"), 1);
    gl.drawArrays(gl.TRIANGLES, 0, 6);
    if (alphaBlend)
        gl.disable(gl.BLEND);
}

function drawTeapot(reflection, perspectiveMatrix) {
    gl.useProgram(teapotProgram);
    initAttributeVariable(gl, teapotProgram.a_Position, model.vertexBuffer);
    initAttributeVariable(gl, teapotProgram.a_Normal, model.normalBuffer);
    initAttributeVariable(gl, teapotProgram.a_Color, model.colorBuffer);
    var N = normalMatrix(modelMatrix, true);
    gl.uniform3fv(cameraPositionLoc, eyePos);
    gl.uniform4fv(teapotLightPositionLoc, flatten(lightPos));
    gl.uniform3fv(teapotLightIntensityLoc, flatten(lightIntensity));
    gl.uniform1f(teapotAmbientIntensityLoc, ambientIntensity);
    gl.uniform1f(teapotDiffusionCoefficientLoc, diffusionCoefficient);
    gl.uniform1f(teapotSpecularCoefficientLoc, specularCoefficient);
    gl.uniform1f(teapotShininessCoefficientLoc, shininessCoefficient);
    gl.uniformMatrix4fv(teapotViewMatrixLoc, false, flatten(V));
    gl.uniformMatrix4fv(teapotPerspectiveMatrixLoc, false,
flatten(perspectiveMatrix));
    gl.uniformMatrix4fv(teapotLightViewMatrixLoc, false, flatten(lightV));
    gl.uniformMatrix4fv(teapotLightPerspectiveMatrixLoc, false, flatten(lightP));
    gl.uniformMatrix4fv(teapotModelMatrixLoc, false, flatten(modelMatrix));
    gl.uniformMatrix3fv(teapotNormalMatrixLoc, false, flatten(N));
    gl.uniform1i(gl.getUniformLocation(teapotProgram, "shadowMap"), 0);
```

```
    gl.uniform2fv(teapottexMapScaleLoc, texMapScale);
    var R = mat4();
    if (reflection) {
        var V_r = vec3(0.0, 1.0, 0.0);
        var P_r = vec3(0.0, -1.0, -3.0);
        R[0][0] = 1.0-2.0*Math.pow(V_r[0],2.0);
        R[0][1] = -2.0*V_r[0]*V_r[1];
        R[0][2] = -2.0*V_r[0]*V_r[2];
        R[0][3] = 2.0*(dot(P_r, V_r))*V_r[0];
        R[1][0] = -2.0*V_r[0]*V_r[1];
        R[1][1] = 1.0-2.0*Math.pow(V_r[1],2.0);
        R[1][2] = -2.0*V_r[1]*V_r[2];
        R[1][3] = 2.0*(dot(P_r, V_r))*V_r[1];
        R[2][0] = -2.0*V_r[0]*V_r[2];
        R[2][1] = -2.0*V_r[1]*V_r[2];
        R[2][2] = 1.0-2.0*Math.pow(V_r[2],2.0);
        R[2][3] = 2.0*(dot(P_r, V_r))*V_r[2];
    }
    gl.uniformMatrix4fv(teapotReflectionMatrixLoc, false, flatten(R));
    gl.drawElements(gl.TRIANGLES, g_drawingInfo.indices.length,
gl.UNSIGNED_SHORT, 0);
}
```