Computer Graphics - 02561: Project Report

Topic: Planar Reflector

Giacomo Barzon: s201434

December 8, 2020

Introduction

This is the report related to the Final Project of the *Course Computer Graphics - 02561*.
The code of this project was produced in group with *Matteo Bordin - s201655*, we decided however to make individually two different project reports.
We chose to implement a planar reflector as our final project. We followed closely the instructions listed in the project initiator[2] together with the papers that were to provided in order to implement a working planar reflector.
In the following sections I'm going to explain in detail what we have done.

The webpage with all of the working exercises completed by me can be found at the following link

Part 1

We started from the result of part 2 of the Worksheet 9 which consisted in the implementation of shadow mapping.
After adjusting the scene as described in the project initiator[2] the first thing to do was drawing a reflected teapot like explained in [4]. To do that we created a reflection transformation matrix R.

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

Where P is a vertex of the planar reflector and V is a vector perpendicular to the plane.
We first render the teapot, then we can use the same shader to draw the reflection by multiplying the model view matrix of the teapot by the reflection transformation matrix that we built before. We also have to multiply the vector related to our light by the same reflection transformation matrix in order to mirror the lighting too.
After doing so we should see a result similar to the one that can bee seen in figure 1.
To allow us to notice much more easily the reflection of the teapot i temporarily removed the rendering of the plane.

Part 2

If we reinsert our ground plane and blend it with the mirrored surface by making it partially transparent like we did in the Worksheet 8 we can obtain a nice mirrored reflection of the teapot like the one that you can see in figure 2.

Part 3

This approach however has one major problem, if we move our teapot high enough we can notice that part of the teapot appears below the ground quad breaking thus the illusion like in figure 3. We can solve this problem by using the stencil buffer like explained in chapter 3 of [3]. The stencil buffer stores an integer value for each pixel, and allows us to discard the fragments related to specific pixel that we don't want to render.
For example we can set all the fragments that we want to render to 1 and all the other ones to 0, and set the stencil test so that it renders only fragments related to a pixel with a value in the stencil buffer of 1, all the other ones will be ignored.

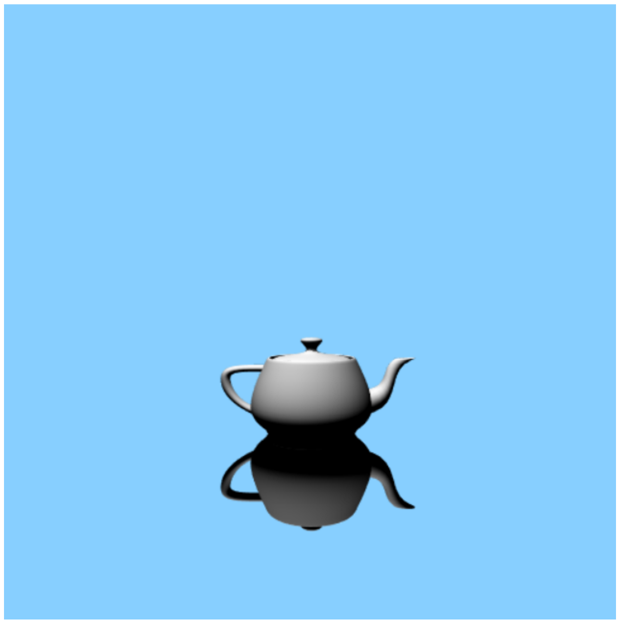
Figure 1: Result from Part 1

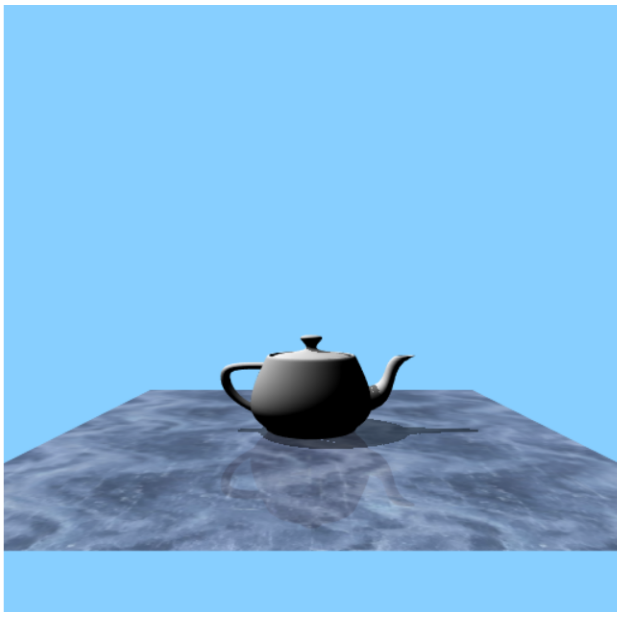
Figure 2: Result from Part 2

Firstly we clear all our buffers and enable the stencil test and the depth test:

```
glClearStencil(0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
 GL_STENCIL_BUFER_BIT);
glEnable(GL_DEPTH_BUFFER_BIT);
glDisable(GL_STENCIL_TEST);
```

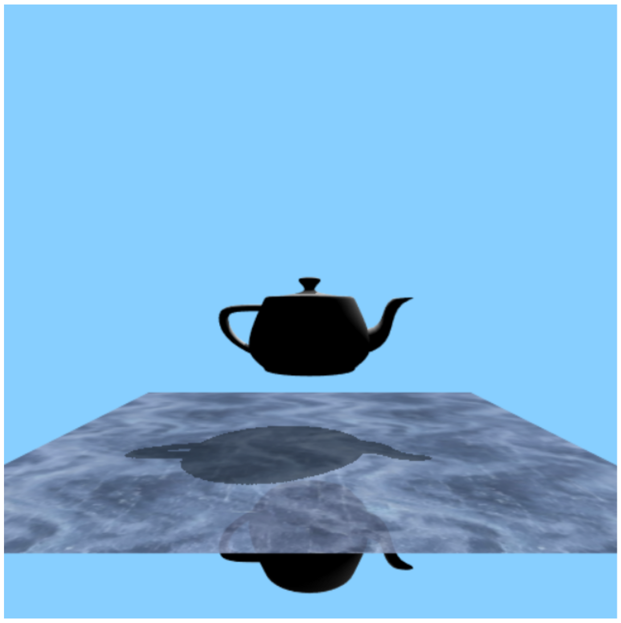

Figure 3: Part of the object not covered by the ground are rendered too

We render all our non reflective objects in the scene(In this case our teapot) and then for each mirror that we have in our scene we do the following operations.

- We setup the stencil test in order to write a value of 1 into the stencil buffer whenever the depth test passes and we also disable the color buffer.

  ```
  glEnable(GL_STENCIL_TEST);
  glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
  glStencilFunc(GL_ALWAYS, 1, ~0);
  glColorMask(0,0,0,0);
  ```

  Now we draw our plane, this tags in the stencil buffer all the pixels of our plane with a value of 1. The depth test ensures that all the pixels of our plane that are occluded are not going to be tagged.

- We setup the depth test to always pass and to set the depth range to write only the farthest possible value for all the updated pixels. Moreover we set the stencil test to update only pixels tagged with a value of 1.

  ```
  glDepthRange(1,1);
  glDepthFunc(GL_ALWAYS);
  glStencilFunc(GL_EQUAL, 1, ~0);
  glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
  ```

  After this we draw our ground mirror surface. This step resets the depth buffer to its cleared maximum value for all the mirror's visible pixels.

- Restore the depth test, color mask, and depth range to their standard settings

  ```
  glDepthFunc(GL_LESS);
  glColorMask(1,1,1,1);
  glDepthRange(0,1);
  ```

  Now we are ready to render the reflection since all the pixels belonging to it are tagged with a stencil value of 1 and the stencil test is still set to update only the pixels with a value of 1. Moreover the less than depth test will determine visible surfaces appropiately.

- Finally, we reset to zero the stencil value of all the mirror's pixels. Before doing this we have to remember to render also our reflective surface otherwise it wonìt' appear on the screen.

  ```
  glColorMask(0,0,0,0);
  glStencilOp(GL_KEEP, GL_KEEP, GL_ZERO);
  glDepthFunc(GL_ALWAYS);
  renderMirrorSurfacePolygons(thisMirror);
  glDepthFunc(GL_LESS);
  glColorMask(1,1,1,1);
  ```

Notice how we skipped the fourth step addressed in chapter 4 of [3]. This step in fact assumes that we can use some specific functions of OpenGl that are unfortunately unavailable in WebGl. The problems addressed by this step however are going to be fixed with the next part of the project initiator[2].

If we applied the process correctly we should obtain a result similar to the one in figure 4.

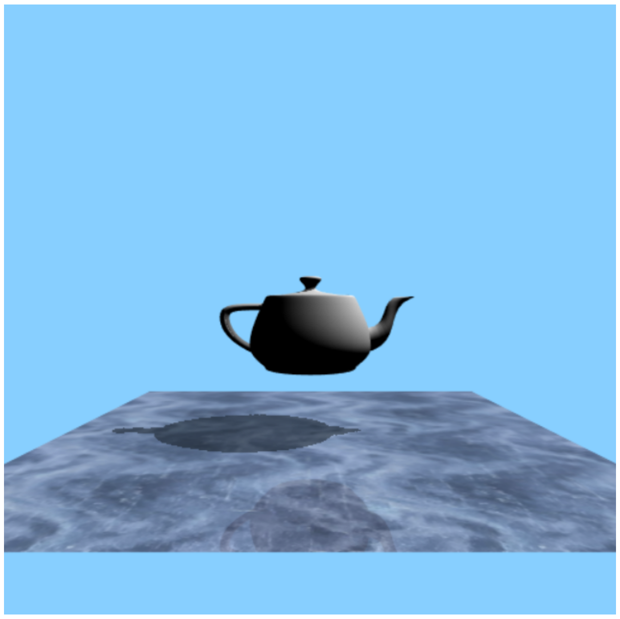

Figure 4: Result from Part 3

Part 4

This approach however has still one more problem. The reflector plane reflects also objects that are behind the reflector surface. This problem can be seen very clearly in figure 5. An approach to solve

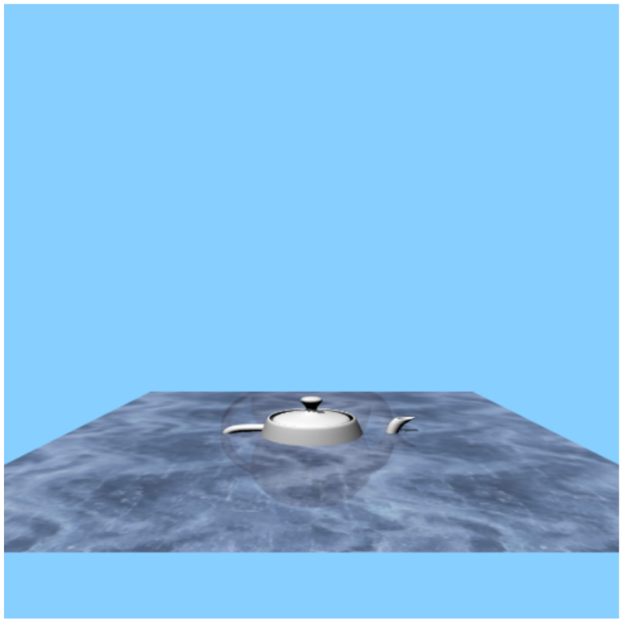

Figure 5: The reflector plane reflects also objects that are behind the reflective surface

this is explained in detail in [1]. This approach is based on an user defined clipping plane which clips all the objects below him, and thus doesn't render them.

Normally every primitive is clipped on the six sides of the view frustum, instead of adding a seventh clipping plane which represents the reflective surface we want to modify the perspective matrix that we normally use to render our objects in the scene in a way that repositions the near plane in the position of our reflective surface.
The approach described in [1] is implemented in the following function which is provided in the project initiator[2].

```
function modifyProjectionMatrix(clipplane, projection) {
    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;
}
```

This function requires that we provide the equation of our plane surface and the projection matrix and returns a transformed projection matrix that clips all of the objects below the plane given as input.
If we upload this transformed projection matrix to the shaders instead of the normal one while rendering the reflection we can solve the problem described above.

Now if we correctly implement all the steps listed above the final result should look like the one in figure 6

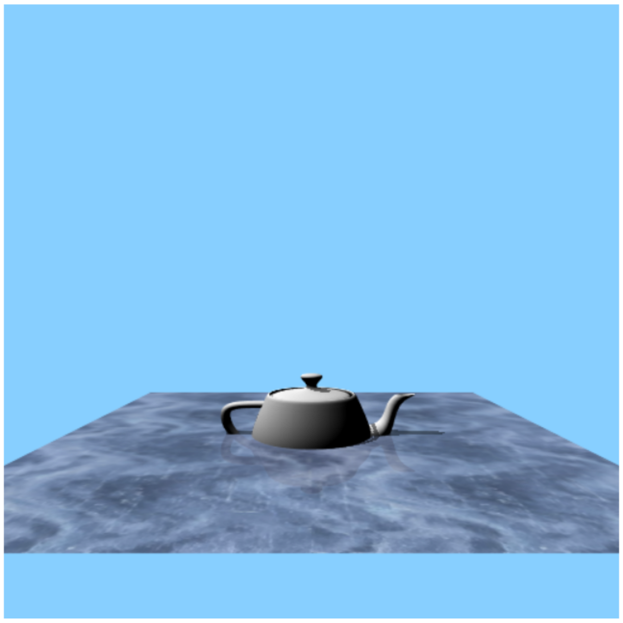

Figure 6: Final Result

Bibliography

[1] Lengyel E. “Reflections and Oblique Clipping”. In: *Mathematics for 3D Game Programming and Computer Graphics* (2005).

[2] Jeppe Revall Frisvad. *Project Initiator 1: Planar Reflector*. URL: `https://cn.inside.dtu.dk/cnnet/filesharing/download/439f0c67-9972-4085-88ad-10c5acbadaf9`.

[3] Kilgard M. “Improving Shadows and Reflections via the Stencil Buffer”. In: *Advanced OpenGL Game Development* (1999).

[4] Blythe D. McReynolds T. “Reflections. In Advanced Graphics Programming Using OpenGL”. In: *Advanced Graphics Programming Using OpenGL* (2005).