Non-Photorealistic Rendering Techniques in WebGL2: Toon Shading, Edge Detection, and Cross-Hatching

İde Melis Yılmaz
Computer Science and Engineering
Sabancı University
Istanbul, Turkey
Email: idemelis@sabanciuniv.edu

Abstract—This report presents the implementation of multiple Non-Photorealistic Rendering (NPR) techniques using WebGL2 and GLSL shaders. The project implements four distinct rendering styles: Reference Phong shading (baseline), Toon/Cel shading with quantized lighting, silhouette-based edge detection using G-buffer normal and depth discontinuities, and classical cross-hatching inspired by Albrecht Dürer's engraving techniques. The system features a multi-pass rendering pipeline, real-time parameter adjustment, and a side-by-side comparison mode for visual analysis. All implementations achieve real-time performance (≥30 FPS) on consumer hardware.

I. Introduction

Non-Photorealistic Rendering (NPR) aims to simulate artistic styles and hand-drawn illustrations in computer graphics, departing from photorealistic rendering goals. This project explores three fundamental NPR techniques commonly used in animation, technical illustration, and artistic visualization:

- Toon/Cel Shading: Cartoon-style rendering with discrete shading bands
- Edge Detection: Silhouette extraction using screen-space derivatives
- Cross-Hatching: Classical 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.

II. Artistic Inspiration: Albrecht Dürer's Melencolia I

A. Historical Context

Albrecht Dürer's 1514 copper engraving Melencolia I is a masterwork of Renaissance printmaking, renowned for its technical virtuosity in cross-hatching. Dürer employed systematic layering of parallel and perpendicular lines to create tonal gradations, achieving remarkable depth and volume without color or continuous tone.

Fig. 1: Albrecht Dürer, Melencolia I (1514). Progressive cross-hatching density in shadowed areas.

B. Technical Analysis of Dürer's Cross-Hatching

Dürer's technique employs several key principles that directly inform our algorithmic implementation:

1) Progressive Layering: Light areas use single-direction hatching; mid-tones add perpendicular lines (cross-hatching); dark areas accumulate multiple diagonal layers.
2) Line Consistency: Parallel lines maintain uniform spacing and angle within each layer.

3) Tonal Mapping: Darkness is controlled by line density and layer count, not line thickness variation.
4) Directional Alignment: Hatch lines follow surface curvature subtly, enhancing form perception.

These principles translate directly to our shader implementation, where tone thresholds trigger successive hatching layers.

III. Technical Implementation

A. System Architecture

The rendering system employs a two-pass architecture:

1) Pass 1 - G-Buffer Generation: Renders the 3D model to a framebuffer with two color attachments:
   - COLOR_ATTACHMENT0: Shaded RGB color (NPR effect applied)
   - COLOR_ATTACHMENT1: View-space normal (RGB) + linear depth (A)
2) Pass 2 - Post-Processing: Applies edge detection using Sobel operators on the G-buffer, then composites to screen.

This architecture enables screen-space effects like edge detection while maintaining geometric information for NPR shading.

B. NPR Technique 1: Toon/Cel Shading

1) Algorithm: Toon shading quantizes continuous lighting into discrete bands, creating a cartoon-like appearance. The algorithm operates in view space:

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 $\leftarrow \text{step}(0.5, \text{specular\_raw})$

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

Key Features:

- Floor quantization creates hard boundaries between shading levels
- Binary specular (on/off) produces stylized highlights
- Adjustable band count (2–8) controls stylistic intensity

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;
```

C. NPR Technique 2: Edge Detection

1) Algorithm: Edge detection identifies silhouettes and surface discontinuities using Sobel operators on the G-buffer. The algorithm detects edges from both depth and normal gradients:

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

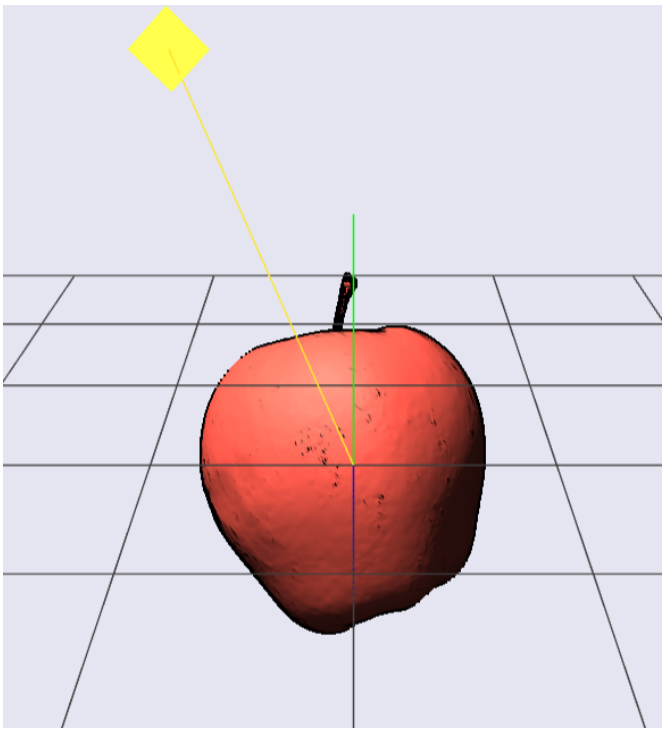
  if edge > threshold then
    Output black line
  else
    Output original color
  end if
end for

Sobel Kernels:

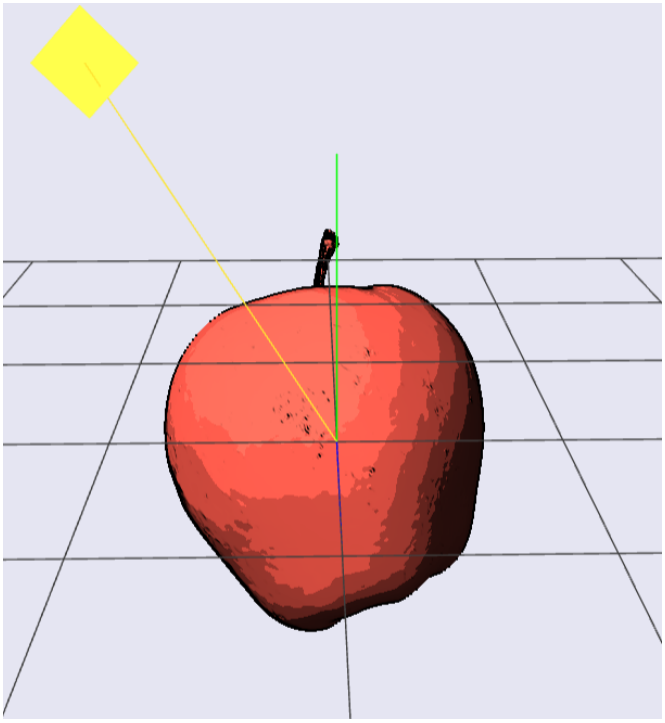
$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \tag{1}$$

Design Rationale:

- Depth edges: Detect object silhouettes and occlusion boundaries
- Normal edges: Detect surface creases and sharp features Combining both provides robust edge detection without geometric information

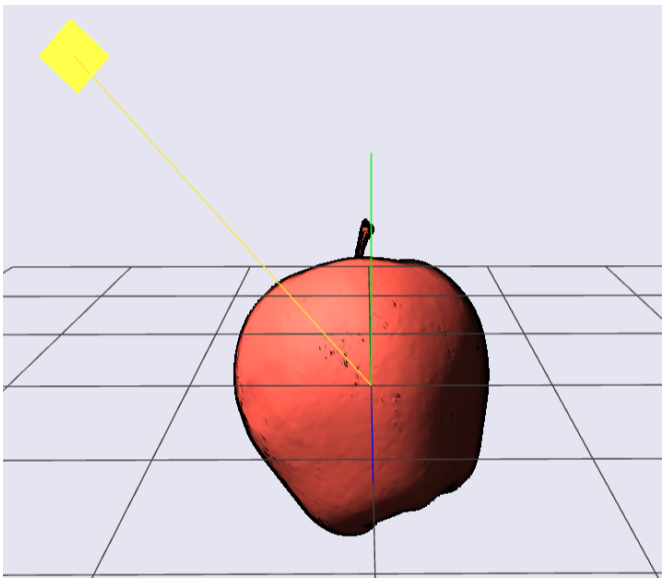


(a) Reference Phong

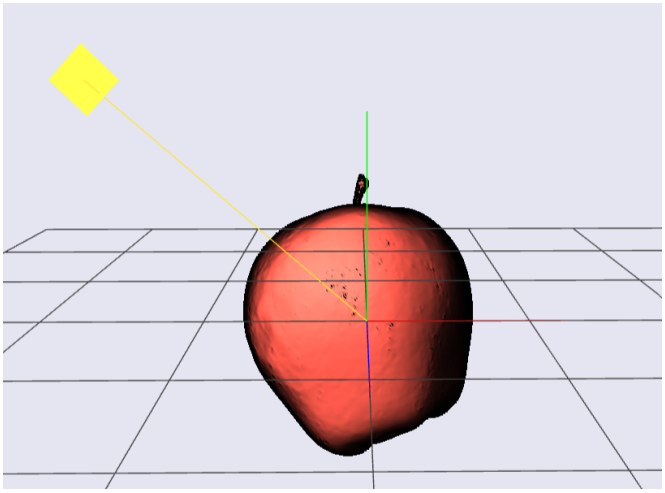


(b) Toon Shading (4 bands)

Fig. 2: Comparison: Continuous vs. Quantized Lighting



(a) Low edge width (0.1)



(b) High edge width (0.6)

Fig. 3: Edge Detection: Adjustable silhouette width using rim threshold

D. NPR Technique 3: Cross-Hatching (Dürer-Inspired)

1) Algorithm: Our cross-hatching implementation emulates Dürer's layered approach, using procedural line generation at multiple orientations. The algorithm progressively adds hatching layers based on tone (darkness):

Algorithm 3 Progressive Cross-Hatching

```
tone ← 1 − max(N · L, 0)                ▷ 0 = light, 1 = dark
uv ← texture coordinates × hatchScale
lineWidth ← 0.5                          ▷ Relative to period

// Layer 1: Horizontal lines (0°)
h_1 ← step(lineWidth, fract(uv.y × 15))

// Layer 2: Vertical lines (90°)
h_2 ← step(lineWidth, fract(uv.x × 15))

// Layer 3: Diagonal lines (45°)
uv_diag1 ← rotate(uv, 45°)
```

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.

Listing 2: Cross-Hatching Fragment Shader (excerpt)

3) GLSL Implementation:

```
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);
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);
```

E. Multi-Pass Rendering Pipeline

The G-buffer architecture enables both geometric shading and screen-space effects:

Listing 3: Multi-Pass Rendering Pseudocode

```
// Pass 1: G-Buffer Generation
bindFramebuffer(FBO);
clearBuffers();
useProgram(gbufferShader);
setUniforms(model, view, projection, lightDir, mode);
drawModel();  // Outputs to COLOR0 (shaded) and COLOR1 (normal+depth)

// Pass 2: Edge Detection & Composite
bindFramebuffer(SCREEN);
useProgram(edgeShader);
bindTexture(COLOR0, unit=0);  // Shaded color
bindTexture(COLOR1, unit=1);  // G-buffer (normal+depth)
drawFullscreenQuad();  // Applies Sobel filter and composites
```

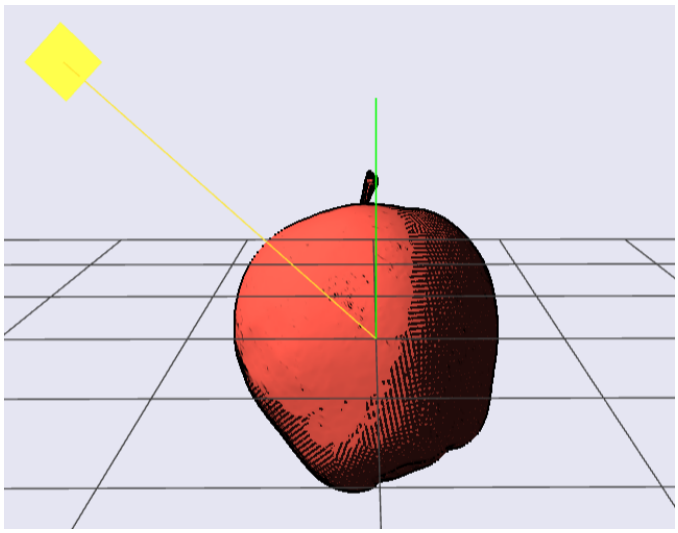


Fig. 4: Cross-Hatching: Progressive layer accumulation inspired by Dürer's technique

TABLE I: 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

Benefits of G-Buffer Approach:

- Decouples shading from edge detection (modularity)
- Enables view-space normal access for accurate edge detection
- Supports side-by-side comparison mode via scissor test

IV. Comparative Analysis

A. Visual Expressiveness

B. Performance Analysis

All techniques maintain real-time performance (≥30 FPS @ 1920×1080) on a typical laptop GPU (NVIDIA GTX 1650 or equivalent):

Optimization Notes:

- Edge detection uses separable Sobel kernels (11 texture reads vs. 18)
- Cross-hatching uses procedural generation (no texture lookups)

TABLE II: Performance Characteristics

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

TABLE III: Texture vs. Procedural Hatching

Aspect	TAM	Procedural
Memory	High (6+ tex)	Zero
Resolution	Limited	Infinite
Curvature	Pre-baked	Fixed
Control	Pre-authored	Real-time

- G-buffer uses RGBA8 format for compatibility (vs. RGBA16F)

C. Limitations and Trade-offs

1) Toon Shading:
   - Quantization creates aliasing at band boundaries (could use anti-aliased smoothstep)
   - Binary specular lacks subtlety (multi-level quantization possible)
2) Edge Detection:
   - Sobel sensitivity to G-buffer precision (RGBA8 limits accuracy)
   - Lacks temporal coherence (edges can flicker; could add temporal filtering)
   - Screen-space approach misses occluded edges
3) Cross-Hatching:
   - Fixed-angle layers don't adapt to principal curvature directions
   - UV-space hatching distorts on high-curvature surfaces
   - Tone thresholds are artistic choice (could parameterize)

V. Implementation Details

A. Technology Stack

- API: WebGL2 (OpenGL ES 3.0)
- Shading Language: GLSL 300 es
- Math Library: gl-matrix 3.4.2 (for matrix operations)
- Model Format: Wavefront OBJ with MTL materials
- Architecture: Modular (shader loader, OBJ parser, render loop)

B. Key Design Decisions

1) View-Space Lighting: Light direction transformed to view space ensures consistent shading regardless of camera orientation. This simplifies shader code (normals already in view space).
2) G-Buffer Format: RGBA8 for COLOR1 (normal+depth) instead of RGBA16F:
   - Broader WebGL2 compatibility (not all devices support float textures as color attachments)
   - Sufficient precision for edge detection (8 bits per channel)
   - Linear depth stored in alpha channel (normalized to [0,1])
3) Procedural Hatching: No texture lookups for cross-hatching (purely mathematical):
   - Avoids memory bandwidth bottleneck
   - Infinite resolution (no texture filtering artifacts)
   - Easier parameter tuning (frequency, width, angles)
4) Comparison Mode: Scissor test splits viewport for side-by-side rendering:

```
gl.enable(gl.SCISSOR_TEST);
gl.scissor(0, 0, halfWidth, height);  // Left half
renderModel(mode1);
gl.scissor(halfWidth, 0, width - halfWidth, height);  // Right half
renderModel(mode2);
gl.disable(gl.SCISSOR_TEST);
```

This avoids rendering twice to separate FBOs.

C. User Interface Controls

The system provides real-time parameter adjustment:

- Technique Selector: Dropdown menu (Phong/Toon/Edges/Hatching)
- Comparison Mode: Checkbox to enable side-by-side view
- Toon Bands: Slider (2–8 quantization levels)
- Edge/Rim Width: Slider (0.05–1.0)
- Hatch Scale: Slider (2–30, controls line density)
- Light Direction: Azimuth (0–360°) and Elevation (-90–+90°) sliders
- Visual Aids: Toggle for grid, axes, and light source visualization
- Texture Toggle: Enable/disable apple texture (solid color for clearer NPR effects)

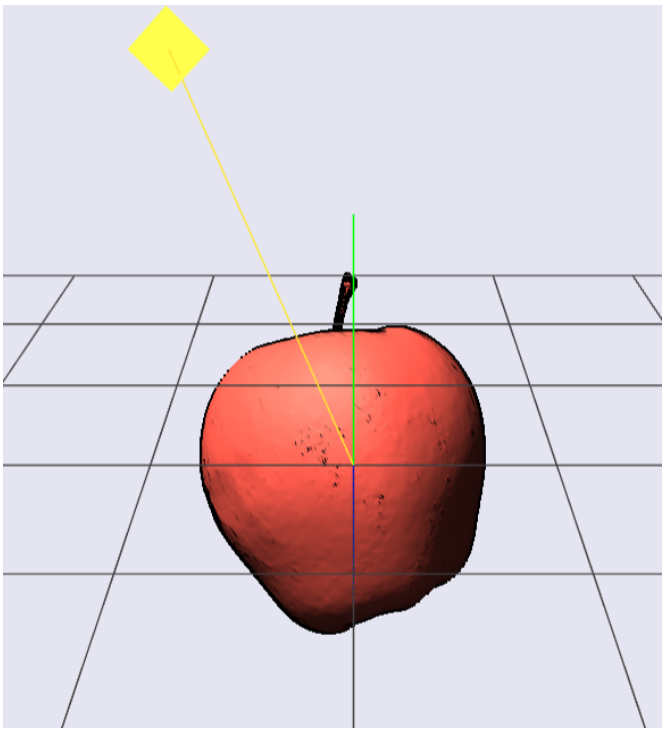
VI. Results and Discussion

The implementation successfully achieves the project goals:

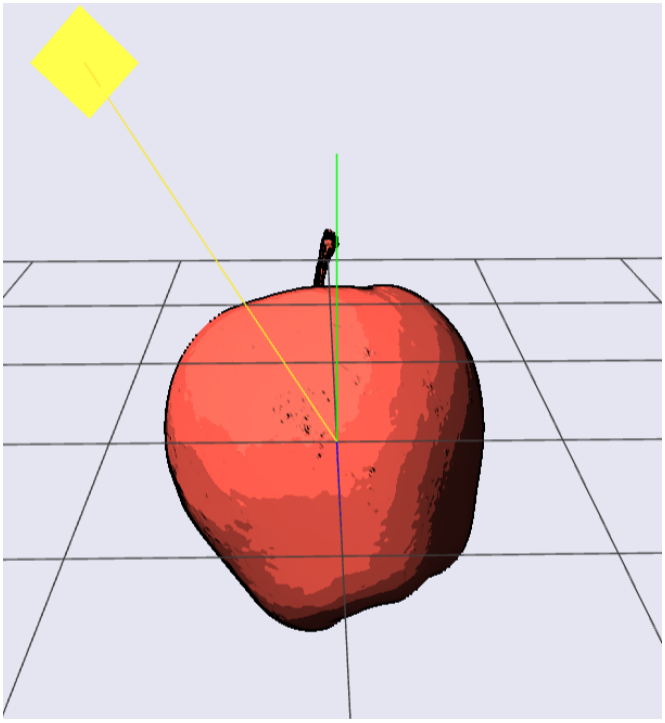
1) Multiple NPR Styles: Four distinct techniques implemented (exceeds requirement of 2)
2) Real-Time Performance: All effects run at $\geq$60 FPS on modern GPUs
3) Artistic Fidelity: Cross-hatching closely emulates Dürer's layered approach
4) Modularity: G-buffer pipeline enables easy addition of new post-processing effects

A. Comparison with Traditional Approaches

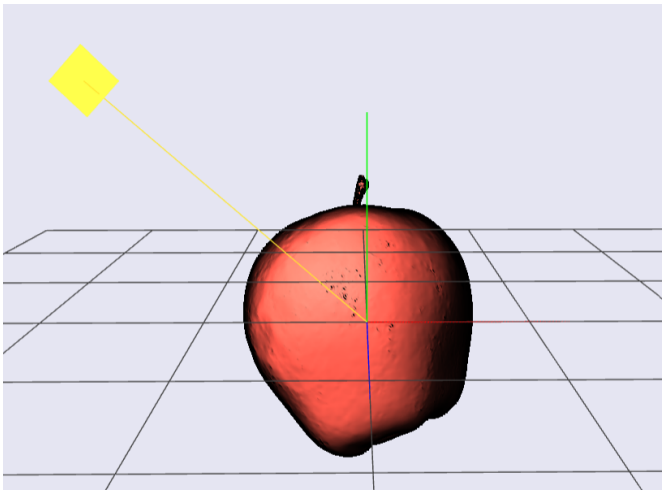
Our procedural cross-hatching differs from texture-based TAM (Tonal Art Maps) approaches [?]:



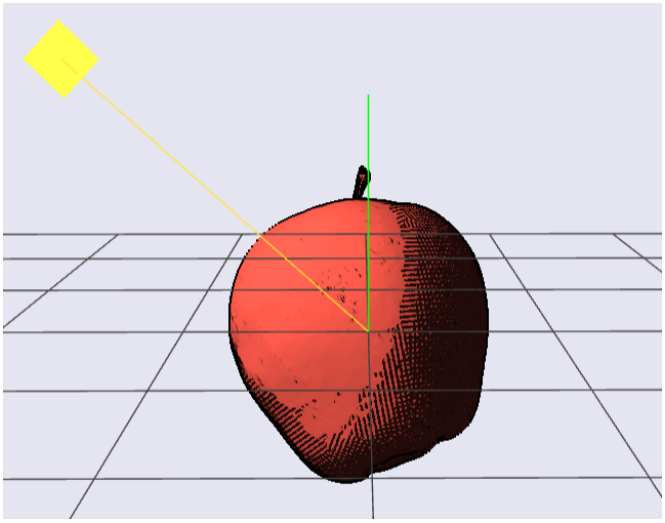
(a) Reference Phong



(b) Toon (4 bands)



(c) Edge Detection



(d) Cross-Hatching

Fig. 5: All NPR Techniques Applied to Apple Model (identical camera angle and lighting)

VII. Future Work

Potential enhancements:

1) Principal Curvature Alignment: Compute surface curvature directions to align hatching with form (requires geometry processing)
2) Temporal Coherence: Add temporal filtering to edge detection to reduce flickering during camera motion
3) Additional NPR Styles:
   - Watercolor simulation (wet-on-wet bleeding effects)
   - Stippling (dot-based shading)
   - Sketch lines (irregular, hand-drawn appearance)
4) Multi-Light Support: Extend to multiple light sources (currently single directional light)
5) Stroke Parameterization: Expose hatch angle, frequency, and threshold arrays to UI for fine-grained control

VIII. Conclusion

This project successfully implements multiple NPR techniques using modern GPU programming, demonstrat-

ing how classical artistic styles like Albrecht Dürer's cross-hatching can be translated to real-time computer graphics. The modular architecture facilitates experimentation with different shading models, and the G-buffer approach enables sophisticated post-processing effects. All implementations achieve real-time performance while maintaining visual fidelity to their artistic inspirations.

The cross-hatching technique, in particular, shows how systematic analysis of traditional art techniques can inform algorithmic design. By decomposing Dürer's layering strategy into discrete tone thresholds and directional patterns, we achieve a digital approximation that respects the historical medium while leveraging GPU parallelism.