NPR Shading using Toon Shading, Cross-Hatching and Edge Detection Algorithms

Jose Borquez[*] Arturo Yepez[†]

Lund University
Sweden

Abstract

This report navigates the convergence of the intricate dance between an artist's strokes and the precision of edge detection algorithms, particularly using the Sobel filter, a mathematical filter that allow us to obtain more defined borders when analyzing the normal and depth of a 3D scene. Additionally, the study delves into the world of toon-shading, a stylized approach inspired by comics and animation, transcending the boundaries of realism when trying to approach that style, playing with the use of grey-scales and their threshold for being draw amongside some clever tricks to get better results by using cross-hatching. Throughout, methodologies are dissected, nuances are uncovered, and the synthesis of traditional craftsmanship and state-of-the-art technology is celebrated. Join us in unraveling the tapestry of hand-drawn shading, where creativity and algorithms intertwine to elevate visual storytelling to new heights.

1 Introduction

This project is heavily inspired by the work of Youtuber **Useless Game Dev**, on *"Moebius-style 3D rendering"*, where a Unity version is implemented, trying to imitate the art style of Moebius paintings.

We delve into the confluence of artistic finesse and technological prowess, an intersection where artistry converges with rendering technology. Our focus lies on the nuanced domain of hand-drawn shading, where the eloquent strokes of artistic expression seamlessly integrate with the precision of edge detection algorithm to delimitate the space and give more of a drawn vibe, achieving it using the Sobel filter.

Using the Sobel filter alongisde the line-art shading let us do an insightful exploration into the intricate interplay of light and shadow, as we navigate the sophisticated realm of line-art shading, recognized more formally as toon shading. Together, we shall meticulously unravel the methodologies, uncover subtle intricacies, and gain an appreciation for the harmonious amalgamation of traditional craftsmanship and state-of-the-art technology in the pursuit of visual excellence.

2 Algorithm and Implementation

Our project uses a two-pass rendering algorithm. The first one applies the Toon Shading and Cross-Hatching to the scene, and fills the normal and depth buffers, which are used during the second pass for the edge detection algorithm to define the outline of every object in view. The reason to use a two-pass algorithm, is that for the edge detection, we need an already rendered view of the scene, so that we can read information from the neighbour pixels.

2.1 First Pass

2.1.1 Toon Shading

For the toon shading, we use the dot product between the light direction and the normal of the geometry, per pixel. This will give us information about which geometries are facing the light. By defining different thresholds and the corresponding output grey-scale values, we can create a toon-shading effect. Both the thresholds and the output grey-scale values are passed as `uniform float` values to the shader.

The default values for the thresholds and their corresponding output grey-scale values are

- ≥ 0.9: Specular light, pure white, 1.0
- $0.7 - 0.9$: Highlight, 0.9
- $0.5 - 0.7$: Base color, 0.7
- $0.2 - 0.5$: Shadow, 0.2
- ≤ 0.2: No lightning, pure black, 0.0

This values can be adjusted using the `GUI`.

2.1.2 Cross Hatching

During the first pass, the original diffuse texture of every 3D object is discarded, and instead, the cross-hatching texture is applied. Same as the Toon Shading, this is done by comparing the dot product between the light direction and the normal, per pixel, using the same threshold values. As we have defined 5 different grey-scale values for the toon-shading, we would need 5 different textures for the cross-hatching, however, 2 of these textures grey-scale values are pure white and pure black, therefore, no texture is needed. For the remaining 3 values, we used only one RGB texture, encoding the 3 cross-hatching textures in each of the RGB channels. This allows use to load only one texture that wraps around the object in a more natural way.

The cross-hatching texture scale can be adjusted using the `GUI`.

2.1.3 Dithering

To give a more natural look, dithering is applied to the lightning calculations. This is done by using a pseudo-random number generator using the texture coordinates as seeds. This allow us to blend between the different toon shadings and cross-hatching textures.

Both the amount of dithering and the texture scale for the dithering can be adjusted using the `GUI`.

2.2 Second Pass

From the First Pass, the Normal and Depth Buffers, as well as the rendered view, are passed to the Second Pass as textures, to be used for the Edge Detection and Final Render.

2.2.1 Edge Detection

For the Edge Detection, we used Sobel Operators (Or Sobel Filters), which involves estimating the edges present in an image by using the following Kernels:

[*]e-mail: jo0464bo-s@student.lu.se
[†]e-mail: ar8732ye-s@student.lu.se

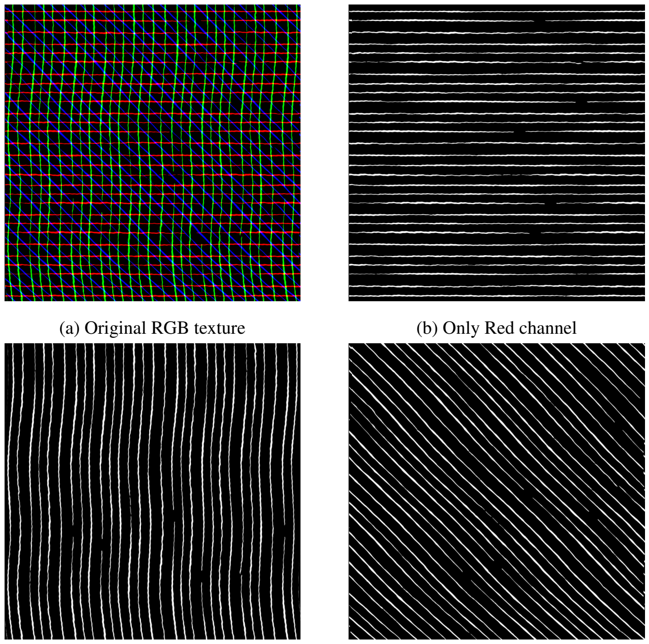

(a) Original RGB texture (b) Only Red channel (c) Only Green channel (d) Only Blue channel

Figure 1: Cross-hatching RGB texture.

$$S_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

$$S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

This two kernels are convolved in the neighborhood of each pixel to identify the regions where the change (gradient) is maximized. To detect an edge, we use the max value of both results, and compare them to a threshold passed to the shader as `uniform float` values. This is done for the Normal Buffer, and for the Depth Buffer. In our implementation, different threshold values can be used for the Normal and Depth Buffers.

2.3 Final Render

A final render is done using both the rendered view from the first pass, and the edge detection results from the second pass. Whenever an edge is detected, that pixel is painted black on top of the scene rendered by the first pass. If no edge is detected, the pixel is left as it is.

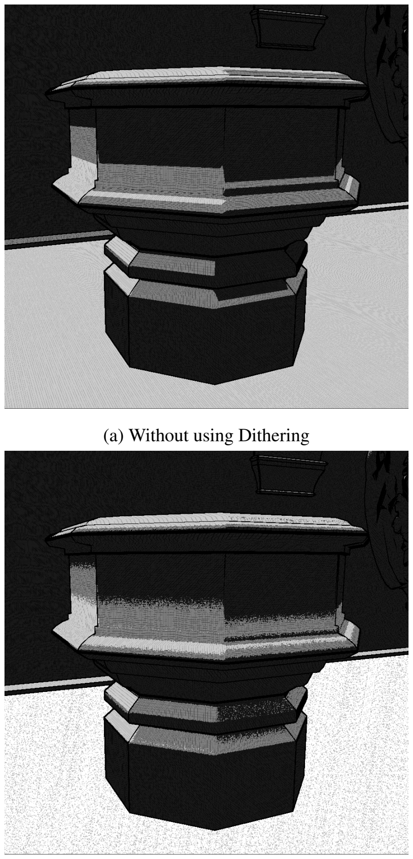
3 Results

The toon shading, cross-hatching, and edge detection, as well as the final result, can be seen in Figure 3.

Sample images for the same scene view, using the same light direction, but with different parameters for each step, can be seen in Figure 4

4 Discussion

We are highly satisfied with the visual results of our implementation. It is highly customizable, at a low performance cost. Sadly, due to time constrains, many ideas and potential improvements were discarded in the final version.


(a) Without using Dithering

(b) Using Dithering

Figure 2: Dithering Effect

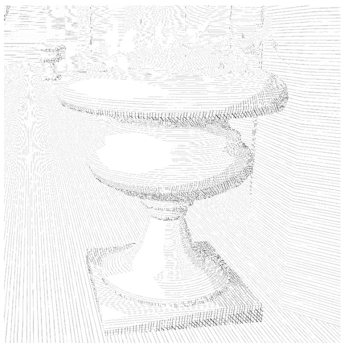
4.1 Potential improvements

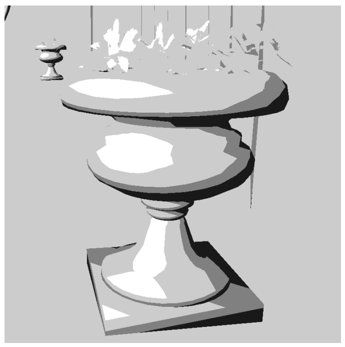
Many improvements can be made to our implementation

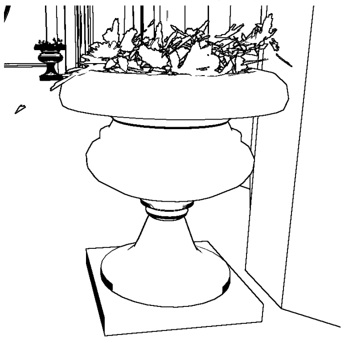
- The first improvement to be done, it's the addition of colored textures for the toon shading part. Right now, our implementation discard every texture embedded in the 3D object, as every single geometry it's drawn using our shader. This improvement could add great value to the final aesthethics of our shader.
- Another improvement, is the idea of using different shading for different objects in the scene. This includes both different toon shadings, as well as different cross-hatching textures.
- The final version of our implementation uses 3 rendering passes instead of 2. One for the toon shading and cross-hatching, one for the edge detection, and one for the final view. This was done for debugging reasons, and can be easily turned into 2 passes again.

References

YOUTUBE: USELESS GAME DEV. Moebius-style 3d rendering. `https://www.youtube.com/watch?v=jlKNOirh66E/`.


(a) Cross-Hatching only


(b) Toon shading only


(c) Edge detection only

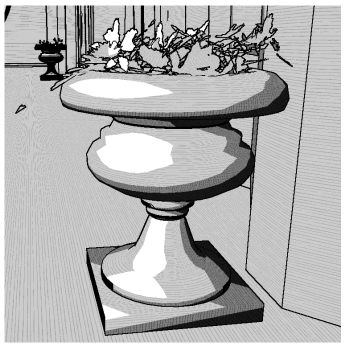
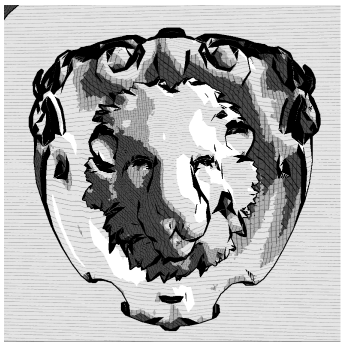
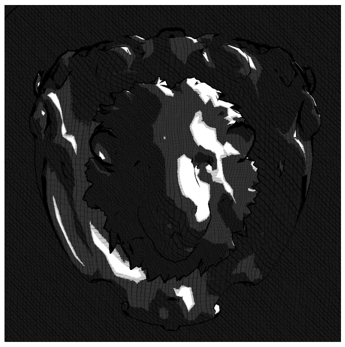
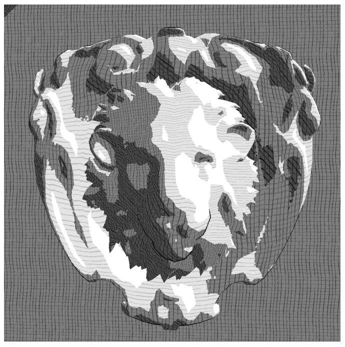

(d) Final scene view

Figure 3: All 3 steps in our algorithm, and the final scene view, no dithering


(a)


(b)


(c)

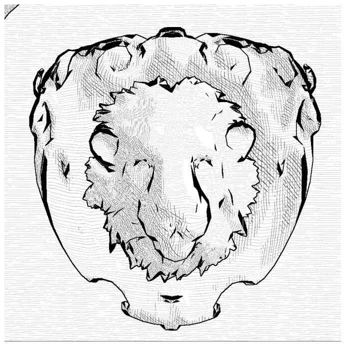

(d)

Figure 4: Different results using the same light direction, but different parameters for every step.