Thirteenth Eurographics Workshop on Rendering (2002)
P. Debevec and S. Gibson (Editors)

Real-Time Halftoning: A Primitive For Non-Photorealistic Shading

Bert Freudenberg, Maic Masuch, and Thomas Strothotte

Institut für Simulation und Graphik, Otto-von-Guericke-Universität Magdeburg, Magdeburg, Germany

Abstract

We introduce halftoning as a general primitive for real-time non-photorealistic shading. It is capable of producing a variety of rendering styles, ranging from engraving with lighting-dependent line width to pen-and-ink style drawings using prioritized stroke textures. Since monitor resolution is limited we employ a smooth threshold function that provides stroke antialiasing.

By applying the halftone screen in texture space and evaluating the threshold function for each pixel we can influence the shading on a pixel-by-pixel basis. This enables many effects to be used, including indication mapping and individual stroke lighting. Our real-time halftoning method is a drop-in replacement for conventional multitexturing and runs on commodity hardware. Thus, it is easy to integrate in existing applications, as we demonstrate with an artistically rendered level in a game engine.

1. Introduction

A number of recent works on halftoning and artistic screening has demonstrated that a wide range of styles can be achieved with these methods. Yet, the underlying idea of halftoning, thresholding against a halftone screen, is very simple. We propose a function that implements halftoning in the texture blending hardware. This provides a new, flexible primitive for a variety of non-photorealistic rendering styles that still run at full texturing speed of the hardware.

The particular contribution of this work is

- a real-time halftoning method for non-photorealistic shading that also antialiases the halftoned image (§3),
- a scheme that allows to store multiple texture layers of increasing darkness in a single texture and extract a number of these layers according to a target tone on a per-pixel basis (§4),
- an approach for fast non-photorealistic shading effects, in particular indication mapping and lighting-dependent strokes (§5),
- a simple implementation of the above that runs on commodity graphics hardware and is easy to integrate in existing applications, thereby creating a simple to use non-photorealistic shading method (§6).

2. Related Work

Since we aim at real-time rendering of non-trivial scenes, drawing each shading stroke separately is too expensive (see Figure 6 which consists of millions of individual strokes). We have to use the existing texturing hardware. Both Lake et al. and Praun et al. have explored a similar direction[1, 2], which is to store multiple layers of strokes in textures and choosing at run-time which of these to display. While Lake can control the shading per polygon (i.e., flat shading), Praun's approach provides control at vertex level (i.e., gouraud shading). Both methods do not support per-pixel shading effects, like darkening parts of polygons with surface textures.

Store multi-layers of strokes in textures, choose

Recently, Durand et al. demonstrated an interactive drawing system[3] that implements a smooth threshold function via SGI's pixel-textures. It is not designed for rendering 3D scenes but draws strokes individually under user control according to a reference image.

Shading by halftoning has a long tradition in off-line rendering, like the digital engravings and copper plates of Leister[4] or recent work like Ostromoukhov's engravings[5] and Veryovka and Buchanan's[6] halftoning works. These methods are great references for possible styles.

Similar effects can be achieved by procedural texturing as

described by Johnston[7]. However, while real-time shading languages are an active research topic[8], these shaders are still too expensive on common hardware.

3. Halftoning With Texturing Hardware

For our real-time halftoning process we employ the configurable texture blending hardware that is nowadays standard even in commodity hardware, like NVIDIA's GeForce, ATI's Radeon, etc.[9, 10] The basic idea of halftoning is to apply a threshold function per pixel to a target intensity, where the threshold value is stored in the halftone screen. In conventional halftoning, this threshold function is binary: If the target intensity is below the threshold, a black pixel is generated, otherwise a white pixel. We show how this function can be computed in hardware first, before presenting a threshold function that is more apt for interactive application.

3.1. Binary Threshold Scheme

We store the halftone screen as a texture `tex`. The target intensity can be determined in several ways. We assume using the interpolated vertex color `col` for now (see Section 5 for more sophisticated variants). The blending hardware provides a `mux()` function that selects between two colors based on a third:

```
mux(a,b,c): (a <= 0.5) ? b : c
```

so we can use an implementation like

```
tmp = add(tex, col - 0.5);
out = mux(tmp, 0, 1);
```

An example image rendered using this approach is shown in Figure 1(c).

This works indeed well and fast, but has two drawbacks: It only generates black and white pixels, and it uses up two blending stages, which is what the majority of installed graphics hardware provides. The strict use of only black and white pixels leads to very disturbing aliasing of edges and popping pixels when the lighting changes. There is no smooth transition from black to white, as this would require gray values which are explicitly prohibited in the binary threshold scheme.

(a) (b) (c) (d)

Figure 1: *Halftoning samples. (a) halftone screen, (b) target intensity, (c) binary threshold, (d) smooth threshold*

3.2. Smooth Threshold Scheme

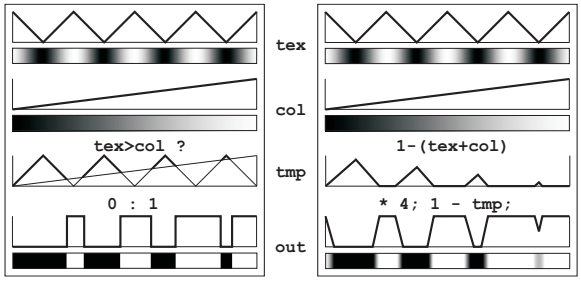
To overcome the deficiencies described in section 3.1 we introduce a *smooth threshold* scheme. It is designed to preserve to a certain degree the gray levels found in the halftoning screen. Strictly speaking, this is not *half*-toning anymore, but since the majority of pixels will be black or white, we will stay with that term. While the binary threshold function has infinite slope at the threshold value, the smooth threshold function uses a constant slope that creates a less rapid transition from white to black.

To implement the smooth threshold function we use the following blending configuration:

```
tmp = add(1 - tex, -col) * C;
out = 1 - tmp;
```

This adds the target intensity `col` to the halftone screen `tex`, inverts it, and scales the result by `C` (we found the value 4 useful for the kind of halftone screens we used) before inverting again. The double inversion (`1-(tex+col)` and `1-tmp`) is necessary because the scaling shifts gray values towards white, while we want to scale towards black (see Figure 2). The result is clamped by the hardware to the 0–1 range. Applying this function instead of a binary threshold results in an antialiased image, as illustrated in Figure 1(d).

An additional advantage of this scheme compared to the binary scheme is that on NVIDIA's register combiner architecture only one general combiner is needed because the final inversion can be executed in the "final" combiner. This leaves one general combiner free to use even on older GeForce class hardware.



(a) (b)

Figure 2: *Halftoning schemes. (a) binary threshold, (b) smooth threshold. Inputs and outputs are shown as both, function plots and actual gray values.*

4. Creating Halftone Screens

To present shading by halftoning, the ratio of black to white pixels has to be varied based on the ~~lighting conditions~~. Artistically, there are two major directions in which this is done: By varying the line width while maintaining a roughly constant stroke density, as in engravings and woodcuts, or by

adding strokes of constant line width which results in varying density, as in pen-and-ink style hatching. We show how to construct halftone screens for these styles now.

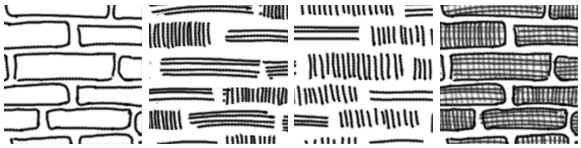
4.1. Engraving

For engravings we can build on a variety of previous work like Ostromoukov's[5] or Veryovka's[6] work. Their dither screens can be directly employed in our real-time system. Precondition for our texture-based approach is a suitable surface parameterization. Usually, the dither screens are two-dimensional, but for the special case of a single set of parallel engraving lines a one-dimensional texture encoding a gray ramp, like the one shown in Figure 2, is sufficient. Figure 5 was rendered using this texture.

4.2. Stroke Textures

Inspired by Winkenbach and Salesin's work[11] we created a method to construct halftone screens that can render prioritized stroke textures in real-time.

The input to the halftone screen encoding process is a set of independent *stroke layers*. These are drawn in a painting package using black ink and antialiased strokes on a transparent background. Each layer will be encoded as gray level: The first layer remains unscaled (ranging from black to white), each successive layer is compressed to range from a lighter gray value to white. Then the layers are drawn into the halftone screen, starting with the last and lightest layer to the first, darkest layer. The process is demonstrated in Figure 3.



(a) (b) (c) (d)

Figure 3: *Stroke map encoding. (a) darkest layer, (b) lighter layer, (c) lightest layer, (d) stroke map*

This encoding scheme for the halftoning screen has the nice property that it is intuitive to the texture artist how a stroke map will be rendered. When shading a surface, the layers will get drawn in order of decreasing darkness. That is, in very light regions only the most dark lines are visible, while in darker regions, additional layers (encoded with lighter values) will appear. While this may sound contrary, it should become obvious by examining Figure 4.

The encoding process could as well be used to compress automatically generated hatching strokes[2], but we aimed at using stroke textures to depict and distinguish materials. A viable alternative might be to automatically extract the layers from real-world textures[12]. Our method also works well

(a) (b) (c)

Figure 4: *Stroke map decoding. (a) stroke map, (b) target intensity, (c) shaded image*

with mipmapping to control stroke appearance depending on distance[13, 2].

5. Special Effects

Since the basic real-time hatching technique does not use up all texturing hardware resources (in fact, only one portion of one combiner and one channel of a texture), there's headroom for adding a range of special effects. In a very similar way to photorealistic rendering, all effects that use textures could be employed.

The effects are generally executed ~~in the earlier texture stages,~~ the last stage must be spared for applying the threshold function. Essentially, the intensity value `col` used in Section 3 is replaced by a computed value. We only present two techniques particularly related to NPR here: texture indication and stroke lighting.

5.1. Indication Mapping

Texture indication is the technique of gradually fading out detail in areas of less visual importance. It was introduced to computer-generated line drawings by Winkenbach[11] and it adds greatly to the non-technical look of these kind of renderings.

Our system implements texture indication by storing a low resolution *indication map* along with the model. A signed value is stored in this map: zero does not change the default lighting, while positive or negative values lighten or darken the rendering. The result of this technique can be seen in Figure 7.

5.2. Individual Stroke Lighting

We use ~~per pixel~~ arithmetic to evaluate the threshold function, so we also can evaluate the lighting at each pixel. We assign a normal to each pixel in a stroke by supplying a normal map suitable for dot-product bump mapping. This allows a single stroke to change it's width depending on the light direction, a technique widely employed in traditional drawing. This effect is used on the brick outlines in Figure 7: The light comes from the upper right, so only the lower and left outline strokes are drawn.

On hardware with two texture combiners we only can do either indication mapping or stroke lighting but not both; combining the bump mapping result with the indication map would require an additional stage. Thus, the full-featured model of Figure 7 can only be viewed on GeForce3 or equivalent hardware.

6. Application

To test our real-time halftoning technique, we modified a level for the Shark3D game engine[14] by replacing textures and tuning the lighting. We didn't even touch the engine source code: all that was needed was to write a new custom shader. This proves the rapid applicability of the method. At a resolution of 1600x1200 pixels running on a notebook with 1.2 GHz Pentium III and a GeForce2Go graphics accelerator this runs with more than 20 frames per second. In fact, the application is CPU limited since the dynamic lightmaps are updated in software. A snapshot from the game level is shown in Figure 6.

7. Conclusions

We have introduced halftoning as a flexible shading primitive for real-time non-photorealistic rendering. We demonstrated how to adapt traditional halftoning for the limited screen resolution with a smooth threshold function, and explored several styles and effects enabled by this method, especially a real-time implementation of stroke textures.

There remain some problems, still. The most visible artifact is the insufficient separation of overlapping strokes. Also, the effects are limited by the number of available combiners, because our technique does not lend itself easily to multi-pass rendering. Also, we have neglected outlines so far, we hope to find similarly flexible methods for drawing edges, soon.

The strongest aspect of our shading method is that it is very easy to use in applications, and only requires commodity hardware to run. We hope to see more non-standard rendering styles employed in various applications, but especially computer games, in the future.

References

1. Adam Lake, Carl Marshall, Mark Harris, and Marc Blackstein. Stylized rendering techniques for scalable real-time 3D animation. In *Proceedings NPAR 2000*, pages 13–20, 2000. 1
2. Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. *Proceedings of SIGGRAPH 2001*, pages 579–584, August 2001. 1, 3
3. Fredo Durand, Victor Ostromoukhov, Mathieu Miller, Francois Duranleau, and Julie Dorsey. Decoupling strokes and high-level attributes for interactive traditional drawing. In *12th Eurographics Workshop on Rendering*, 2001. 1
4. W. Leister. Computer generated copper plates. In *Computer Graphics Forum*, volume 13, pages 69–77, January 1994. 1
5. Victor Ostromoukhov. Digital facial engraving. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 417–424, Los Angeles, California, August 1999. ACM SIGGRAPH / Addison Wesley Longman. 1, 3
6. Oleg Veryovka and John Buchanan. Comprehensive halftoning of 3D scenes. In *Proceedings of Eurographics 99*, Computer Graphics Forum, pages C/13–C/22, 1999. 1, 3
7. Scott F. Johnston. Mock media. In *Advanced RenderMan: Beyond the Companion*, SIGGRAPH 98 Course Notes #11, pages 113–121, 1998. 2
8. Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proc. SIGGRAPH 01*, Computer Graphics Proceedings, Annual Conference Series, pages 159–170, 2001. 2
9. Evan Hart and Jason L. Mitchell. Hardware Shading with EXT_vertex_shader and ATI_fragment_shader. http://www.ati.com/developer/ATIHardwareShading.pdf. 2
10. John Spitzer. Programmable Texture Blending. NVIDIA. http://developer.nvidia.com/view.asp?IO=programmable_texture_blending. 2
11. Georges Winkenbach and David Salesin. Computer-generated pen-and-ink illustration. In *Proc. SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 91–100. ACM Press, July 1994. 3, 6
12. Oleg Veryovka and John Buchanan. Texture-based dither matrices. *Computer Graphics Forum*, 19(1):51–64, 2000. 3
13. Bert Freudenberg, Maic Masuch, and Thomas Strothotte. Walk-Through Illustrations: Frame-Coherent Pen-and-Ink Style in a Game Engine. *Computer Graphics Forum: Proceedings Eurographics 2001*, 20(3):184–191, 2001. 3
14. Spinor GmbH. Shark3d. http://www.shark3d.de/. 4

Acknowledgments

The authors which to thank Folker Schamel for the support in using the Shark3D engine, and our students Bert Vehmeier, Ragnar Bade, Christian Mantei, Birger Schmidt, and Niklas Röber for modeling, texturing, and preparing images for this paper.

Figure 5: *Engraving-style rendering of the Utah Teapot*

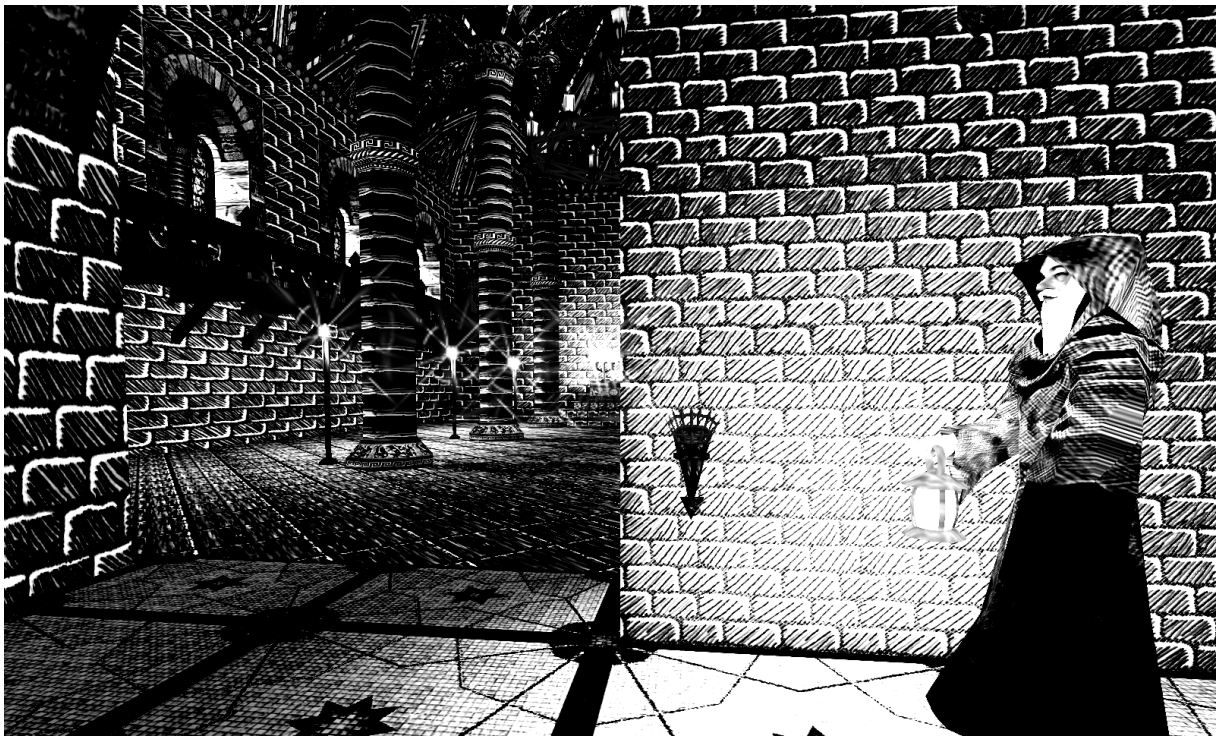


Figure 6: *Screen shot from a level in a game engine*

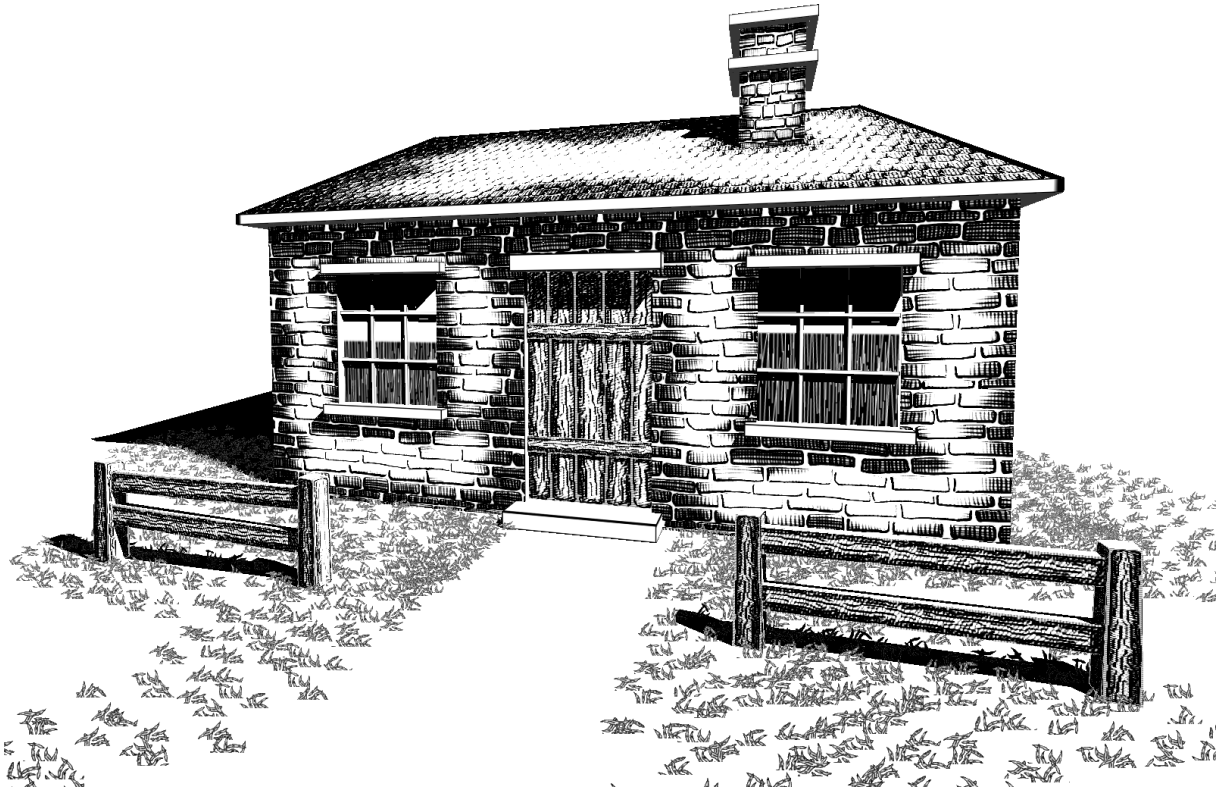


Figure 7: *Pen-and-ink style rendering featuring stroke maps, indication mapping and individual stroke lighting (modeled after Winkenbach's house[11])*

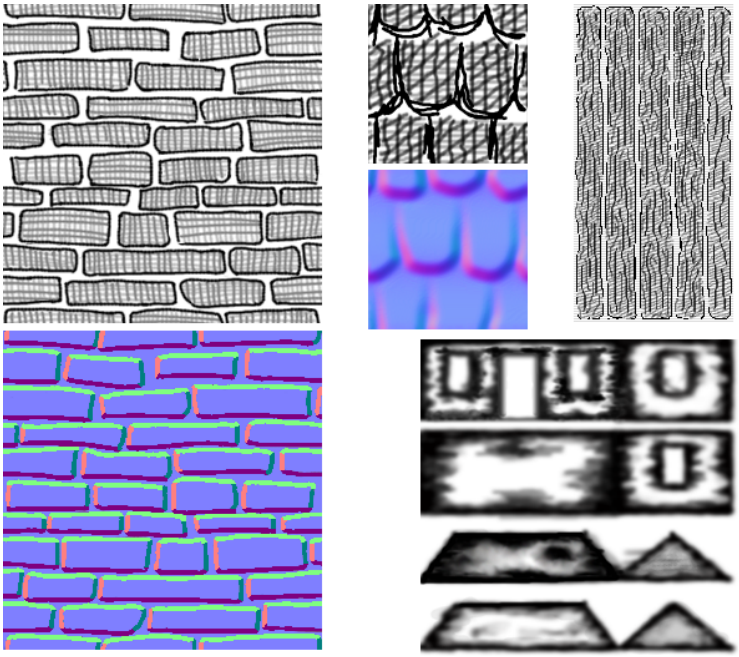


Figure 8: *Some stroke maps, normal maps, and indication maps used in Figure 7*