EDAN35 Real-time hatching

Elliot Jalgard* David Abramian †

Lund University
Sweden

Abstract

This paper is about a project done in the course EDAN35 (High Performance Computer Graphics). Mainly through the use of shaders and employing hatching techniques, we have achieved a real-time hand drawn effect in a 3D environment, applied on objects created for an entirely different purpose.

1 Introduction

The aim of this project is twofold: on one hand we implemented an edge detection algorithm, while on the other we have implemented a hatching algorithm. Both of these algorithms combine to create an overall hand-drawn effect on a scene.

Edge detection algorithms employ a series of techniques to detect the discontinuities in a picture. In our case we applied these techniques to the depth and normal buffers, and used them to be able to draw the edges of objects.

According Praun, E. et al: "Hatching generally refers to groups of strokes with spatially-coherent direction and quality. The local density of strokes controls tone for shading. Their character and aggregate arrangement suggests surface texture. Lastly, their direction on a surface often follows principal curvatures or other natural parametrization, thereby revealing bulk or form." [Emil Praun 2001]

In this project we have successfully implemented a hatching algorithm using as base the Bonobo framework provided for us.

2 Algorithms and Application

2.1 Edge detection

The chosen implementation of edge detection was based on a basic image processing technique of two-dimensional convolution of the image with a specific kernel matrix K. By choosing different kernels we can achieve different effects, such as blurring, sharpening of edges, edge detection, etc. For edge detection, there are a variety of kernels available, which provide different results. Our chosen kernel was the following one:

$$K = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

It is important to note that the total sum of the matrix values is 0. By examining the behavior of this kernel when executing a convolution with an image we can see that for uniformly colored surfaces the output is 0, while when an edge is encountered it gives either a very positive or very negative output, depending on whether the edge is ascending or descending. In a sense, this convolution can be viewed as an approximation of the two-dimensional derivative of the image, which in turn produces the edges. By taking the absolute value of the result of the convolution we get a new image with the edges marked in white.

We applied this edge detection technique both on the normal buffer and on the depth buffer, thus obtaining two mostly complementary results that we added together to get the overall edge image. Finally, in the last step of shading we applied a black color to all the pixels that corresponded to values above a certain threshold on the edge image, thus showing black edges on the objects of the scene.

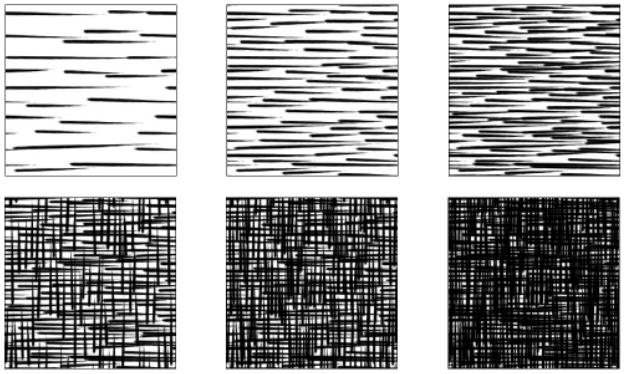

Figure 1: Rendered with the textures mapped to the objects

2.2 Hatching

In this project we followed the hatching method described in [Emil Praun 2001]. In order to achieve hand drawn strokes we apply them as a texture on the objects, instead of rendering them as individual primitives. One important distinction to be made between photorealistic rendering and hatching is that in the former the applied texture is used mainly to convey material detail, while in the later they additionally convey shading. Thus, to get the right effect we need to change the density of the strokes according to the light level of the object. In order to achieve this we employ tonal art maps, which are collections of textures that represent different levels of shading, where every level is computed by adding more strokes to the previous level. This allows us to interpolate between two textures without them clashing with each other. One of the tonal art maps we employed is illustrated in Figure 1.

In order to implement this algorithm we had to send the texture coordinates of all objects from the fill_gbuffer shader to the resolve_deferred shader. We did this through the diffuse buffer, since we do not need the diffuse properties of the materials in order to implement hatching. The lighting section of the program is unchanged form Assignment 2. Most of the processing takes place in the resolve_deffered shader. The first thing we do is calculate a single value of light for each pixel. We do the following formula:

$$Y = 0.2126R + 0.7152G + 0.0722B$$

This formula calculates the luminance of a pixel based on its RGB values, effectively turning a color image into a grayscale one. The next step is the selection of the coordinates we use to map the textures. We decided to allow for two different options: screen-space coordinates and object texture coordinates. By applying the former, the hatching images are fixed to the screen, but the line density is dependent on the light levels of each pixel, while the later

*ada10eja@student.lu.se
†dav.abramian@gmail.com

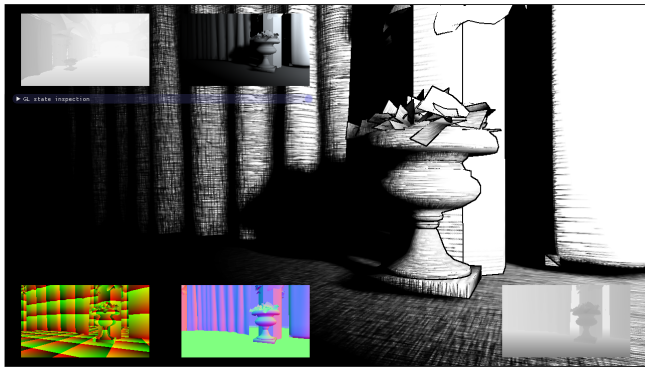

Figure 2: Rendered with the textures mapped to the objects

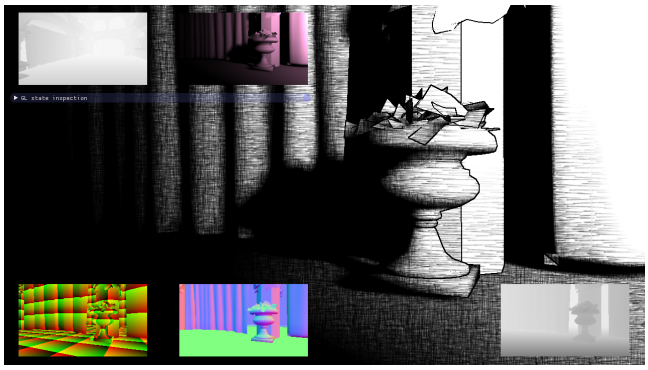

Figure 3: Rendered with the textures mapped to the screen coordinates

applies the hatching texture to each object as if it were a regular texture.

To achieve a seamless transition in line density we set certain threshold for the light values and linearly interpolated consecutive pairs of hatching textures between them. As was mentioned before, the fact that each level of the tonal art map is based on the previous one makes the result of this interpolation smooth and seamless. The last step of the shading process checks if the current pixel corresponds to an edge, and if so paint it black. Otherwise it applies the value calculated previously from the hatching textures.

2.3 Hatching textures

Besides using the hatching textures from [Emil Praun 2001], we tried to experiment with creating our own tonal art maps. We succeeded to do so in two different ways.

The tonal art map composed of different ellipses was created using the Paint.net program, simply by drawing successive layers of ellipses and saving them as different textures. The main challenge with this approach is achieving seamless textures. To do this we employed the SeamlessHelper plugin, which divides the image into four quadrants and switches the places of the opposed ones.

In addition, we wrote a JavaScript tool for creating our own hatching textures. It works by creating a canvas on which we procedurally draw lines of different color and width. This time, too, the main issue was with achieving seamlessness in the textures, but this was eventually solved aswell.

3 Results

We succeeded in implementing both the edge detection and the hatching algorithm. Figures 2 and 3 show a sample of the resulting look.

4 Discussion

We found that mapping the textures to the screen coordinates makes a nice effect from a static viewpoint. Once the camera starts moving around the textures dont follow the objects. The effect didnt give a hand-drawn effect. By using the objects texture coordinates, moving the camera around felt more natural. We did however encounter a problem here, since the texture coordinates are not generic for the spatial dimension. In order to achieve a more uniform effect independent of the size of the object, we came up with a modulo solution. It is not perfect, some objects have tighter textures than others, but the overall effect is close to what we desired.

References

EMIL PRAUN, HUGUES HOPPE, M. W. A. F. 2001. Real-time hatching. *ACM SIGGRAPH*.