Technical University of Denmark

02561 Computer Graphics

Planar Reflector in WebGPU

Blending, Stencil Masking, and Oblique Near-Plane Clipping

Student:
Bashar Bdewi (s183356)

Links:
Lab journal: `https://basharbd.github.io/02561-computer-graphics/`
Project implementation: `https://github.com/basharbd/02561-computer-graphics`

Date:
December 17, 2025

Contents

Abstract

This project implements a real-time planar reflector using WebGPU rasterization. Unlike ray tracing, where reflections are physically simulated, rasterization requires geometric approximations. This implementation constructs a convincing reflection of a dynamic object (a jumping teapot) on a textured ground plane by solving three critical visibility challenges: transparency, boundary clipping, and submerged geometry artifacts.

The final solution integrates four distinct graphics techniques: (1) a reflection transformation matrix to mirror geometry, (2) alpha blending to simulate a semi-transparent surface, (3) stencil buffering to mask the reflection to the reflector's specific footprint, and (4) oblique near-plane clipping (Lengyel's method) to cull geometry located physically behind the mirror plane. The result is a robust rendering pipeline that runs efficiently in a web browser, handling complex view-dependent effects without the computational cost of ray tracing.

1 Introduction

Reflections are essential for grounding objects in a 3D scene, providing visual cues about spatial relationships and surface materials. In real-time rasterization pipelines like WebGPU, planar reflections are typically approximated by rendering the scene twice: once from the standard camera view, and once from a "mirrored" perspective.

While the basic mirrored geometry is trivial to compute, creating a robust illusion requires handling several edge cases. A simple mirrored object will "leak" outside the reflective surface boundaries and will incorrectly display parts of the object that penetrate below the floor. This project addresses these artifacts in the context of a WebGPU pipeline, specifically focusing on the coordination of Render Passes, Depth/Stencil states, and Projection Matrix modification.

2 Method

The scene consists of a teapot moving vertically ($y \in [-1.8, 0]$) above a ground plane at $y = -1$. The rendering process is composed of four logical stages.

2.1 Mirror Rendering via Reflection Transform

The first step creates the illusion of a reflection by mirroring the teapot geometry across the plane $y = -1$. This is achieved by applying a reflection matrix R to the object's Model matrix:

$$R = T(0, -1, 0) \cdot S(1, -1, 1) \cdot T(0, 1, 0) \tag{1}$$

where T represents translation and S represents scaling. $S(1, -1, 1)$ flips the geometry across the XZ plane. The translations move the coordinate system so the scaling occurs relative to the reflector plane ($y = -1$). The reflected model matrix becomes $M_{\text{refl}} = R \cdot M$.

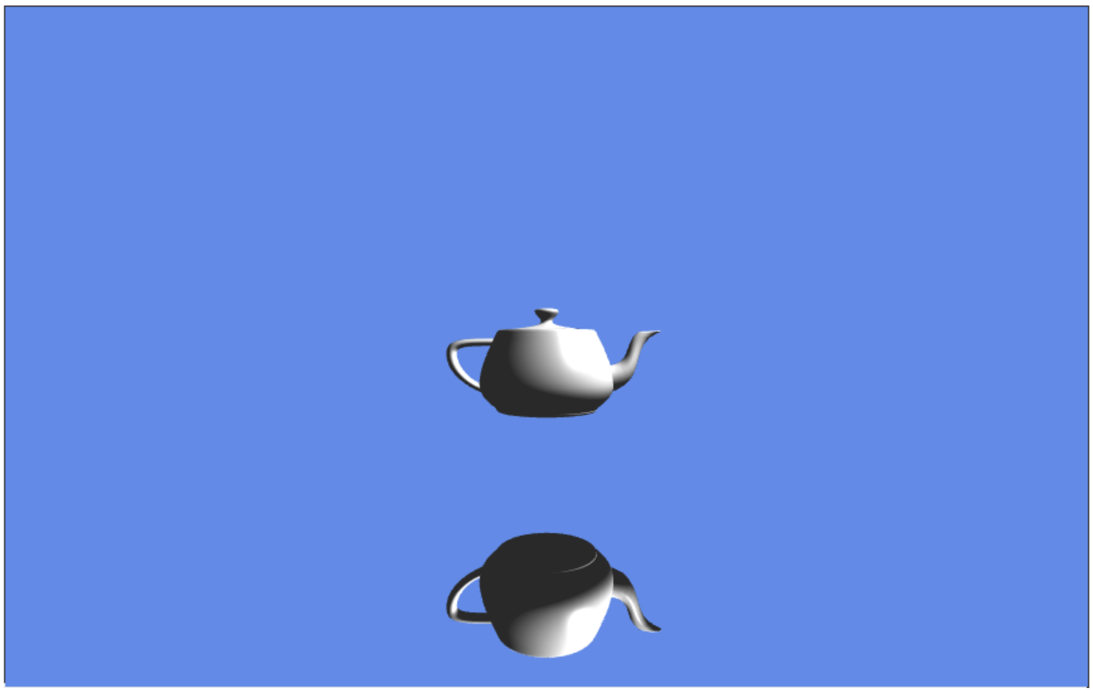

Figure 1: Mirror Rendering: The teapot geometry is reflected across the plane $y = -1$ using a transformation matrix, creating an inverted copy that mimics a physical reflection.

2.2 Ground Blending

To ensure the reflector looks like a polished surface rather than a perfect mirror, the ground plane is rendered with transparency. Standard alpha blending is used:

$$C_{\text{final}} = \alpha \cdot C_{\text{src}} + (1 - \alpha) \cdot C_{\text{dst}} \tag{2}$$

where C_{src} is the textured ground color and C_{dst} is the reflection color already present in the framebuffer. In WebGPU, this is configured in the pipeline blend state with `srcFactor: "src-alpha"` and `dstFactor: "one-minus-src-alpha"`.

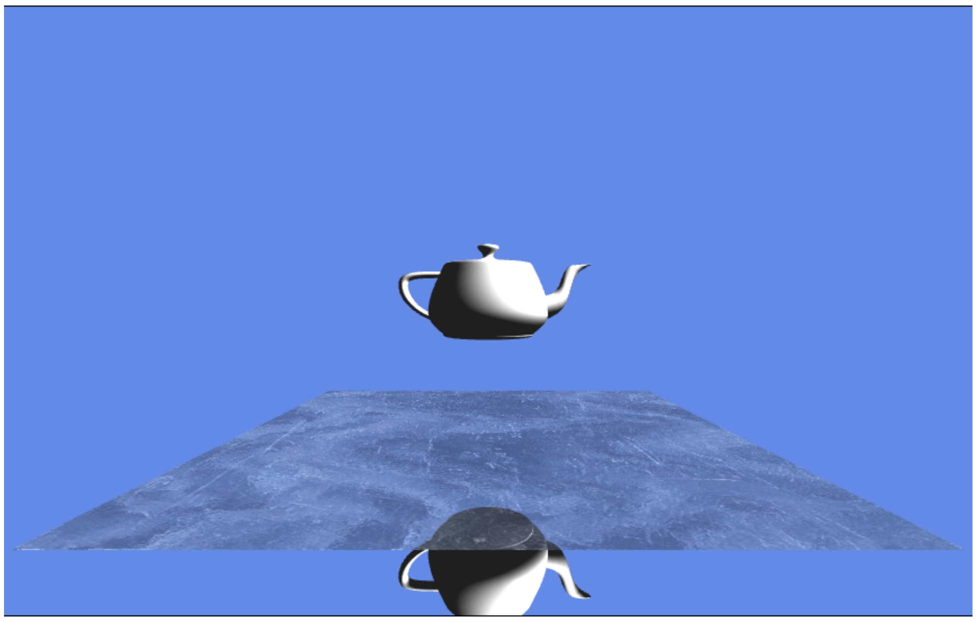

Figure 2: Blending enables the reflected teapot to be seen through the textured ground surface.

2.3 Stencil Masking for Clipping

A naive reflection is drawn everywhere on the screen, even where the ground plane does not exist. To fix this "leaking," the Stencil Buffer is used as a mask.

1. **Mask Pass:** The ground geometry is rendered first. Color writing is disabled (`writeMask: 0`). The stencil operation is set to `replace`, writing a reference value of `1` to the stencil buffer pixels covered by the ground.
2. **Reflection Pass:** The reflected teapot is drawn. The stencil comparison is set to `equal` (ref `1`). Fragments are only generated where the stencil value matches 1, effectively clipping the reflection to the ground's shape.

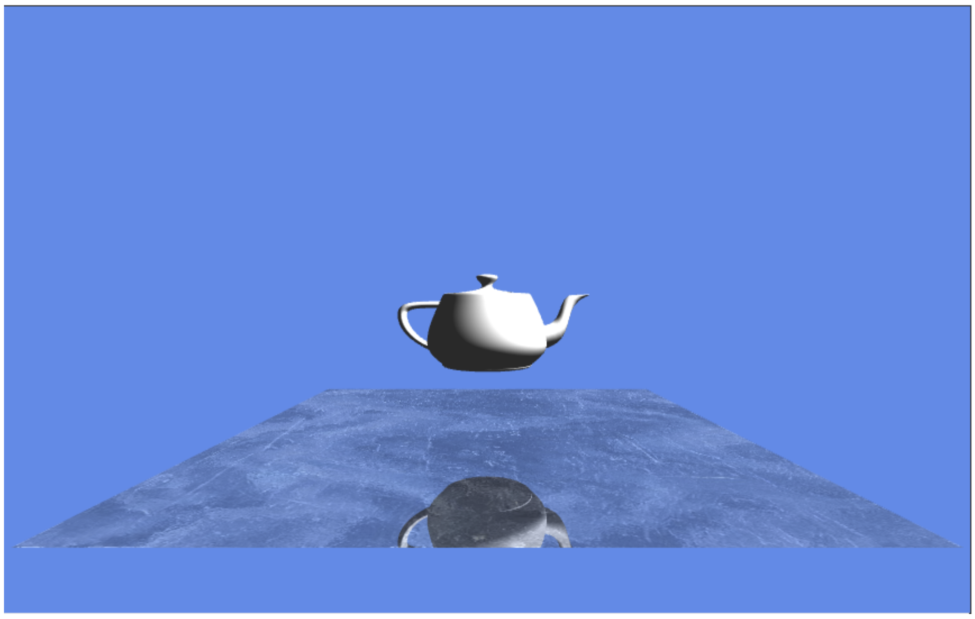

Figure 3: Stencil masking restricts the reflection (bottom) strictly to the rectangular ground area.

2.4 Oblique Near-Plane Clipping

When the teapot dips below the ground ($y < -1$), the simple reflection matrix mirrors the "submerged" geometry above the floor, destroying the illusion. To solve this, we use Lengyel's Method to modify the camera's projection matrix.

The near clipping plane of the view frustum is mathematically moved to align with the reflector plane. Given the reflector plane equation $C = (0, 1, 0, 1)$ in World Space, we transform it to Eye Space:

$$C_{\text{eye}} = (V^{-1})^T \cdot C_{\text{world}} \tag{3}$$

The projection matrix P is then distorted into P' such that any geometry behind C_{eye} is clipped by the hardware rasterizer.

Depth Buffer Inconsistency: A side effect of this distortion is that the depth values generated by P' do not match those of P. This necessitates clearing the depth buffer between rendering the reflection and the real scene.

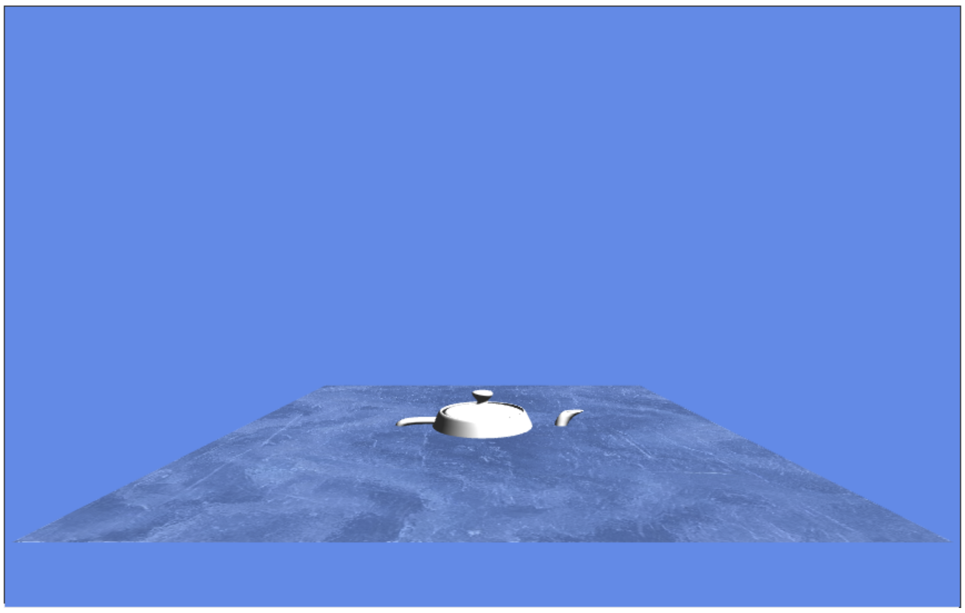

Figure 4: Oblique clipping correctly hides the submerged parts of the teapot in the reflection, creating a physically plausible intersection.

3 Implementation Details

3.1 WebGPU Architecture

The implementation leverages WebGPU's explicit pipeline state management. The application uses a `depth24plus-stencil8` format texture to support combined depth testing and stencil masking.

Texture Management: A specific challenge encountered was correctly mapping the ground texture (`xamp23.png`). WebGPU requires explicit usage flags. The texture was created with:

```
usage: GPUTextureUsage.TEXTURE_BINDING |
       GPUTextureUsage.COPY_DST |
       GPUTextureUsage.RENDER_ATTACHMENT
```

The `RENDER_ATTACHMENT` flag was critical for consistency with earlier implementation stages, ensuring the browser allocated the resource compatibly.

3.2 Render Pass Organization

To accommodate the depth buffer clear required by Oblique Clipping (Part 4), the frame is split into two distinct `GPUCommandEncoder` render passes:

Pass 1 (Reflection Generation):

- **Targets:** Clears Color (Cornflower Blue) and Depth/Stencil.
- **Pipeline A (Mask):** Renders ground to Stencil only.
- **Pipeline B (Reflect):** Renders reflected teapot using Oblique Projection P'. Stencil test enabled.

Pass 2 (Scene Composition):

- **Targets:** Loads Color (preserves reflection), **Clears Depth**.
- **Pipeline C (Ground):** Renders textured ground with blending.
- **Pipeline D (Real):** Renders real teapot using Standard Projection P.

4 Results Discussion

The final application successfully runs at 60 FPS. The visual results (Fig. 4) demonstrate that all artifacts have been resolved:

- The reflection is strictly contained within the quad (Stencil success).
- The "underground" portion of the jumping teapot is invisible in the reflection (Oblique Clipping success).
- The texture is correctly mapped and visible through the reflection (Blending success).

Challenges: The primary engineering difficulty was the handling of the Depth buffer. Oblique clipping modifies the depth distribution, making it impossible to depth-test the real teapot against the reflected teapot using a single depth buffer. Splitting the render pass allowed us to reset the depth state, ensuring correct occlusion for both the virtual (reflected) and real worlds.

Limitations: This technique is limited to planar surfaces. Curved reflectors would require environment mapping (cube maps) or ray tracing. Additionally, since the reflection is a geometric copy, the scene complexity effectively doubles, which could impact performance in scenes with high polygon counts.

Future Work: Extensions could include adding Fresnel terms to the blend function (making the reflection stronger at glancing angles) or implementing screen-space blur to simulate rougher surfaces like brushed metal.

Bibliography

References

[1] McReynolds, T. and Blythe, D. *Advanced Graphics Programming Using OpenGL*. Morgan Kaufmann, 2005. (Planar Reflectors: Ch. 17.1)

[2] Kilgard, M. J. *Improving Reflections via the Stencil Buffer*. NVIDIA Corporation.

[3] Lengyel, E. "Oblique View Frustum Depth Projection and Clipping". *Journal of Game Development*, Vol. 1, No. 2, 2005.

[4] Angel, E. and Shreiner, D. *Interactive Computer Graphics: A Top-Down Approach with WebGL*. 7th ed., Pearson, 2015.

[5] Khronos Group / W3C. WebGPU Specification. https://www.w3.org/TR/webgpu/

Use of Generative AI

I used Gemini (Google) to help structure the report, understand the underlying graphics techniques (blending, stencil masking, oblique clipping), and to assist with implementation-level reasoning specific to WebGPU. I reviewed and adapted all suggestions, tested the code locally, and ensured that the final implementation and report text reflect my own understanding and results.

Contribution

This project was completed individually. No collaborators contributed to the code or report.