To appear in Andrew Kirmse (ed.), *Game Programming Gems 4*, Charles River Media, 2004. All rights reserved.

Real-Time Halftoning: Fast and Simple Stylized Shading

Bert Freudenberg, Maic Masuch, and Thomas Strothotte – University of Magdeburg
bert@isg.cs.uni-magdeburg.de, masuch@isg.cs.uni-magdeburg.de, tstr@isg.cs.uni-magdeburg.de

This gem introduces halftoning as a way for implementing non-photorealistic rendering styles for computer games. The technique uses only the conventional multi-texturing pipeline on common hardware. We show how to create halftone screens for images that resemble pen-and-ink drawing styles and how to implement fast halftone rendering with modest pixel shading hardware.

Introduction

In this gem we introduce a technique borrowed and adapted from non-interactive hardcopy to real-time environments. Similar approaches comparable to the one presented in this gem are great resources for inspiration. Some are less flexible [Lake00, Praun01] while others are more complicated [Webb02].

Halftoning in its original form is the procedure used to print images with gray levels using only black ink. It does so by varying the size of ink dots, and thus, the ratio of paper area to inked area. Viewed from a distance, a certain tone is perceived. A similar effect is used when an artist is drawing a picture with pen and ink, where more lines are put in areas that should appear darker. Yet another variation is employed in engravings or wood cuts, where the width of lines is adjusted to depict variations in shading. All these styles can be recreated by applying real-time halftoning.

Similar to traditional halftoning, when the image intensity changes in a given neighborhood of pixels, pixels are not dimmed. Instead, *more* black pixels are shown while the remaining pixels stay white, so the overall perceived intensity is lower. One interesting difference between halftoning a static image versus an interactive environment is that the halftone screen is not fixed to the screen but rather attached to the objects in 3D space. Otherwise, the objects would appear to “swim” behind the halftone screen, which is also known as the “shower-door” effect.

Principle

The basic ingredient in halftoning that determines the visual appearance most is the *halftone screen*. This is a gray-scale texture containing *threshold* values. To create a halftoned image from a given input image, the intensity of each pixel is compared to the corresponding threshold value in the halftone screen, and a black or white pixel is written depending on the outcome of the comparison (see Figure 1). If H is the halftone screen threshold value and L is the target intensity, we could write the threshold function in a C-like fashion as

$$H > (1-L)\ ?\ 1 : 0.$$

Here, $(1-L)$ represents “darkness” which reflects the subtractive color model of ink on paper.

So to make real-time halftoning work, we need a halftone screen texture, and a function to perform the threshold operation. The halftone texture specifies how the image intensity should be mapped into a black-and-white rendering. It may be as simple as the regular dot raster used to print the images in this book, or as complex as the brick texture in Figure 3.

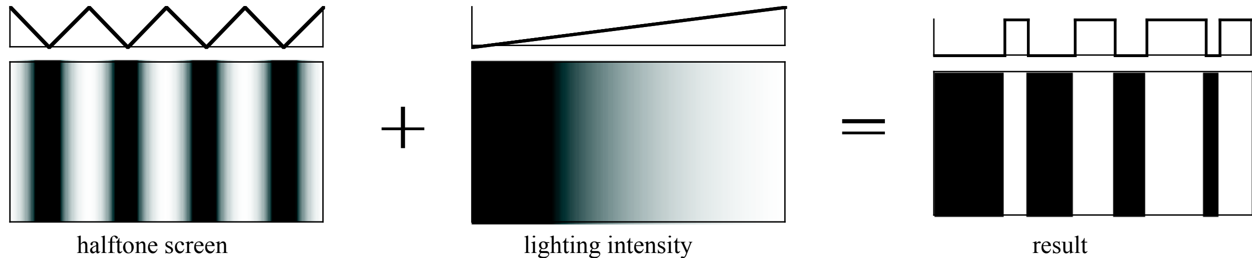

Figure 1. Traditional halftoning with a threshold function. Each value in the halftone screen is compared to the intensity value. If the halftone value is higher than the darkness (which is one minus intensity), a white pixel is generated, otherwise a black pixel.

Creating Halftone Screens

Halftone screens can be created either procedurally or manually. The latter is more desirable since it offers more artistic flexibility. One example of a procedural halftone screen would be smooth grayscale stripes as used in Figure 1. This results in lines whose widths depend on the lighting and resembling a woodcut printing style (see Figure 2). Other procedural screens can be created for hatching [Lake00, Praun01].

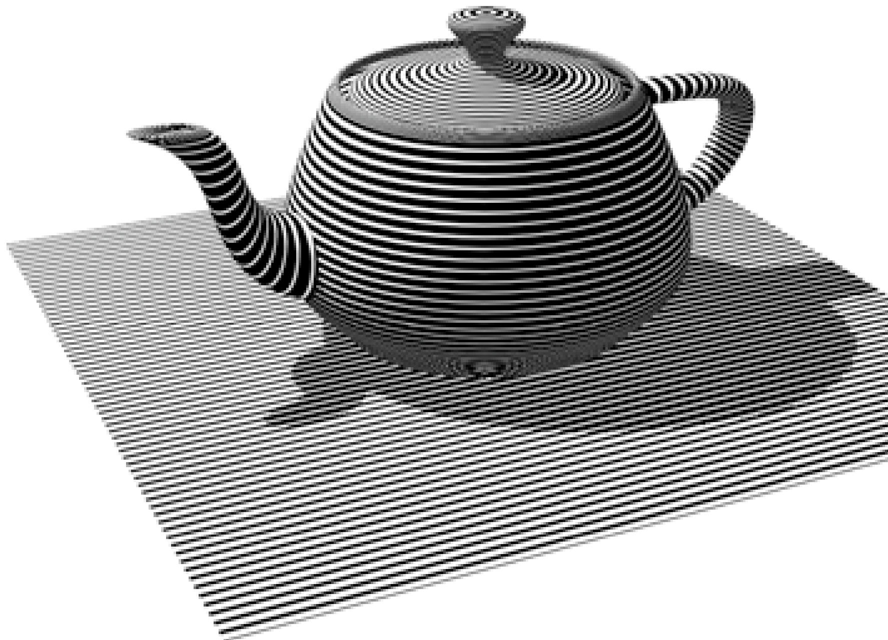

Figure 2. Halftoning with grayscale stripes like in Figure 1 creates a woodcut-like style. Shading is conveyed by varying line widths.

For drawing a halftone screen by hand, we developed a layer-based process that yields nice results (see left three images of Figure 3). Each layer is drawn in black on transparent layers in

PhotoShop. As more layers accumulate, the screen gets darker and darker. This is exactly what we want for halftoning. When all layers are painted, they are combined into one halftone screen by changing each layer into a distinct shade of gray. The basic layer remains black, whereas successive layers are assigned lighter grays. The compositing needs to be performed in reverse order so the black layer is drawn on top of the lighter layers. The resulting image is exported as a single 8-bit grayscale texture.

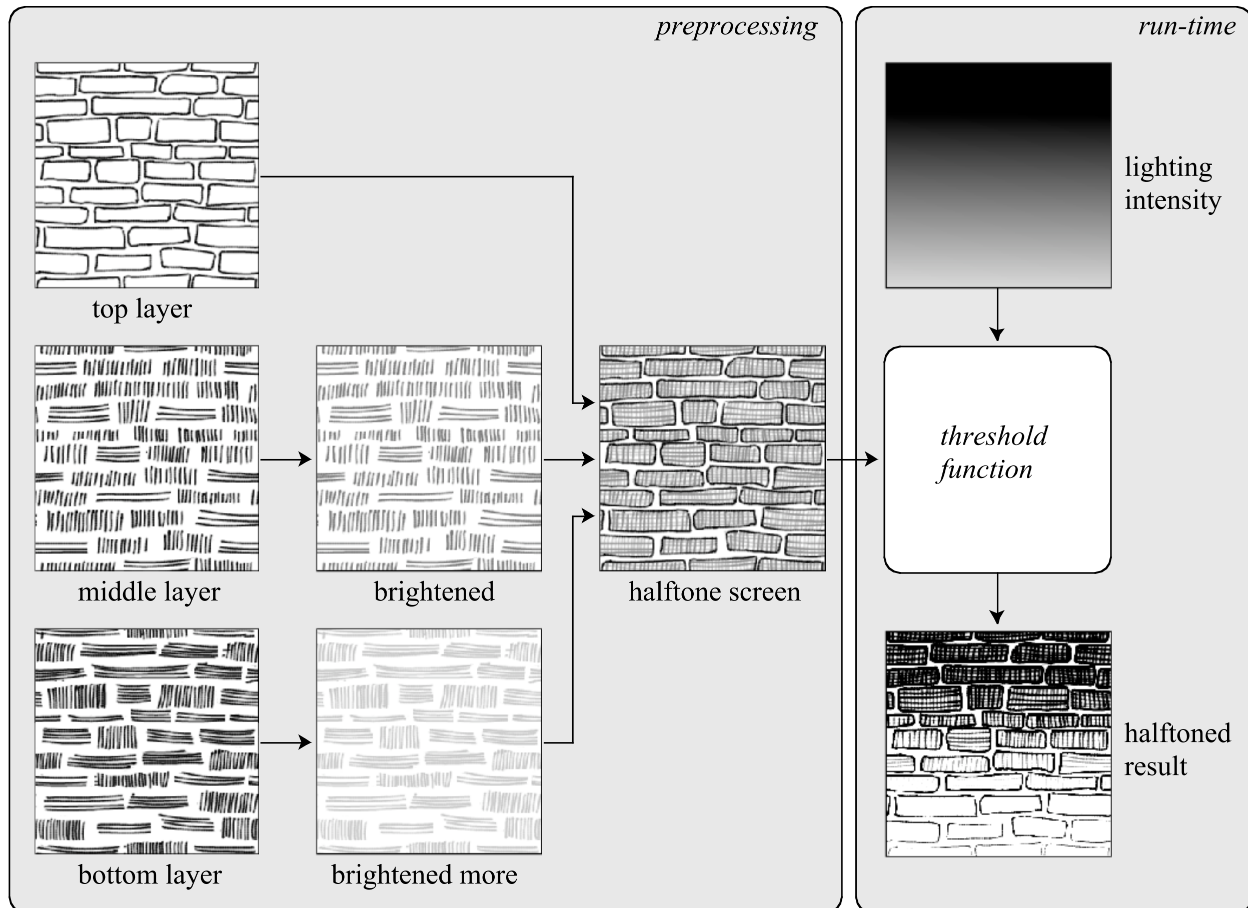

Figure 3. Halftoning with layered strokes. Several stroke layers are combined into a single halftone texture by encoding the layer's priority as gray level. At runtime, the lighting intensity is thresholded against this texture to produce a halftoned rendering. In result, more strokes are displayed in regions of lower intensity.

This layer-based authoring worked well for us, but you can use any method to create the halftone texture. One way to think of the values in the halftone screen is that each pixel encodes a "priority" for drawing strokes – darker strokes have higher priority and get displayed first, even in bright regions. Lower-priority strokes (drawn in light gray) are only displayed in darker areas, whereas white parts of the halftone screen will always remain blank in the rendering, too.

A Threshold Function For Limited Pixel Shaders

For real-time halftoning, the threshold function, which takes a texture and an intensity value as input, has to be evaluated at each pixel. This can be implemented with a conditional operation

 All rights reserved.

that outputs black or white fragments depending on a comparison. While more advanced pixel shader versions offer a compare operation that can be used, the least common denominator in OpenGL is support for the `texture_env_combine_ARB` extension. We came up with a formula for a *smooth* threshold function (in contrast to the sharp function normally used for thresholding) that only uses functionality provided by this extension. In the following, H stands for the halftone screen, and L for the lighting intensity:

$$1-4\left(1-\left(H+L\right)\right).$$

Since we are dealing with colors here, every operation implicitly clamps its result to the [0,1] interval. Unfortunately, the straight implementation of this formula would need three texturing stages (sum, invert and scale, invert again), while some common graphics boards only provide two. With a little rearrangement, though, we get

$$1-4\left(\left(1-H\right)-L\right)$$

which can be implemented in two stages (scaled difference of inverted operand, inversion). When we experimented with higher values than the constant 4 in this equation, we lost one nice advantage of the smooth threshold function, which is that it produces anti-aliased images. This is a form of shader anti-aliasing that removes high image frequencies introduced by a shader (a sharp step function essentially has unlimited frequency and will alias at any resolution).

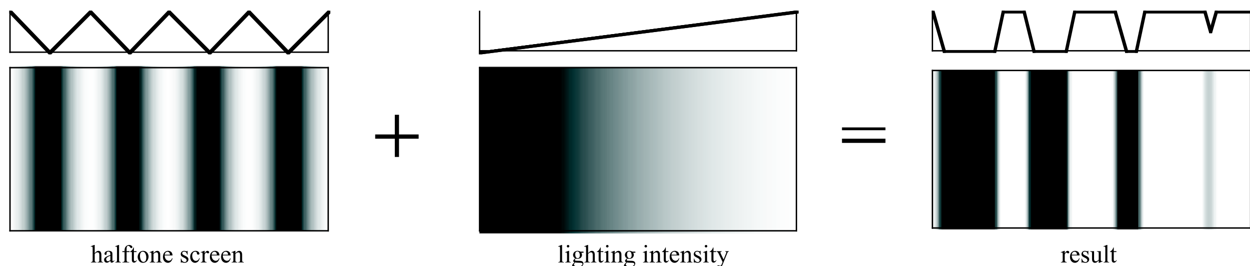

Figure 4. With a smooth threshold function, the resulting image is not only black and white, but some gray levels are preserved. This reduces shader aliasing artifacts.

Threshold Function Implementation

To implement the threshold function described in the previous section, two multi-texturing stages need to be setup in the following way to perform the thresholding function:

```
// macro to keep lines short
#define TexEnv(pname, param) \
    glTexEnvi(GL_TEXTURE_ENV, pname, param)

// stage 0: smooth threshold function 4*((1-L)-H)
glActiveTexture(GL_TEXTURE0);
TexEnv(GL_TEXTURE_ENV_MODE, GL_COMBINE);
TexEnv(GL_COMBINE_RGB, GL_SUBTRACT);
TexEnv(GL_SOURCE0_RGB, GL_PREVIOUS);
TexEnv(GL_SOURCE1_RGB, GL_TEXTURE);
TexEnv(GL_OPERAND0_RGB, GL_ONE_MINUS_SRC_COLOR);
TexEnv(GL_OPERAND1_RGB, GL_SRC_COLOR);
TexEnv(GL_RGB_SCALE,  4);
```

 All rights reserved.

```
// stage 1: invert function (1-previous)
glActiveTexture(GL_TEXTURE1);
TexEnv(GL_TEXTURE_ENV_MODE, GL_COMBINE);
TexEnv(GL_COMBINE_RGB,GL_REPLACE);
TexEnv(GL_SOURCE0_RGB,GL_PREVIOUS);
TexEnv(GL_OPERAND0_RGB, GL_ONE_MINUS_SRC_COLOR);
```

The first texture stage gets the per-vertex lighting value as `SOURCE0` and the halftone texture as `SOURCE1`. The first operand is subtracted from 1 using the `ONE_MINUS_SRC_COLOR` input mapping, and the overall operation is set to `SUBTRACT`. Finally, the result is multiplied by 4. The second texture stage only uses one operand and replaces its input with the inverse, again using the input mapping.

Note that you can perform all of this in the alpha portion of the texture combiners as well. This would free up the RGB portion for other uses. This equation can also be implemented in a shader program if the graphics hardware supports pixel shaders.

Example Implementation

To demonstrate how easy it is to integrate real-time halftoning into a traditional game engine [Shark3D], we converted a conventionally textured demo level [Requiem00] to a pen-and-ink comic style. In Figure 5 you can see the scene before and after our mod.

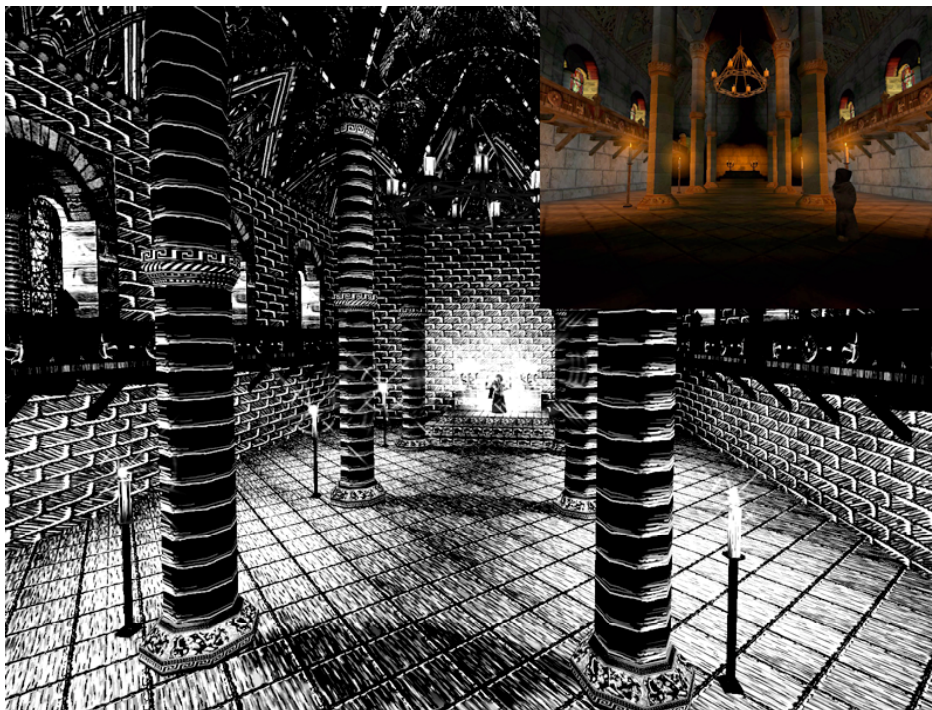

Figure 5. A conventionally textured and lit level (shown as inset) was converted into a dark comic style by employing real-time halftoning. Note how the shading is depicted by varying the number of strokes displayed.

First, the textures were converted to grayscale. Similarly, the colored light sources were changed to grayscale. Some of the textures, like in the ceiling or the colored glass windows, looked good enough after the grayscale conversion even without further touch up. Others, most notably the walls and floor, were drawn in a pen-and-ink like hatching style using the layer drawing technique described previously. Other changes include new textures for the candles' glow effects and the adjustment of lighting intensity to create harder shadows. On the programming side, only the shaders had to be rewritten to implement the threshold function.

Conclusion

This gem has presented real-time halftoning, a simple and fast technique for non-photorealistic shading suitable for games. It can provide a distinct look for an entire game or for special sequences. One can imagine an introduction for a game where the main character reads a comic novel and gets sucked right into it. There are many non-photorealistic visual styles yet to be explored in real-time graphics.

References

[Freudenberg02] Bert Freudenberg, Maic Masuch, Thomas Strothotte, "Real-Time Halftoning: A Primitive for Non-Photorealistic Shading," *Proceedings of the 13th Eurographics Workshop on Rendering*, pp. 227–231.

[Isenberg03] Tobias Isenberg, Bert Freudenberg, Nick Halper, Stefan Schlechtweg, Thomas Strothotte, "A Developer's Guide to Silhouette Algorithms for Polygonal Models," *IEEE Computer Graphics and Applications* 23(4): pp. 28–37, July/August 2003.

[Lake00] Adam Lake, Carl Marshall, Mark Harris, Marc Blackstein, "Stylized Rendering Techniques for Scalable Real-Time 3D Animation," *Proceedings of NPAR 2000, Symposium on Non-Photorealistic Animation and Rendering*, pp. 13–20.

[Praun01] Emil Praun, Hughes Hoppe, Matthew Webb, Adam Finkelstein, "Real-Time Hatching," *Proceedings of SIGGRAPH 2001*, pp. 581–586.

[Requiem00] Punkt im Raum, "Requiem: A technology study for the Shark 3D engine", available online at http://www.punkt-im-raum.com/eng/projekte_requiem.shtml

[Shark3D] Spinor GmbH, "Renderer and Shader Architecture," Shark3D engine technology white papers, available online at http://www.shark3d.com/goto/technology_render.html

[Strothotte02] Thomas Strothotte and Stefan Schlechtweg, "Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation," Morgan Kaufmann, San Francisco, 2002.

[Webb02] Matthew Webb, Emil Praun, Adam Finkelstein, Hugues Hoppe, "Fine Tone Control in Hardware Hatching," *Proceedings of NPAR 2002, International Symposium on Non Photorealistic Animation and Rendering*, pp. 53–58.