CS 405 Project 2

İde Melis Yılmaz - 32400

December 13, 2025

1 Introduction

In this Project i implemented 3 Non-Photorealistic Rendering (NPR) techniques: **Toon/Cel Shading** (cartoon-style with discrete shading bands), **Edge Detection** (silhouette extraction using screen-space derivatives), and **Cross-Hatching** (classical pen-and-ink illustration). The implementation uses WebGL2's programmable pipeline with a custom G-buffer-based multi-pass architecture for real-time rendering.

2 Artistic Inspiration: Albrecht Dürer's *Melencolia I*

For the cross-hatching technique, i took inspiration from Albrecht Dürer's *Melencolia I* engraving. Dürer used systematic layering: light areas use single-direction lines, mid-tones add perpendicular cross-hatching, and dark areas accumulate multiple diagonal layers with uniform spacing.

Figure 1: Albrecht Dürer, *Melencolia I* (1514). Progressive cross-hatching for shading.

I implemented this by adding more line layers as areas get darker. When the tone passes certain thresholds, new hatching directions are added, which creates that gradual darkening effect.

3 Comparative Analysis

3.1 Visual Expressiveness

Each technique has its own artistic style and works best for different purposes:

Table 1: Expressive Characteristics of NPR Techniques

Technique	**Artistic Style**	**Best For**
Phong	Photorealistic baseline	Technical accuracy
Toon	Cartoon/anime cel	Games, animation
Edges	Technical illustration	Form clarity
Hatching	Classical engraving	Artistic prints

3.2 Performance and Limitations

All techniques run at 60+ FPS on my laptop GPU (NVIDIA GTX 1650) at 1920×1080. I optimized edge detection with smart sampling (11 texture reads instead of 18), and cross-hatching is purely mathematical so it doesn't need any textures.

Limitations: Toon band boundaries can be jagged; edges may flicker with camera movement due to RGBA8 precision; hatching uses fixed angles that distort on curved surfaces.

4 Technical Implementation

I implemented 3 NPR techniques using a custom two-pass G-buffer pipeline for screen-space effects with geometric information access.

4.1 Technique 1: Toon/Cel Shading

4.1.1 Algorithm

Toon shading is the simplest NPR technique i implemented. It takes normal continuous lighting and quantizes it into discrete bands to create that cartoon-like appearance. I implemented this in view space to keep the math simpler:

Algorithm 1 Toon Shading

$\mathbf{N} \leftarrow$ normalized view-space normal
$\mathbf{L} \leftarrow$ normalized view-space light direction
$\text{diffuse\_raw} \leftarrow \max(\mathbf{N} \cdot \mathbf{L}, 0)$
$\text{bands} \leftarrow$ user-defined band count (2–8)
$\text{diffuse\_quantized} \leftarrow \lfloor \text{diffuse\_raw} \times \text{bands} \rfloor / \text{bands}$

$\mathbf{H} \leftarrow \text{normalize}(\mathbf{L} + \mathbf{V})$
$\text{specular\_raw} \leftarrow \max(\mathbf{H} \cdot \mathbf{N}, 0)^{32}$
$\text{specular\_quantized} \leftarrow \text{step}(0.5, \text{specular\_raw})$

$\text{color}_{\text{final}} \leftarrow \text{baseColor} \times (0.25 + \text{diffuse\_quantized})$
$\quad + \text{lightColor} \times \text{specular\_quantized} \times 0.25$

The floor function creates sharp boundaries between light and shadow, while the step function makes specular either fully on or off (no gradients). I added a slider to control band count from 2 to 8 so the cartoon effect can be adjusted.

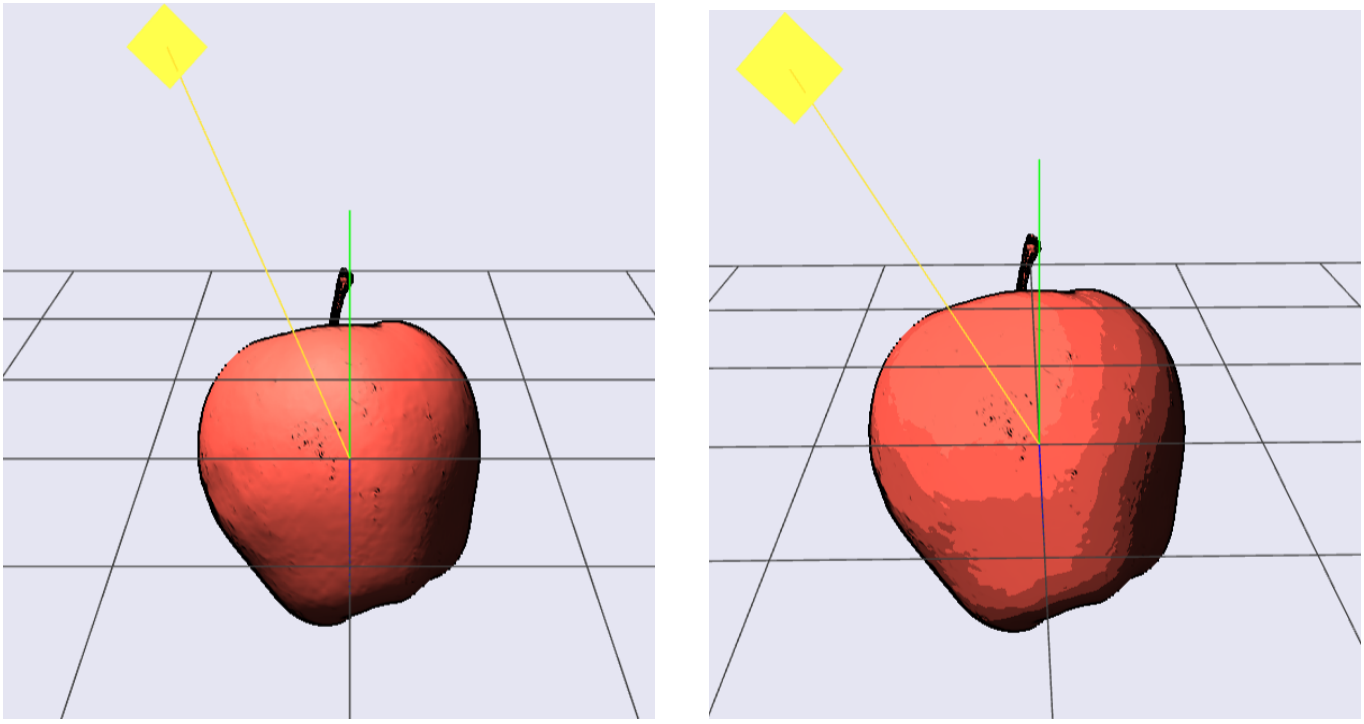
4.1.2 Implementation

Listing 1: Toon Shading (GLSL excerpt)

```
float bands = u_bands;
float diffuse = max(dot(N, L), 0.0);
float quantized = floor(diffuse * bands) / bands;

vec3 H = normalize(L + V);
float spec = pow(max(dot(N, H), 0.0), 32.0);
float specStep = step(0.5, spec);

vec3 result = baseColor * (0.25 + quantized)
            + lightColor * specStep * 0.25;
```



(a) Reference Phong (b) Toon Shading (4 bands)

Figure 2: Comparison: Continuous vs. Quantized Lighting

4.2 Technique 2: Edge Detection

4.2.1 Algorithm

For edge detection, i use Sobel operators on the G-buffer data. I detect edges from two different sources: depth discontinuities (where objects separate) and normal discontinuities (where surfaces crease). Combining both gives robust edge lines:

Algorithm 2 Sobel-Based Edge Detection

Input: G-buffer textures (normal+depth)
Output: Edge mask

for each pixel (x, y) **do**
    Sample 3×3 neighborhood: $\{d_{-1,-1}, d_{-1,0}, \ldots, d_{1,1}\}$ (depth)
    Sample 3×3 neighborhood: $\{\mathbf{n}_{-1,-1}, \mathbf{n}_{-1,0}, \ldots, \mathbf{n}_{1,1}\}$ (normals)

    // Depth gradient (Sobel X/Y)
    $\nabla_x d \leftarrow \sum_i d_i \times G_x[i]$
    $\nabla_y d \leftarrow \sum_i d_i \times G_y[i]$
    $\text{edge}_{\text{depth}} \leftarrow \sqrt{(\nabla_x d)^2 + (\nabla_y d)^2}$

    // Normal gradient
    $\nabla_x \mathbf{n} \leftarrow \sum_i \mathbf{n}_i \times G_x[i]$
    $\nabla_y \mathbf{n} \leftarrow \sum_i \mathbf{n}_i \times G_y[i]$
    $\text{edge}_{\text{normal}} \leftarrow |\nabla_x \mathbf{n}| + |\nabla_y \mathbf{n}|$

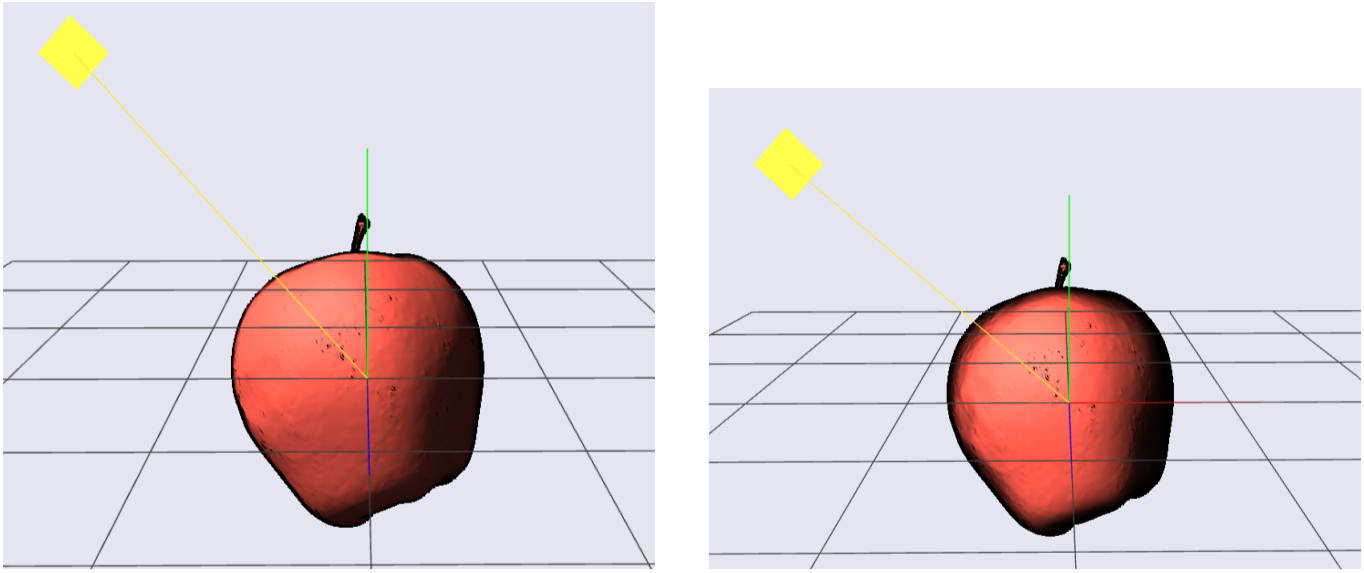
    // Combined edge strength
    $\text{edge} \leftarrow w_d \times \text{edge}_{\text{depth}} + w_n \times \text{edge}_{\text{normal}}$

    if edge > threshold **then**
        Output black line
    else
        Output original color
    end if
end for

Sobel Kernels:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \tag{1}$$

I used both depth and normals because depth edges catch the outer silhouette and overlapping objects, while normal edges detect surface details like creases and sharp corners. Combining them gives complete edge detection without extra geometry processing.



(a) Low edge width (0.1)

(b) High edge width (0.6)

Figure 3: Edge Detection: Adjustable silhouette width using rim threshold

4.3 Technique 3: Cross-Hatching (My Extension)

4.3.1 How I Extended It: Progressive Layer System

This is the technique that includes my extension. Instead of using pre-made textures like most hatching implementations do, i created a fully procedural system that generates lines mathematically. I designed it to follow Dürer's progressive layering approach where darker areas accumulate more and more line layers:

Algorithm 3 Progressive Cross-Hatching

tone $\leftarrow 1 - \max(\mathbf{N} \cdot \mathbf{L}, 0)$ ▷ 0 = light, 1 = dark
uv $\leftarrow$ texture coordinates $\times$ hatchScale
lineWidth $\leftarrow 0.5$ ▷ Relative to period

// Layer 1: Horizontal lines (0°)
$h_1 \leftarrow \text{step}(\text{lineWidth}, \text{fract}(\mathbf{uv}.y \times 15))$

// Layer 2: Vertical lines (90°)
$h_2 \leftarrow \text{step}(\text{lineWidth}, \text{fract}(\mathbf{uv}.x \times 15))$

// Layer 3: Diagonal lines (45°)
$\mathbf{uv}_{\text{diag1}} \leftarrow \text{rotate}(\mathbf{uv}, 45°)$
$h_3 \leftarrow \text{step}(\text{lineWidth}, \text{fract}(\mathbf{uv}_{\text{diag1}}.y \times 15))$

// Layer 4: Counter-diagonal (-45°)
$\mathbf{uv}_{\text{diag2}} \leftarrow \text{rotate}(\mathbf{uv}, -45°)$
$h_4 \leftarrow \text{step}(\text{lineWidth}, \text{fract}(\mathbf{uv}_{\text{diag2}}.y \times 15))$

// Layers 5 & 6: Denser lines for very dark areas
$h_5 \leftarrow \text{step}(0.35, \text{fract}(\mathbf{uv}.y \times 22))$
$h_6 \leftarrow \text{step}(0.35, \text{fract}(\mathbf{uv}_{\text{diag1}}.y \times 22))$

// Progressive accumulation
hatch $\leftarrow 1.0$ ▷ Start with white (no lines)
if tone > 0.12 **then**
    hatch $\leftarrow h_1$ ▷ Single direction
end if
if tone > 0.30 **then**
    hatch $\leftarrow \min(\text{hatch}, h_2)$ ▷ Cross-hatch begins
end if
if tone > 0.45 **then**
    hatch $\leftarrow \min(\text{hatch}, h_3)$ ▷ Add diagonal
end if
if tone > 0.60 **then**
    hatch $\leftarrow \min(\text{hatch}, h_4)$ ▷ Full cross-hatch
end if
if tone > 0.75 **then**
    hatch $\leftarrow \min(\text{hatch}, h_5)$ ▷ Increase density
end if
if tone > 0.88 **then**
    hatch $\leftarrow \min(\text{hatch}, h_6)$ ▷ Maximum darkness
end if

$\text{color}_{\text{lit}} \leftarrow \text{baseColor} \times (0.3 + \mathbf{N} \cdot \mathbf{L} \times 0.7)$
$\text{color}_{\text{ink}} \leftarrow \text{baseColor} \times 0.15$ ▷ Dark ink
return $\text{mix}(\text{color}_{\text{ink}}, \text{color}_{\text{lit}}, \text{hatch})$

Each line layer is generated by rotating the UV coordinates and using `fract(uv.y * frequency)` to create periodic patterns. The `step` function turns this into black/white lines. I rotate the UVs by 0°, 45°, 90°, and -45°for different directions. The `min` operation accumulates layers - wherever lines overlap, the result gets darker.

I implemented 6 layers (most tutorials only do 2-3) with tone thresholds at 0.12, 0.30, 0.45, 0.60, 0.75, and 0.88. This gives smoother darkening like Dürer's engravings. Everything is procedural so there's no texture memory cost.

Listing 2: Cross-Hatching

```
float tone = 1.0 - clamp(dot(N, L), 0.0, 1.0);
vec2 uv = v_uv * u_hatchScale;

// Generate layers with rotation
float h1 = step(0.5, fract(uv.y * 15.0));
float h2 = step(0.5, fract(uv.x * 15.0));
vec2 d1 = mat2(0.707,-0.707,0.707,0.707) * uv;
float h3 = step(0.5, fract(d1.y * 15.0));
vec2 d2 = mat2(0.707,0.707,-0.707,0.707) * uv;
```

```
float h4 = step(0.5, fract(d2.y * 15.0));

// Accumulate based on tone
float hatch = 1.0;
if (tone > 0.12) hatch = h1;
if (tone > 0.30) hatch = min(hatch, h2);
if (tone > 0.45) hatch = min(hatch, h3);
if (tone > 0.60) hatch = min(hatch, h4);

vec3 lit = baseColor * (0.3 + ndotl * 0.7);
vec3 ink = baseColor * 0.15;
vec3 result = mix(ink, lit, hatch);
```

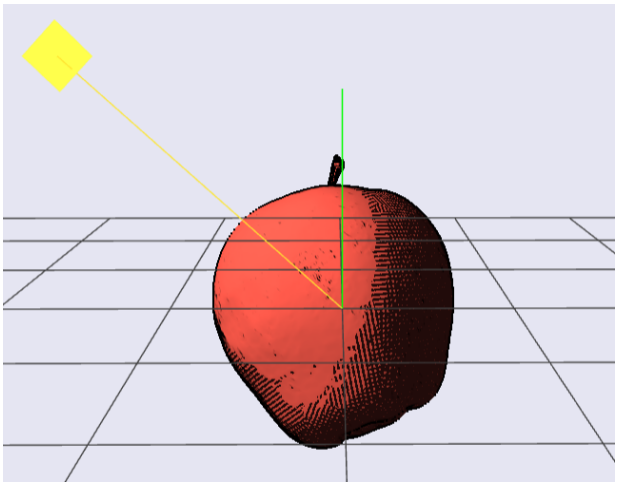


Figure 4: Cross-Hatching: Progressive layer accumulation inspired by Dürer's technique

5 Conclusion

This project taught me a lot about GPU programming and artistic rendering. I implemented 3 NPR techniques that all run at 60+ FPS on my GTX 1650.

The most interesting part was the cross-hatching extension. Instead of using textures like most implementations, i made it fully procedural with 6 layers. This means zero memory cost, infinite resolution, and real-time parameter control.

The G-buffer pipeline made it easy to experiment with different effects. The two-pass setup gives me screen-space edge detection while keeping geometric data for proper shading.

The hardest part was tuning the tone thresholds (0.12, 0.30, 0.45, 0.60, 0.75, 0.88) for the cross-hatching. I wanted gradual darkening like Dürer's engravings and i found suitable thresholds with trying.