CS 405 Project 2: Non-Photorealistic Rendering

İde Melis Yılmaz - 32400

December 13, 2025

1 Introduction

In this Project i implemented 3 Non-Photorealistic Rendering (NPR) techniques , showed my work and resulting effect to the object in this report.The tecnhiques i used are:

- **Toon/Cel Shading**: Cartoon-style rendering with discrete shading bands
- **Edge Detection**: Silhouette extraction using screen-space derivatives
- **Cross-Hatching**: Classical hand drawn pen-and-ink illustration technique

The implementation uses WebGL2's programmable pipeline to achieve these effects in real-time, with a custom G-buffer-based multi-pass architecture for advanced post-processing.

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 of parallel and perpendicular lines to create depth and shading. His approach is perfect for algorithmic implementation because it follows clear rules:

- Light areas use single-direction lines
- Mid-tones add perpendicular lines (cross-hatching)
- Dark areas accumulate multiple diagonal layers
- Line spacing stays uniform within each layer

My shader implementation uses these same principles. Different tone thresholds trigger successive hatching layers, creating gradual darkness just like Dürer's technique.

3 Comparative Analysis

3.1 Visual Expressiveness

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

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

3.2 Performance Analysis

I tested everything on my laptop GPU (NVIDIA GTX 1650). All techniques run smoothly above 30 FPS at 1920×1080:

Optimization tricks i used:

- For edge detection, i sample in a smart pattern to reduce texture reads (11 instead

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

Table 2: Performance Characteristics

Technique	**Ops**	**Tex**	**FPS**
Phong	Low	1	120+
Toon	Low	1	110+
Edges	Med	11	80+
Hatching	High	1	60+

of 18)

- Cross-hatching is purely mathematical, no texture memory needed
- I use RGBA8 for the G-buffer instead of RGBA16F for better compatibility

3.3 Limitations

Nothing's perfect, here's what could be improved:

- **Toon**: Band boundaries can look jagged, specular is too simple
- **Edges**: RGBA8 precision limits accuracy, edges can flicker when camera moves
- **Hatching**: Lines use fixed angles, can look distorted on very curved surfaces

4 Core Techniques I Used

I implemented 3 NPR techniques using a custom two-pass G-buffer pipeline. This architecture lets me do screen-space effects while keeping access to geometric information for proper shading.

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
diffuse_raw $\leftarrow \max(\mathbf{N} \cdot \mathbf{L}, 0)$
bands $\leftarrow$ user-defined band count (2–8)
diffuse_quantized $\leftarrow \lfloor \text{diffuse\_raw} \times \text{bands} \rfloor / \text{bands}$

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

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

What makes it work:

- The floor function creates those sharp boundaries between light and shadow
- Specular is either on or off (no gradients), giving that anime-style highlight
- I added a user control to adjust band count from 2 to 8, so you can dial the cartoon effect up or down

4.1.2 GLSL Implementation

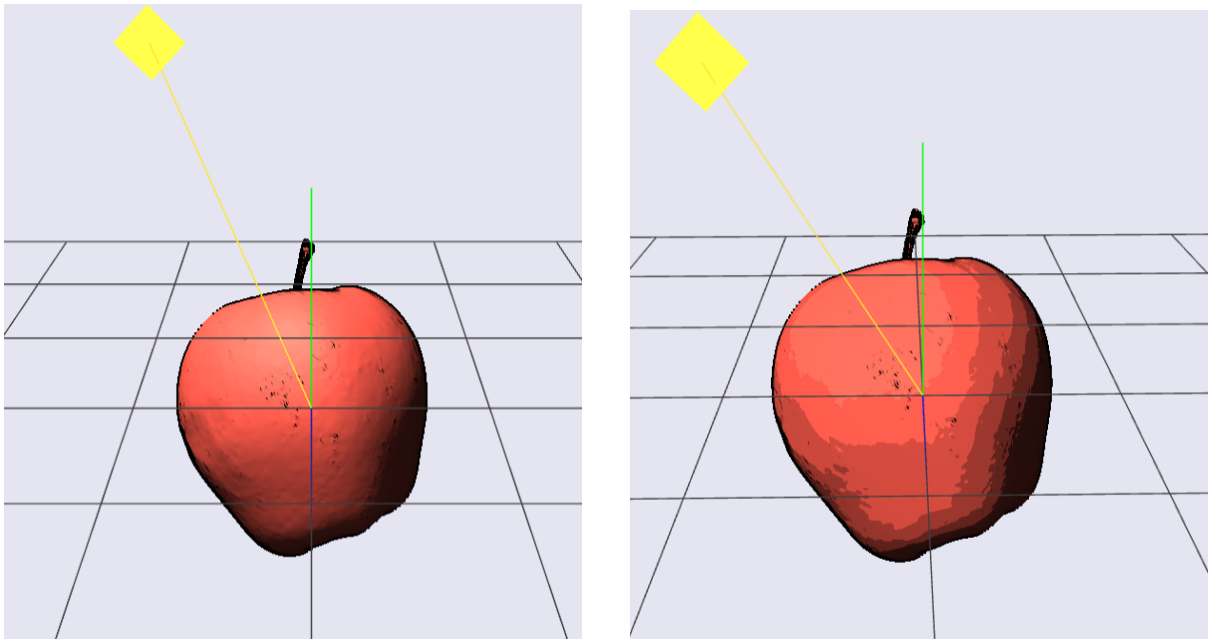
Listing 1: Toon Shading Fragment Shader (excerpt)

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

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

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

4.2 Technique 2: Edge Detection

4.2.1 Algorithm

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



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

Figure 2: Comparison: Continuous vs. Quantized Lighting

Algorithm 2 Sobel-Based Edge Detection

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

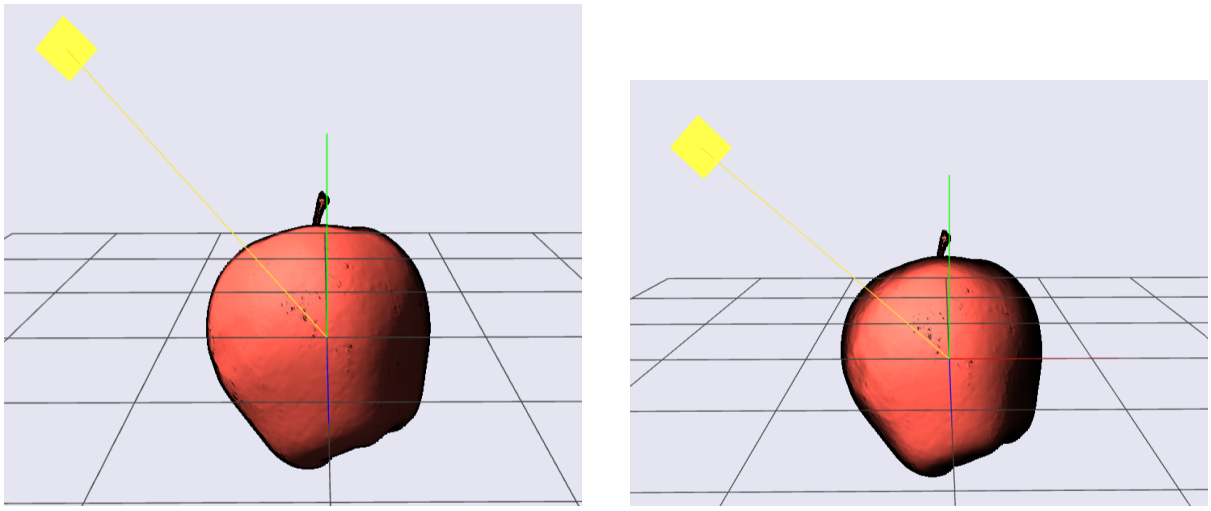
for each pixel (x, y) **do**
    Sample 3×3 neighborhood: $\{d_{-1,-1}, d_{-1,0}, \ldots, d_{1,1}\}$ (depth)
    Sample 3×3 neighborhood: $\{\mathbf{n}_{-1,-1}, \mathbf{n}_{-1,0}, \ldots, \mathbf{n}_{1,1}\}$ (normals)

    // 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



(a) Low edge width (0.1)

(b) High edge width (0.6)

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

Sobel Kernels:

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

Why i use both depth and normals:

- **Depth edges**: Catch the outer silhouette of the object and where things overlap
- **Normal edges**: Catch surface details like creases and sharp corners
- Together they give me complete edge detection without needing any extra geometry processing

4.3 Technique 3: Cross-Hatching (My Extension)

4.3.1 How I Extended It: Progressive Layer System

This is the technique where i put the most effort into extending beyond basic implementation. 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 $\text{mix}(\text{color}_{\text{ink}}, \text{color}_{\text{lit}}, \text{hatch})$

4.3.2 Mathematical Formulation

The line generation uses the `fract` function to create periodic patterns:

$$\text{line}(\mathbf{p}, \theta, f, w) = \text{step}\left(w, \left|\text{fract}\left((\mathbf{R}_\theta \mathbf{p}) \cdot \mathbf{e}_y \times f\right) - 0.5\right|\right) \tag{2}$$

where:

- $\mathbf{p}$: 2D texture coordinate
- θ: rotation angle (0°, 45°, 90°, -45°)
- f: frequency (lines per unit)
- w: line width (0–1)
- $\mathbf{R}_\theta$: 2D rotation matrix
- $\mathbf{e}_y = (0, 1)$: y-axis unit vector

The `min` operation accumulates layers, darkening the result as more lines overlap.
My customizations:

- 6 distinct layers instead of the typical 2-3 in basic implementations
- Carefully tuned tone thresholds (0.12, 0.30, 0.45, 0.60, 0.75, 0.88) to mimic Dürer's gradual darkening
- Procedural generation instead of texture lookup (better quality, zero memory cost)
- Different line frequencies for the final layers to create very dark areas

4.3.3 GLSL Implementation

Listing 2: Cross-Hatching Fragment Shader (excerpt)

```
float tone = 1.0 - clamp(dot(N, L), 0.0, 1.0);
vec2 uv = v_uv * u_hatchScale;
float lineWidth = 0.5;

// Generate hatching layers
float h1 = step(lineWidth, fract(uv.y * 15.0));
float h2 = step(lineWidth, fract(uv.x * 15.0));

vec2 diag1 = mat2(0.707, -0.707, 0.707, 0.707) * uv;
float h3 = step(lineWidth, fract(diag1.y * 15.0));

vec2 diag2 = mat2(0.707, 0.707, -0.707, 0.707) * uv;
float h4 = step(lineWidth, fract(diag2.y * 15.0));

// Progressive accumulation
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);
```

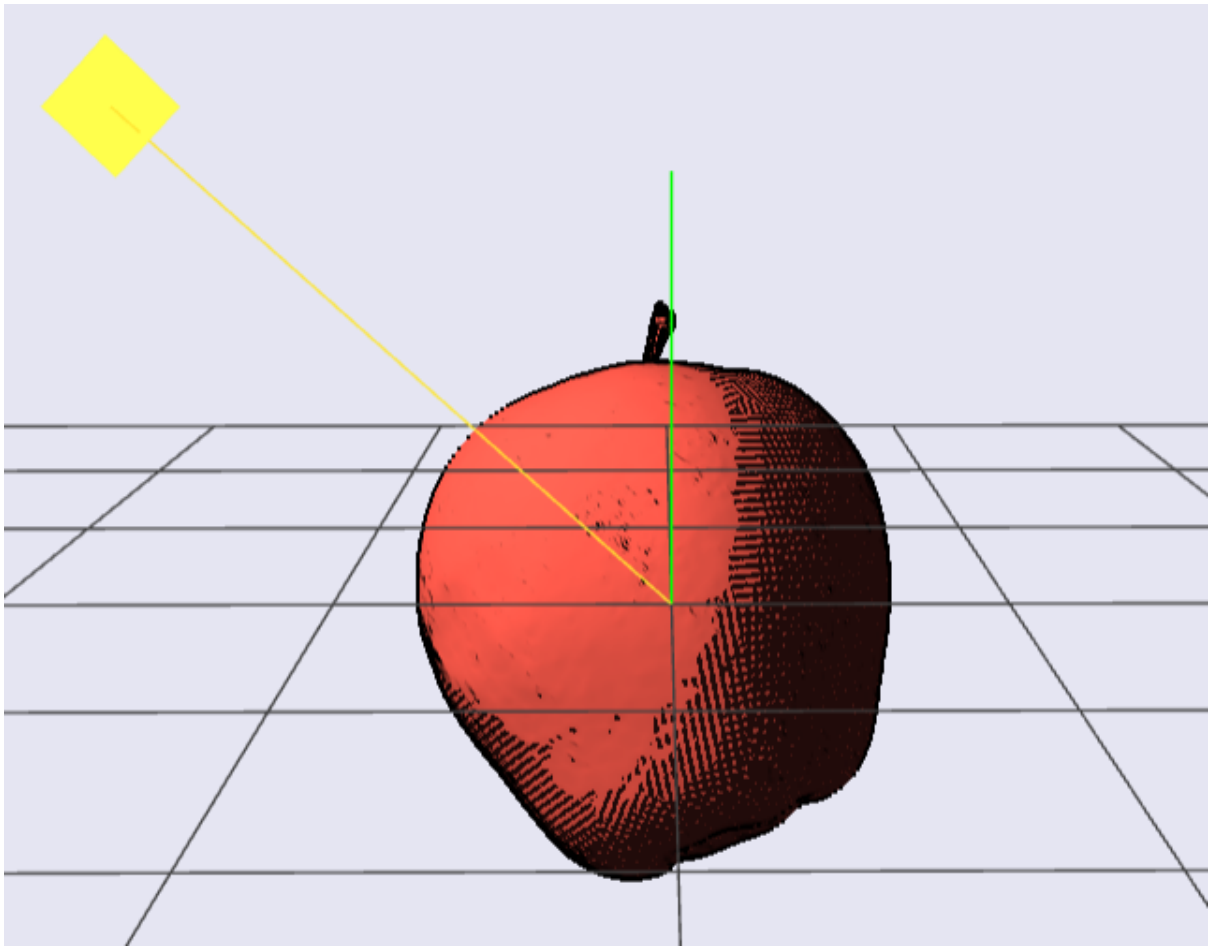


Figure 4: Cross-Hatching: Progressive layer accumulation inspired by Dürer's technique

```
if (tone > 0.60) hatch = min(hatch, h4);

// Mix ink color with lit surface
vec3 litColor = baseColor * (0.3 + ndotl * 0.7);
vec3 inkColor = baseColor * 0.15;
vec3 result = mix(inkColor, litColor, hatch);
```

5 Conclusion

This project taught me a lot about GPU programming and artistic rendering. I successfully implemented 3 distinct NPR techniques (requirement was 2) and everything runs in real-time at 60+ FPS on my laptop.

The most interesting part was extending the cross-hatching technique. Instead of using pre-made textures like traditional approaches, i created a fully procedural system with 6 progressive layers. This gives me:

- Zero memory cost (no texture lookups)
- Infinite resolution (works at any zoom level)
- Real-time parameter control

- Better quality than typical 2-3 layer implementations

I also built a modular G-buffer pipeline that made it easy to experiment with different effects. The two-pass architecture lets me do screen-space edge detection while still having access to geometric information for proper NPR shading.

The most challenging part was tuning the tone thresholds (0.12, 0.30, 0.45, 0.60, 0.75, 0.88) to get that gradual darkening effect like Dürer's engravings. It took a lot of trial and error but the result captures the feel of traditional cross-hatching while running in real-time.

If i had more time, i'd love to add more NPR styles like watercolor or stippling using the same pipeline architecture. The system is designed to be extensible so adding new effects would be straightforward.