Planar reflection

Authors

Leonardo Rodovero - s240095
Samer Bujana Escalona - s240033

December 17, 2024

Contents

1 Introduction

Realistic reflections are an integral part to achieve convincing computer graphics scenes. Regardless of how common they are, the process to achieve them is nontrivial in a rasterization-based environment like WebGL. While ray tracing can handle reflections of arbitrary complexity, planar reflections using conventional rasterization require careful application of transformations, blending, and buffer techniques. In particular, planar reflectors are commonly used to simulate mirrors, still water surfaces, or polished floors.

Several references were key to understand and implement our work. The foundations of planar reflections and clipping are derived from McReynolds [1], blending techniques are discussed by Angel [2], and the oblique near plane clipping approach is presented by Lengyel [3]. Additional insights come from practical worksheets and exercises from the lectures that guide the implementation details in a WebGL context.

Our work begins with the basic geometry mirroring of an object across a reflection plane. Partial transparency is introduced to simulate a convincing reflective surface. The stencil buffer is then used to ensure that the reflection only appears within the boundaries of the reflector. Finally, the oblique near plane clipping technique is employed to correctly clip submerged geometry, avoiding unrealistic appearances of object parts hidden behind the reflection plane.

2 Method

The starting point for the reflection process is identifying the reflecting plane. Suppose the reflector is a plane defined by a point $\mathbf{P}$ on it and a normal vector $\mathbf{V} = (V_x, V_y, V_z)$. The reflection of an object across this plane can be achieved using a reflection matrix R. Following Goldman and McReynolds [1], the reflection matrix in homogeneous coordinates is defined as:

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

Applying R to the object's vertex positions produces a mirrored version of the geometry. To make the reflection visible beneath the ground plane, the ground is drawn with an α value below 1.0. Using alpha, blending is described by Angel [2] as:

$$C_{\text{final}} = \alpha_s C_s + (1 - \alpha_s) C_d,$$

where C_s is the source color (the ground), C_d is the destination (the reflected object), and α_s is the source alpha. This ensures the reflected object appears as if under a translucent surface.

Next, to keep the reflection within the plane's boundaries, the stencil buffer is used. This buffer, introduced in Angel [2] and McReynolds [1], allows us to define specific pixels where drawing is permitted. By first rendering the ground plane into the stencil buffer with color

disabled, we assign a particular stencil value (e.g., 1) to those pixels covered by the reflector. Following rendering of the object will be restricted to pixels matching the set stencil value. This way, there will not be any reflections outside of the margins of the reflector plane.

The final step involves handling geometry when its position is located beneath the reflector. If an object passes behind the reflection plane, its reflection should no longer be rendered. To achieve this, we use oblique near plane clipping as described by Lengyel [3]. Before continuing let's briefly clarify some concepts:

- *Viewing frustum*: it is a 3D geometric shape that represents the visible region of a scene. Defines what the camera can see.
- *Near plane*: this plane is perpendicular to the camera's view direction and the closest of the viewing frustum. Defines the minimum visible distance. Objects closer than this plane are clipped.

The idea is to modify the projection matrix so that its near plane aligns with the reflection plane. This causes the standard GPU pipeline to clip all geometry behind that plane automatically, with no extra shader complexity.

To achieve oblique clipping, we define a clip plane (a, b, c, d) in eye space. The modified projection matrix is computed by a method such as:

$$\text{modifyProjectionMatrix}(\textbf{clipplane}, \textbf{projection}) \rightarrow \textbf{projection}',$$

where **projection** is the original perspective projection matrix. The method adjusts the third row of **projection** based on the **clipplane** coefficients so that the near plane coincides with the reflector. After rendering the reflected objects with this modified projection, the depth buffer must be cleared to restore consistency for following passes, as suggested by Lengyel [3].

3 Implementation

The implementation in WebGL follows standard procedures. After setting up the camera (e.g. at $(0, 1, 1)$) and the projection (e.g., using a 50-degree field of view), the scene is rendered as follows:

1. Render the main object (a teapot) normally using a Phong-shaded program.
2. Enable blending:

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

   Set the ground plane to have an $\alpha < 1.0$ so it is partially transparent.
3. Enable the stencil buffer when creating the WebGL context:

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

Draw the ground plane into the stencil buffer only (colorMask and depthMask off), marking pixels with a known value, e.g., 1.

4. Re-enable color and depth, set the stencil function to allow drawing only where the stencil equals 1, and render the reflected teapot using R. The reflection now appears only when beneath the ground plane.

5. Compute a clip plane corresponding to the reflection plane in eye space. Use 'modifyProjectionMatrix' from Lengyel's approach [3] to produce an oblique projection. Render the reflected object again with this new projection so that any parts behind the reflector are clipped. Clear the depth buffer afterwards.

For reference, this is our code for modifying the projection matrix:

```
function modifyProjectionMatrix(clipplane, projection) {
    var oblique = mult(mat4(), projection);

    var pos1 = (sign(clipplane[0]) + projection[0][2]) / projection[0][0];
    var pos2 = (sign(clipplane[1]) + projection[1][2]) / projection[1][1];
    var pos3 = -1.0;
    var pos4 = (1.0 + projection[2][2]) / projection[2][3];

    var q = vec4(pos1, pos2, pos3, pos4);
    var s = 2.0 / dot(clipplane, q);

    pos1 = clipplane[0] * s;
    pos2 = clipplane[1] * s;
    pos3 = clipplane[2] * s + 1.0;
    pos4 = clipplane[3] * s;

    oblique[2] = vec4(pos1, pos2, pos3, pos4);

    return oblique;
}
```

After these steps, the rendered output shows a convincing planar reflection that handles transparency, clipping to the plane's area, and manages reflections when the geometry is out of the reflection space.

4 Results

The final result is a scene where a teapot rests on a partially transparent reflective surface. The reflection of the teapot is visible when the object lies beneath the ground plane,

neatly confined to the plane's boundaries. As the teapot moves or "jumps", the reflection changes accordingly, never showing parts of the object that lie behind the reflective plane. The end result can be seen Fig. 1) that can be viewed from various angles and under different lighting conditions.

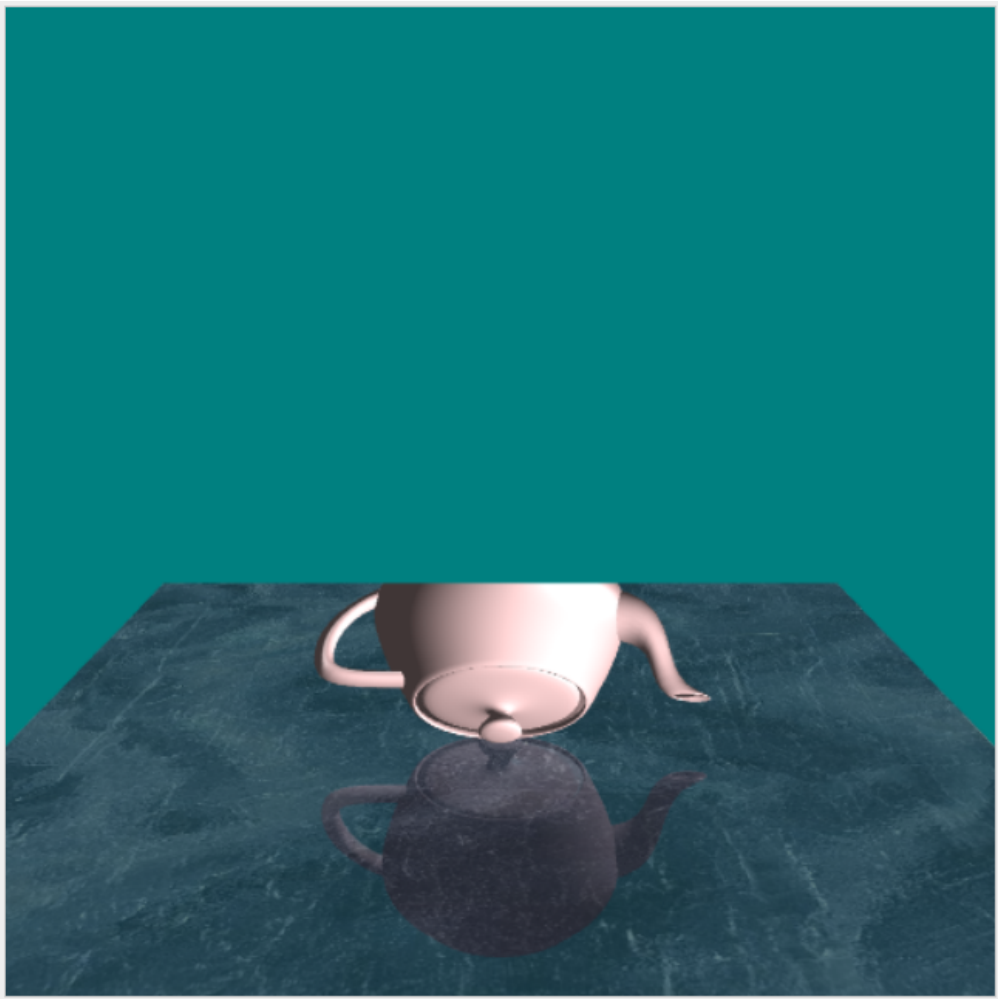

Figure 1: Clipping an object that lies under the ground plane using the view frustum

5 Discussion

The combination of geometric reflection via matrix transformations, blending for visual coherence, stencil buffering for spatial constraints, and oblique near plane clipping for submerged geometry produces a fully functional and efficient pipeline for planar reflections. Each step leverages standard hardware features and well-established techniques from computer graphics literature. Although more advanced scenarios (e.g., curved reflectors, multiple bounces) would demand more complex approaches, this project illustrates how to achieve quality planar reflections in a real-time rasterization context.

By referencing McReynolds [1] for planar reflections, Angel [2] for blending and buffers, and Lengyel [3] for oblique clipping, we adopt proven methods that are both conceptually clear and technically feasible. This approach can serve as a template to build upon, integrate with other effects, or adapt to other platforms and shading languages.

6 Contributions

This project was developed collaboratively by Leonardo and Samer, who both contributed to the overall execution. While the Introduction and Discussion sections were written jointly the responsibility for the core content derived from the four integrated parts was divided between them.

Leonardo took the lead on:

- **Part 1 - Reflected Object:** He was primarily responsible for formulating the reflection transformation matrices, explaining the geometric theory behind mirroring objects across a planar surface, and integrating references from McReynolds[1] to ground the approach in established literature.
- **Part 4 - Frustum View for Clipping Submerged Objects (Oblique Near Plane Clipping):** He researched and implemented the mathematical details of aligning the near plane of the view frustum with the reflector plane, drawing extensively on [3] work.

Samer took the lead on:

- **Part 2 - Reflector:** He focused on reintroducing the ground plane and establishing blending techniques ([2]) to achieve partial transparency. Samer integrated these ideas into WebGL, ensuring that the reflected object appeared realistically beneath a semi-transparent reflective surface.
- **Part 3 - Clipping with stencil buffer:** Samer implemented the stencil-buffer-based clipping technique, as outlined by [1]. He managed the setup of the stencil buffer so that the reflection would appear only where the reflector geometry resided.

References

[1] T. McReynolds and D. Blythe, *Advanced graphics programming using OpenGL*. Elsevier, 2005.

[2] E. Angel and E. Shereiner, "Interactive computer graphics, a top-down approach with webgll," 2015.

[3] E. Lengyel, *Mathematics for 3D game programming and computer graphics*. Course Technology Press, 2011.