Real-Time Stroke-Based Halftoning

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

vorgelegt der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
von

Dipl.-Inf. Bert Freudenberg

geboren am 14. Februar 1972 in Zittau

Magdeburg, 19. November 2003

Abstract

This work deals with the non-photorealistic rendering of geometric models. We pursue an approach that combines the expressiveness of stroke-based rendering and the efficiency of halftoning to create a new genre of real-time rendering methods that are applicable to interactive applications such as computer games. Like in traditional halftoning, we create images with black color on a white background. Since strokes are used for shading, we refer to the ensuing methods as *stroke-based halftoning.*

To display strokes for this purpose, we introduce explicit and implicit rendering techniques specifically designed to be accelerated by modern graphics hardware. Explicit techniques determine the geometric extent of strokes before rendering them using vertex programs, while the implicit approaches are based on textures that latently contain stroke information which is revealed in the rendering process. Because outline strokes are important to the visual style created by stroke-based halftoning, methods to render these outlines are developed. Again, both explicit and implicit techniques are introduced. The usability of the new rendering approach is examined in case studies from the fields of archaeological visualization and non-photorealistic game rendering.

Zusammenfassung

Diese Arbeit beschäftigt sich mit der nicht-photorealistischen Darstellung geometrischer Modelle. Der verfolgte Ansatz verbindet die Ausdrucksstärke von linienbasierten Darstellungen mit der Effizienz von Halbton-Verfahren und schafft so eine neue Klasse von Echtzeit-Darstellungsmethoden, die in interaktiven Anwendungen wie beispielsweise Computerspielen angewandt werden können. Wie in traditionellen Halbton-Verfahren werden Bilder mit schwarzer Farbe auf weißem Grund erzeugt. Da Linien zur Schattierung verwendet werden, wird die neue Methode als *linien-basierte Halbtonerzeugung* bezeichnet.

Um für diesen Zweck Linien anzuzeigen, werden explizite und implizite Techniken eingeführt, die speziell auf die Beschleunigung durch moderne Graphikhardware ausgerichtet sind. Explizite Techniken bestimmen die geometrische Form der Linien vor der Darstellung mittels Vertexprogrammen, während die impliziten Techniken auf Texturen basieren, die latent die Linieninformationen enthalten, welche durch den Darstellungsprozess sichtbar werden. Da auch Umrisslinien wichtig für den durch linien-basierte Halbtonerzeugung hervorgerufenen visuellen Stil sind, werden Methoden entwickelt, um solche Umrisse darzustellen. Wiederum werden sowohl explizite als auch implizite Techniken vorgestellt. Abschließend wird die Anwendbarkeit des neuen Darstellungsverfahrens anhand von Fallstudien aus den Bereichen der archäologischen Visualisierung und der nicht-photorealistischen Spielegraphik untersucht.

Contents

1 Introduction

Non-photorealistic rendering has grown to become an important field of computer graphics over the last decade. Several hundred papers have been published as well as two books [Gooch and Gooch, 2001; Strothotte and Schlechtweg, 2002]. A dedicated biannual ACM conference was started in 2000. Despite the academic excitement surrounding these advances, NPR is rather sparingly used in production environments. Nonetheless, animated feature films using NPR have appeared, and some games are rendered in a cartoon style.

However, most of the large number of non-photorealistic rendering styles developed in research laboratories has not yet found their way into practice. Especially in interactive environments, very few NPR techniques are used. We conjecture that this is largely due to the absence of practical, fast, and robust rendering techniques that fit well into the traditional production process and yield high-quality results.

Such novel rendering styles are particularly apt for computer games, which traditionally explore unique looks and artistic qualities. However, the techniques need to work in real-time because of the interactive nature of games across a wide range of hardware, so as not to limit unnecessarily the potential customer base. Also, the CPU load should be minimal to free up resources for other tasks that the game engine needs to perform besides rendering.

1.1 Stroke-Based Rendering

This work focuses on a particular aspect of non-photorealistic rendering, generally referred to as *stroke-based shading*. Within computer graphics, shading with strokes is a technique that is unique to NPR: In photorealism, the basic unit is a pixel; in contrast, stroke-based rendering has as its basic unit the stroke, which is of larger size than a pixel. Shading is not conveyed by just dimming individual pixels, but by varying strokes in size, density, or color. The result is akin to traditional techniques: stroke width adjustment is used in wood-cut printing, varying stroke density is used in pen-and-ink illustration to depict shading via hatching, and the color of strokes defines the appearance in oil paintings.

While the concept of stroke-based shading has been at the center of research interest for quite some time [Winkenbach and Salesin, 1994; Deussen and Strothotte, 2000; Csébfalvi et al., 2001] and sophisticated implementations producing interesting images do exist (see Figure 1.1), to orchestrate the large number of strokes typically used to produce a well balanced, informative and/or esthetically pleasing rendition has preempted the design of real-time applications to make use of it. Vast numbers of strokes are required that need to be individually adjusted to create lighting effects. Frame coherence has to be ensured, as well as the dynamic adaption of stroke density to the size of a depicted object.

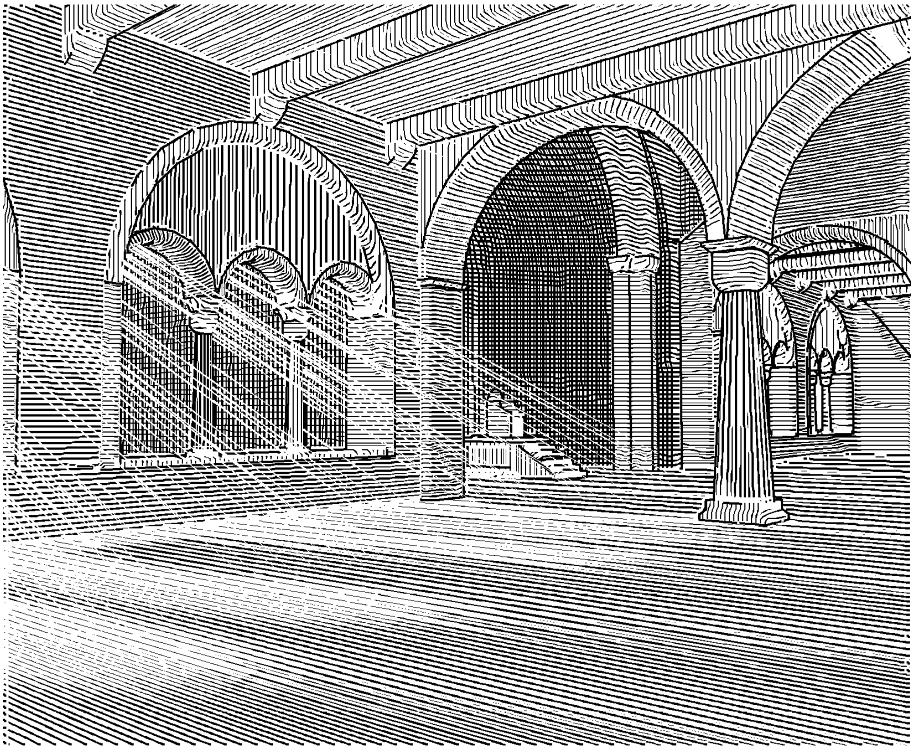


Figure 1.1: Example of stroke-based rendering [Hamel, 2000]

1.2 Halftoning

Another field in computer graphics that employs vast numbers of primitives to represent different intensities is halftoning. In contrast to stroke-based rendering, it uses very fine primitives with the goal to make patterns invisible (see Figure 1.2). Halftoning is typically used in conjunction with imaging devices that can only use a small number of colors, such as a fixed number of different inks in a printer. With halftoning methods, continuous-tone images can be reproduced on such machines.

Traditional halftoning, which seeks to minimize the recognizability of individual halftone primitives, has been extended to “artistic halftoning”. Here, the primitives are typically larger and are prominent in the image, giving it a particular style (see Figure 1.3).

Halftoning is especially interesting when it is brought into the context of real-time rendering. Threshold-based halftoning methods can operate in parallel, because each image element is created independent of its context. This makes it amenable to hardware acceleration.

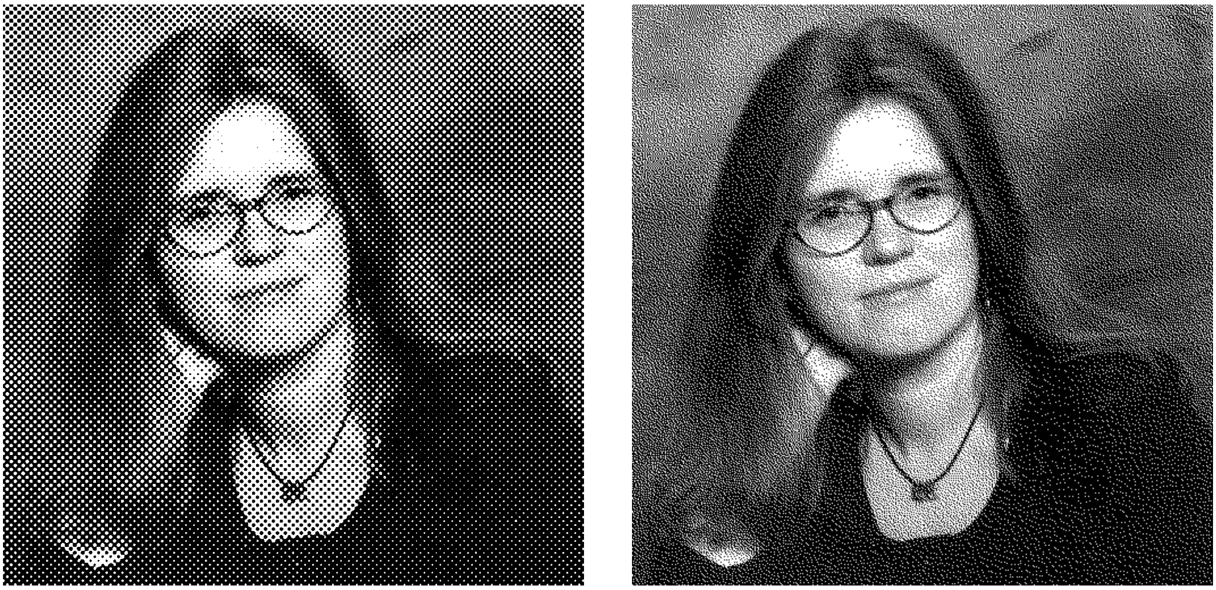


Figure 1.2: Example of halftoning using a clustered-dot dithering (left) and error diffusion (right).

1.3 A Strategy for Stroke-Based Halftoning

This thesis pursues an approach that combines the expressiveness of stroke-based rendering and the efficiency of halftoning to create a new genre of real-time rendering methods that are applicable to interactive applications such as computer games. Like in traditional halftoning, we create images with black strokes on a white background. Such strokes are used for shading, we refer to the ensuing methods as *stroke-based halftoning*. To display strokes for this purpose, we introduce explicit and implicit rendering techniques specifically designed to be accelerated by modern graphics hardware. Explicit techniques determine the geometric extent of strokes before rendering them, while the implicit approaches are based on images that latently contain stroke information which is revealed in the rendering process.

Besides strokes that indicate shading, often strokes are used for displaying contours of objects, too. Because outline strokes are important to the visual style created by stroke-based halftoning, methods to render these outlines are developed. Again, both explicit and implicit techniques are discussed.

Figure 1.3: Example of artistic halftoning [Ostromoukhov and Hersch, 1995]

	implicit stroke generation	**explicit stroke generation**
shading	encoding of strokes in halftone textures and threshold operation by texture combiners	upload of stroke data to GPU and selection by vertex program based on dual threshold values
outlining	rendering of G-Buffers and edge detection filters in texture hardware	silhouette detection by enabling or disabling edges in a vertex program

For all four basic categories, that is for implicit and explicit approaches to both stroke-based shading and outlining, we seek to devise algorithms that employ features of modern graphics hardware. However, if a technique can be implemented on older hardware we prefer this solution to maximize applicability.

1.4 Summary of Results

This thesis devises new terms, concepts, and algorithms for non-photorealistic real-time rendering:

- a number of terms are defined which guide the development of algorithms, in particular stroke-based halftoning, minimal halftoning, implicit and explicit stroke generation, perspective scale, indication map, warp map, silhouette propagation, denormalization filter, contrast-based outlines;
- a view on graphics hardware is presented which exposes features which were designed for photorealism, and which suggests how these can be rededicated to non-photorealistic rendering;
- a method of applying the concept of threshold values as they are used in halftoning to strokes. This is perhaps the key innovation of the entire thesis which yields a break-through in moving from CPU-based stroke computation to graphics hardware based stroke rendering;
- a conceptual differentiation between changes in stroke selection based on perspective scale as opposed to intensity, extending the result of [Winkenbach and Salesin, 1994] who had only one degree of freedom in the selection of their prioritized stroke textures;
- a method to create a constant stroke density via mipmapping;
- a method to create a halftone screen from layers of strokes;
- a smooth threshold operator for real-time halftoning using only one texture unit;
- the use of a secondary texture, for which we coin the term indication map, to define image regions with increased or lowered amount of detail;

- a method to make strokes encoded in a bitmap texture react to lighting individually, which is implemented with texture combiners;
- the concept of the warp-map, which is used to equalize lighting information across all texels in stroke, implemented per dependent texture reads;
- the use of fragment programs to implement procedural pixel-based hatching, which is animated via a noise function;
- a classification method for edges in a polygonal mesh to reduce the number of silhouette edge candidates that need to be tested at run-time;
- a method for attaching surface data to vertices to facilitate silhouette determination on the graphics processor by a vertex program;
- a method to efficiently determine silhouettes in a refined subdivision mesh, for which we coin the term silhouette propagation;
- a fast contour drawing method that subdivides curves instead of surfaces for enhanced rendering quality without the performance impact of full surface subdivision;
- an implementation of the Sobel operator for edge detection on a G-buffer by a special arrangement of texture coordinates and bilinear filtering, which reduces the number of required texture units from six to four;
- a “denormalization filter” that detects edges in a Normal-buffer by observing the denormalization of linearly interpolated normals;
- a method to ensure the contrast of outlines over varying backgrounds;
- The methods have been applied and tested in an archaeological visualization project as well as a computer game.

Several of these contributions have already been published: The use of mipmaps for hatching as well as an explicit outlining method [Freudenberg et al., 2001b], the use of subdivision for drawing outlines [Freudenberg, 2001b], the basic implicit halftoning method [Freudenberg, 2001a] and extensions [Freudenberg et al., 2002], explicit halftoning in the context of stippling [Pastor et al., 2003], and the application in an archaeological visualization [Freudenberg et al., 2001a] as well as in a game engine [Freudenberg and Masuch, 2002]. Since the implicit stroke rendering method is particularly interesting to game developers, it has been accepted for publication in [Freudenberg et al., 2004]. Other publications explored various aspects of non-photorealistic rendering, discussing virtual reconstructions [Masuch et al., 1999; Strothotte et al., 1999], a framework for NPR [Halper et al., 2003], and an overview of silhouette determination algorithms [Isenberg et al., 2003].

1.5 Thesis Overview

First, in Chapter 2, we analyze images that were manually created in traditional media. There are many ways strokes are used for conveying shape, texture, and lighting. The subjects include scientific illustrations as well as comic books, while the techniques covered include woodcut printing, pen-and-ink drawings, and stippling. Cartoons are discussed to serve as motivational aid which reveal techniques useful for animation with strokes.

The analysis shows a wide range of effects that would be useful to employ in an interactive system. The use of strokes for portraying a certain tone serves as archetypical example for the later development of algorithms. Lighting is often found to be physically incorrect, suggesting that an exact reproduction of intensities is not required for an esthetically pleasing image. Also, different styles of outline use are observed that can depend on both the subject and the background.

To facilitate real-time implementations of stroke-based rendering algorithms, Chapter 3 reviews current graphics hardware. The focus hereby lies on PC graphics solutions. We specifically look for features that allow efficient stroke rendering.

Based on the analysis of rendering features available in different PC graphics accelerators, we present a categorization into three generations. While a few features useful for stroke display are available across all graphics processing units (such as line rendering), some were only introduced in more recent generations (such as vertex and fragment programs). The features are discussed to provide a basic understanding of the technology without yet revealing the algorithms introduced later.

Related work in the field of stroke-based rendering is discussed in Chapter 4. Contributions are classified into outline rendering and surface shading techniques with a focus on real-time methods. Furthermore, commercial applications are discussed with respect to their use of stroke-based rendering.

The discussion shows that while many outline rendering techniques have been developed for real-time rendering, surface shading techniques rarely operate at interactive rates. The ones that achieve real-time performance mostly have a high degree of CPU utilization. It is found that games currently employ cartoon shading and silhouette rendering exclusively, other techniques are only available as experimental hidden features. This hints at the desire of game developers to explore novel visual styles, yet they are constrained by the availability of efficient rendering techniques that fit well into the production process in the industry.

In Chapter 5 we introduce new shading methods which produce strokes rather than the usual pixels. In particular we are able to devise a user interface for enabling users to draw their own strokes which are to be used in the halftoning procedures. The strokes are parameterized so as to be able to change their size and density (and thus their apparent intensity as they are used for shading).

Our investigation of such shading in the context of halftoning leads us to define a pair of threshold values which we associate with each stroke (rather than with pixels as would be done in traditional halftoning). Our threshold values are related to the priority values of [Winkenbach and Salesin, 1994], however, by using two independent values we are able to differentiate images more effectively: One of these threshold values is used to select specific strokes so as to reproduce a certain target intensity of the surface, while the other, independent threshold value is used to select specific strokes based on the perspective scale of the object. The crux of the algorithm for implementing these concepts on graphics hardware is the use of vertex programs to manage the translation of these priority values and user-defined strokes into real-time programs for producing the images. As a by-product, stippling images can also be produced using the same basic algorithms.

Chapter 6 now turns to what we have called implicit stroke generation. These are based on textures which contain representations of strokes. We use mipmapping to adjust the stroke density to the perspective scale of an object. We devise a special threshold operation which is applied to pixels but yields results on the stroke level.

The key idea Chapter 6 explores in detail is to use threshold textures mapped onto objects surfaces rather than onto the image plane (as in halftoning) to perform rendering. This then enables us to attain frame coherence without distracting artifacts in the visualization. Next we devise a method of converting stroke textures into threshold matrices automatically. The thresholding approach is computationally inexpensive and enables us to devise the first real-time implementation of indication mapping. Furthermore, individual strokes are made to appear differently depending on the lighting conditions. Special data structures which we called warp-maps can also be implemented in hardware to enable users to render entire strokes in a final image rather than having to deal with individual pixels. Here warp-maps ensure that even if a stroke extends over and above a given region with different illuminations, all pixels of the stroke appear to look identical. The chapter finally examines procedural textures which are implemented as fragment program to convey animated hatching.

We now turn to explicit outline rendering in Chapter 7. An object's outline comprises silhouette edges and feature edges. First we devise a simple method of choosing edges of a polygonal model which are candidates for being parts of silhouettes. This discards edges that will not be displayed, while feature edges are collected to be drawn without a silhouette test. Next the candidate edge data is reorganized for processing on the graphics hardware where a test for whether a given edge is a silhouette or not must be made locally. Finally we study subdivision techniques for silhouette rendering.

Two key ideas emerge from our work with explicit outlines. First, connectivity information for a polygon mesh must be localized so that it can be processed in parallel to determine whether a given edge is in fact a silhouette or not. Second, a particularly convenient organization of geometry is as a subdivision surface to attain efficiency gains for the silhouette determination. To this end we devise a new method called silhouette propagation to speed up significantly the procedure of finding silhouettes on subdivision surface, and we utilize curve subdivision for drawing higher quality outlines.

Implicit outline generation is dealt with in Chapter 8. Here we rely on methods of image processing to compute outlines and build on the concept of G-Buffers [Saito and Takahashi, 1990]. We develop a new method of implementing a Sobel filter which normally requires six samples on hardware with only four texture units. We also propose a new method for finding silhouettes by detecting denormalization induced by linear filtering of an n-buffer. The result is that the Sobel filter produces images of a higher quality than the denormalization filter. However, the latter requires significantly less hardware resources in its processing.

The work with these filters leads us to the new concept of contrast-based outlines. These are ones which are adapted to the background intensity, even when this background changes over the image. We develop two methods, one based on doubling the contours, the other based on choosing the most appropriate color for the outline based on the local intensities of the background over the length of the outline.

Chapter 9 now moves to applications of the theoretical and practical considerations gathered in the course of our research. Two case studies are presented: The first deals with the virtual reconstruction of ancient buildings based on archaeological findings. The system was implemented using a game engine and applies our stroke-based halftoning to produce images which were displayed in a historical exhibition in a museum. The second case study pertains to a computer game in which photorealistic images are normally used but were adapted by us to a non-photorealistic pen-and-ink style.

The result of these case studies is that real-time non-photorealistic rendering can be used to give hints to users about the objects over and above that which can be encoded in photorealistic images. Producing such images in real-time on inexpensive PC hardware means that the images become accessible to everyone on demand. This implies that often the images must "speak for themselves" rather than being accompanied by verbal explanations as are given in videos. Hence moving algorithms to the real-time domain means that the resultant images will be used in different contexts than before, and care must be taken to adapt to this situation accordingly. In the context of computer games, our efficient stroke-based halftoning methods allow novel visual styles to be implemented.

Concluding remarks are made in Chapter 10. We discuss stroke-based rendering methods in conjunction with hardware acceleration and limitations of that approach to the design of algorithms for graphics hardware proposed in this thesis. We finish with comments on future methods of non-photorealistic rendering in interactive environments.

2 Stroke-Based Techniques In Traditional Media

There is a variety of images created with strokes, that is marks made by a drawing tool. This thesis focuses on images where strokes play a dominant role, so the individual strokes are easily discerned in the final image. In this chapter we show a few examples to encourage the reader of this thesis to appreciate the diversity of stroke-based images and to provide a basis for a systematic analysis.

2.1 Examples of Stroke-Based Images

Figure 2.1: "Dragon", woodcut by M. C. Escher. Black and white stripes of varying width indicate shading on the tubular parts, while special patterns are used to depict the scales.

Figure 2.2: “Woman”, by M. C. Escher. Note how strokes are exclusively used to depict shape, not shading. Local changes in width and direction are used to hint at details, and white-on-black as well as black-on-white strokes are used.

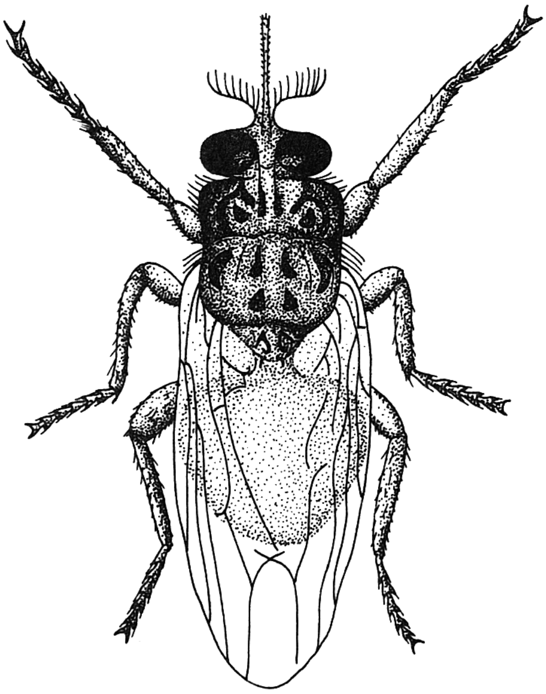


Figure 2.3: Stippled illustration from a biological textbook. Dots are used for shading, while strokes are used for outlines and detail [Crome and Hartwich, 1967].

Figure 2.4: Various patterns indicate surface detail in this pen-and-ink illustration, hatching strokes of varying density are used for shading [Hodges, 1989].

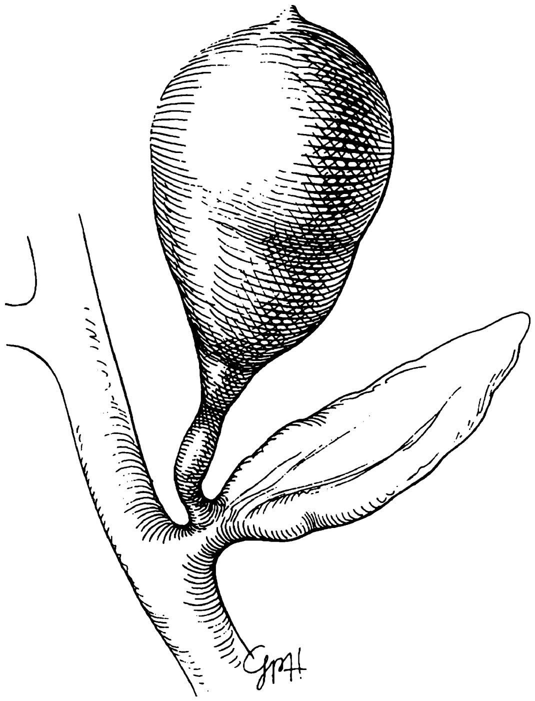


Figure 2.5: Cross-hatching is used in this scientific illustration to indicate various degrees of shading [Hodges, 1989].

Figure 2.6: Hatching as well as black areas are used to indicate shadows and shading in this comic. The tile pattern is drawn in white in the shadow, black in the light, and faded out in the distance [Sclavi and Freghieri, 1997].

Figure 2.7: In these panels from the comic "XIII" the same objects (chimney, roof) are portrayed with varying degrees of detail [Vance and van Hamme, 1984].

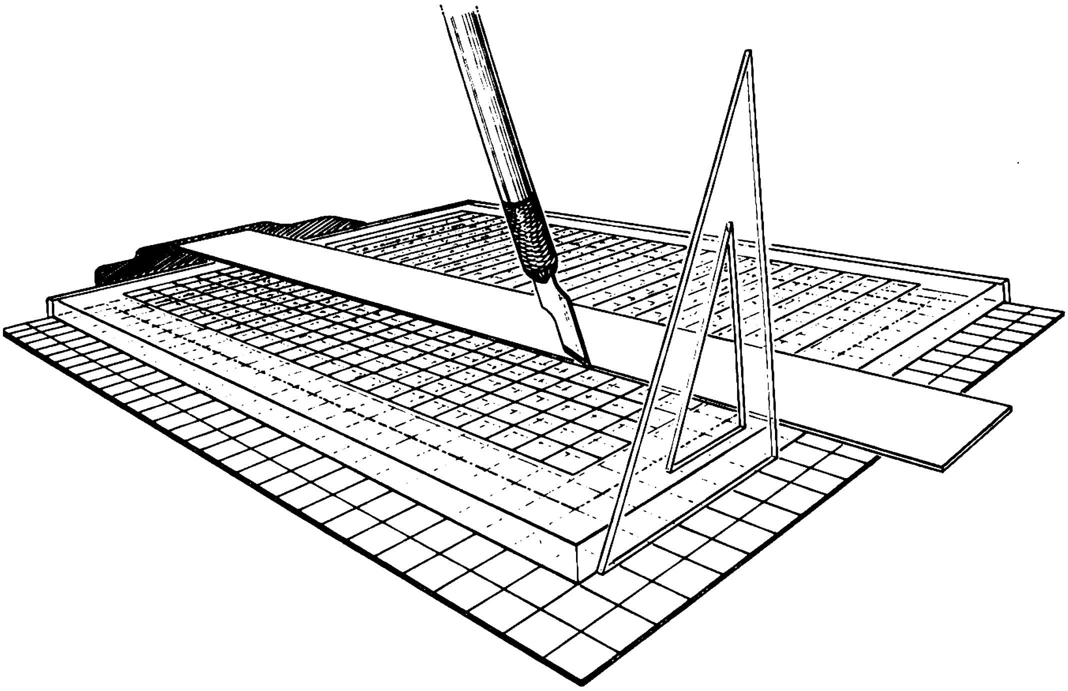


Figure 2.8: In this technical illustration, only strokes of one direction are drawn in the distance. A white border is left around the silhouette of objects [Hodges, 1989].

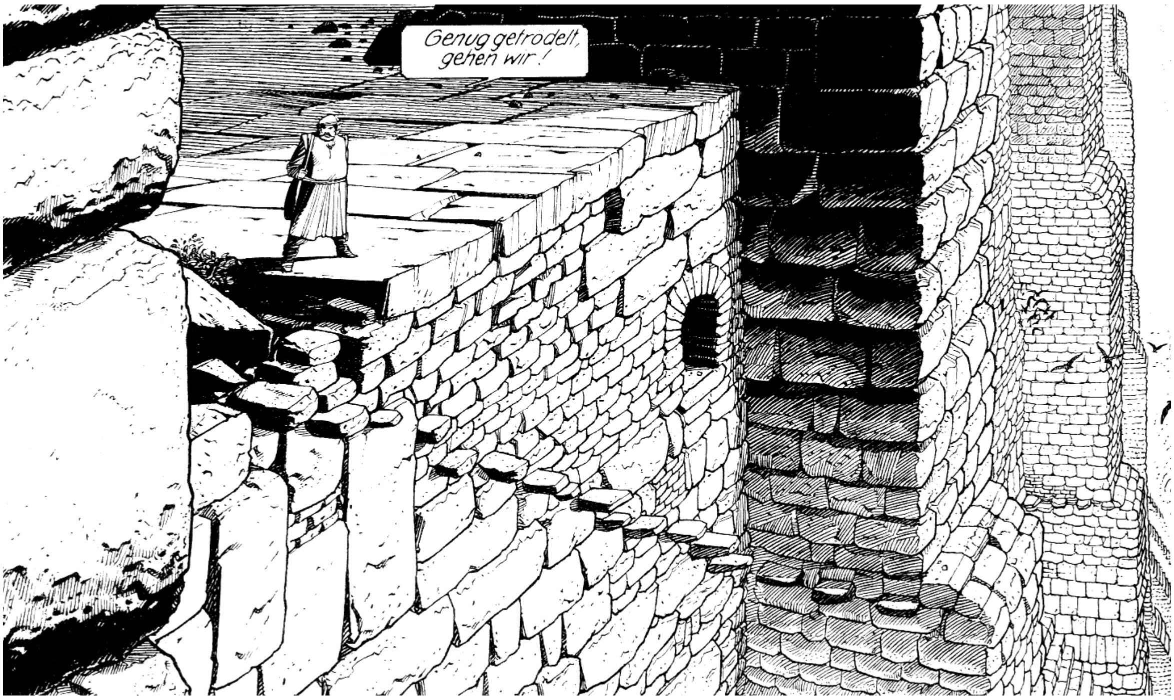


Figure 2.9: The light distribution in this scene from a comic book does not reflect reality but is chosen to create a certain image composition [Schuiten and Peeters, 1994].

Figure 2.10: "The peasant and his wife", engraving by Albrecht Dürer. Very fine strokes indicate shading and surface structure.

Figure 2.11: "Faust", etching by Rembrandt van Rijn. A wide range of intensities is created by many layers of hatching.

Figure 2.12: "Forty-Eight Hour Absence", lithograph by Honoré Daumier. Individual strokes are much less recognizable in lithography.

Figure 2.13: "Dream", woodcut by M. C. Escher. White and black silhouettes delimit the objects.

Figure 2.14: Colored and black borders are used in this drawing from “Akira” [Mash-Room, 1996].

Figure 2.15: Frame from the animated cartoon “Princess Mononoke”, [Miyazaki, 1997]. Discrete shading and simple outlines are used on animated objects, while the background is painted in full color.

2.2 Techniques

Woodcuts, as shown in Figures 2.1, 2.2, and 2.13 are made by drawing onto a block of wood and cutting away all the space around the strokes. Alternatively, the strokes themselves are cut away, creating a negative image. The result is a *relief* of the drawing. To make a print, ink is applied to the surface and pressed on paper. Only ink on the embossed parts is transfered to the paper, creating the final image. Materials other than wood are used for relief techniques, too, such as linoleum to create linocuts.

In contrast to relief printing, in *intaglio* printing strokes are cut into the surface and ink is applied to the grooves, while the other surface parts are wiped clean. When paper is put on top of the surface and pressure is applied, it picks up ink from the grooves. Figure 2.10 shows a copper engraving which is an example of this process, other intaglio techniques include etchings (as shown in Figure 2.11) and mezzotints.

A third major printing technique is *surface* printing, which uses a flat surface. Lithography, for example, works by drawing with greasy ink on limestone. For printing, the stone is moistened and rolled with water-repellent ink, which will only adhere to the greasy parts. By applying pressure, the ink is transferred to paper (see Figure 2.12). The same basic principle is used in modern offset printing, with the major difference that the image is reproduced photographically onto the printing surface.

Other techniques create strokes on paper without an intermediate printing surface. Figures 2.3 to 2.9 show examples of pen-and-ink drawings. A pen is used to apply ink directly onto the paper, the resulting drawing is the final image. Many other drawing implements can be used to put strokes on paper. However, most of them do not create such a crisp rendition of each stroke like pen-and-ink. Not only can strokes be created by this technique, but also dots, like in the stippled Figure 2.3.

Traditional animation rarely uses strokes to indicate shading. Much more common is the use of outlines as shown in Figure 2.15. Different drawing techniques are used for animated characters and inanimate backgrounds.

2.3 Lighting Effects

In these images, we can see lighting being depicted by two basic means. One is the varying density of strokes, the other varying the width while keeping the density constant.

Examples of the density variation are found in the cross-hatching of Figures 2.5, 2.10, and 2.11. Multiple series of strokes are drawn at an angle to create increasingly dark regions. In Figure 2.9, there is only one series of strokes, which creates an intermediate tone that is put in contrast to otherwise almost completely white and black regions. Dots are used instead of elongated strokes in part of Figure 2.3.

The stroke width is varied most prominently in the woodcuts shown in Figures 2.1 and 2.13. The stroke density in these images depends on the subject and distance, but not on the lighting.

Positive and negative strokes are sometimes used in the same image, like in Figure 2.2. In Figures 2.6 and 2.9, white strokes show outlines in shadowed areas.

The lighting in such images is not necessarily accurate. Specifically, in Figure 2.9 it is hard to imagine that the shading is correct, as it is mostly dark on top of the wall and becomes lighter below and in the distance. Hardly this appearance can be caused by daylight. We think the intensities were chosen this way to create an interesting image composition, rather than to reflect reality in this imaginary scene.

2.4 Perspective Effects

The patterns created with strokes are not only varied according to the intended intensity of the image, but also sometimes depend on the distance or size of an object.

For example, in Figure 2.6 the lines marking tiles on the street fade out in the distance, first the series receding into the background, then the other series parallel to the horizon, too. The hatching strokes on the tiles in the foreground are not drawn on tiles further away.

In Figure 2.7 we see the same house drawn from multiple distances. In the close-up, individual bricks are portrayed on the chimney and wooden planks on the gable are indicated by coarse hatching. These surface details are not shown when the house is viewed from a distance.

One can argue that not distance to an object is the most important factor for choosing a drawing style in these settings, but rather the actual size of the object in the image. We think this is indeed the case, because this encompasses objects of different scale, as well as objects appearing smaller because they are tilted with respect to the viewer, so the actual size in the image is much smaller than if it was viewed frontally.

2.5 Surface Detail

Depicting the shape by indicating lighting is not the only purpose for strokes on surfaces. Depicting the material is similarly important. For example, Figure 2.4 uses different patterns of strokes to show different kinds of feathers on the hummingbird. In Figure 2.9 rough stones are indicated by irregular strokes, while Figure 2.1 uses multiple patterns distinguish between the dragon's body parts.

2.6 Outline Effects

Most of the images shown in this chapter use outlines to delineate objects and indicate features on the surface. The notable exception is Figure 2.2, which consequently gets a very “open” look.

In the figures set on a white background, black outlines are used. If stark contrasts are part of the image, white is used in the black parts, like in Figures 2.6 and 2.9. Sometimes double contours are used which outline both in black and white next to each other, like in Figures 2.13 or 2.14.

2.7 Summary

In summary, we see that there are a variety of stroke-based drawing techniques that would be useful to employ in an interactive system. Shading can be portrayed by varying the density or width of strokes. The stroke density does not only depend on lighting, but also on the perspective size of an object. The lighting is not necessarily accurate, so an exact reproduction of tone like in photorealistic rendering is not a first-hand goal. Apart from shading we have observed the use of outlines in these images, and how they are used to delineate objects.

3 Graphics Hardware

Fast rendering of virtual scenes is necessary for achieving interactivity. Graphics hardware has been greatly improving rendering speed over the last years, especially in the consumer-oriented PC sector. In fact, even professional image generators are now based on COTS (commercial-of-the-shelf) technology [ATI, Evans & Sutherland, 2001]. However, to most efficiently use the hardware, applications have to be specifically tuned to make use of the hardware's capabilities.

In this chapter, we will review the graphics hardware features that are employed for fast halftone rendering in this thesis. The actual techniques developed will be described in later chapters.

3.1 Scope and Categorization

The discussion of real-time halftoning techniques in this thesis is oriented on what common graphics hardware in PCs provides nowadays. This is motivated by the desire to create methods applicable in interactive applications such as computer games. In particular, a game for the PC market needs to support a wide range of hardware – if a game relied on techniques that exclusively work on the very latest hardware, the potential customer base would be severely reduced. Only now in the end of 2003 the first game titles will be released that actually *require* at least a first-generation GPU (graphics processing unit) and will not work on lesser hardware. We will categorize GPUs in 3 generations, with the first one being introduced in 1999, the same year that preliminary work on this thesis started.

The first generation of GPUs was introduced in 1999 with NVIDIA's "GeForce 256" graphics accelerator, followed by ATI Technologies' "Radeon" in 2000. Boards of this generation accelerate the rendering pipeline starting from geometry transformation and lighting, a task that until then was left to the CPU. Additionally, they support the flexible combination of multiple textures.

Second-generation GPUs were introduced in 2001 with NVIDIA's "GeForce 3" and ATI's "Radeon 8500". They made the vertex processing stage programmable and offer more powerful texture operations, like dependent texture reads.

The third, and current, generation of GPUs supports programs not only on the vertex level, but on the fragment level, too. Boards of this generation were introduced in 2002 and include 3Dlabs' "Wildcat VP", ATI's "Radeon 9700", and NVIDIA's "GeForce FX".

The pace of this advance seems to be slowing down, there is no public estimate yet when the next generation of GPUs will be introduced, nor what features they will provide. A feature warranting to speak of the fourth generation would be support for programmable primitive processing.

Formally, we define boards as belonging to a specific generation based on the natively supported OpenGL extensions[1]:

first generation: GL_ARB_texture_env_combine

second generation: additionally, GL_ARB_vertex_program and dependent texture reads[2]

third generation: additionally, GL_ARB_fragment_program

The techniques in this theses were developed and tested on boards by 3Dlabs, ATI, and NVIDIA. There are a handful more companies in the PC desktop graphics market, but they are not as influential as these three at the time of this writing.

There are two low-level graphics APIs in use today that provide software access to PC graphics hardware. The *OpenGL* API is available on multiple platforms, including Apple MacOS X, GNU/Linux, and Microsoft Windows, while *DirectX* is only supported on Windows. Throughout this thesis OpenGL terminology is used. Since DirectX provides similar functionality, all techniques would be implementable with it as well.

3.2 Hardware for Non-Photorealistic Rendering

The commonly available graphics hardware is optimized towards photorealism. Both major hardware vendors in the game-oriented PC graphics board sector are touting "cinematic rendering", and they do not have cartoons in mind. Rendering large, detailed scenes with many polygons, realistic lighting, lifelike animations, and a high image quality are high-priority items on the wish list of game developers and consumers.

To implement non-photorealistic rendering techniques on this hardware, creative use has to be made of the features found in graphics processors today. Of major assistance in this is the increasing flexibility of the processors. Certain parts of the graphics pipeline are not hard-wired anymore to accelerate a fixed function, like evaluating the Phong lighting model or modulating a texture with a lighting result. Instead, these parts are made configurable to perform many functions, or even programmable with a custom program. These units can be rededicated to perform non-photorealistic functions.

Other techniques are shared between photorealistic and non-photorealistic rendering. Since the very same hardware is used, the optimization methods to achieve real-time performance are similar. There are many sources providing information on optimizing

---

[1] The OpenGL hardware registry provides a database of boards and their supported extensions [Nuydens, 2003].

[2] There are only vendor dependent OpenGL extension for dependent reads, see Section 3.5.1

graphics performance, including the developer-relation web sites of hardware vendors and books (for example, [Akenine-Möller and Haines, 2002]).

In the following, we will describe the hardware features that are utilized in this thesis. The discussion is structured along the four major sections of the general pipeline model of graphics processors: The *geometry* is transmitted from the CPU to the GPU. Operations on its individual *vertices* are performed. After rasterization, processing continues on *fragments*, which are finally stored as *pixels* in the frame buffer (see Figure 3.1).

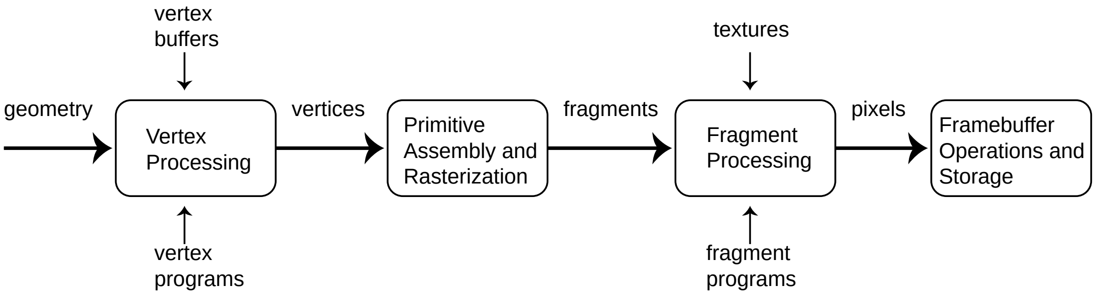


Figure 3.1: Graphics processing pipeline

3.3 Geometry Transfer and Primitives

The geometric data needs to be transfered from the CPU to the GPU. It is used to assemble *primitives*, like triangles, quads, lines, or points.

3.3.1 Vertex Buffers

One performance bottleneck in real-time rendering is the transfer of vertex data to the GPU. One way to increase the performance is to cache vertex data in memory that is efficiently accessible by the GPU. This is enabled by using so-called vertex arrays or vertex buffers. While in the course of this thesis vendor-specific vertex array extensions have been used, there is now a standard ARB_vertex_buffer_object extension, which also was promoted to the core specification of OpenGL 1.5.

If the data stored in a vertex buffer can be reused over multiple frames, the CPU is relieved of the task to continually submit geometry to the GPU. This is especially relevant for games, which can use those additional CPU resources to simulate the game environment, generate sound, or other computation-intensive tasks.

However, without additional measures this is only applicable to static geometry. If dynamic changes are required in each frame, the geometric data must be adapted continuously, which usually requires CPU involvement. To create dynamic behavior from static data on the GPU, vertex programs can be used.

3.3.2 Primitives

In photorealistic rendering, *triangles* are the universal primitive type that is used almost exclusively. For increased efficiency, triangles can be submitted as strips, fans, or indexed lists, which reduces the amount of redundant vertex transmissions. Triangles are usually not disconnected, but form a triangle *mesh* which describes an entire surface.

Triangles and meshes still play a major role in non-photorealism as the basic drawing primitive, especially since their appearance can be modified by programming subsequent pipeline stages. However, particularly in stroke-based rendering, parameterized primitives are manipulated and drawn individually. Contrary to triangles as surface elements, their on-screen extent often is not strictly determined by the laws of perspective projection. Rather, a fixed on-screen size might be desired, such as brush strokes of a specific width.

The OpenGL API supports such primitives in a basic form. These primitives are *lines* and *points*. A line is specified by two vertices and a line width in pixels, a point by one vertex and the point size, also in pixels. Width and point size are independent of the depth of the line, thus breaking the strict perspective. This makes them amenable for stroke-based rendering.

However, lines and point primitives are quite limited in their expressiveness. Line widths and point sizes cannot be chosen freely, but are limited by the driver to discrete sizes and maximum extents (see Section 3.7 for exemplary limits). The line width is constant for all lines submitted in one call, thus tapering is not possible. A texture is only applied in the direction of the line, not over its width; a point is textured with only one color sampled at its center. Furthermore, a point is clipped like a vertex, independent of its size. This means as soon as the point center is clipped, the whole point disappears.

For points, extensions exist to allow some more flexibility. The point size can be manipulated by a vertex program, although it still is clamped to the implementation-specific maximum. Correct two-dimensional texturing can be achieved through the ARB_point_sprite extension.

No such extensions exist for lines, though. This is unfortunate, because in practice this means lines have to be emulated by quads, which require twice the number of vertices. If future hardware supports a programmable primitive processor, it might be possible to more flexibly assemble parameterized lines.

3.4 Vertex Processing

In first-generation GPUs, vertices are processed by a fixed set of calculations. While this yields an improvement in performance because the CPU is relieved of the transformation and lighting load, it also restricts the flexibility to perform arbitrary computations. To combine freedom of implementation with the performance benefits of the GPU, the second generation of GPUs introduced a programmable vertex processing unit.

3.4.1 Fixed Function

Traditionally, the vertex processing stage transforms vertices from modeling coordinates to screen coordinates, and performs lighting calculations. It takes per-vertex data like positions, normals, and texture coordinates in combination with state variables like the transformation matrixes and material parameters, and computes transformed coordinates, colors, and texture coordinates. The exact formulas used for these calculations are defined in the OpenGL specification [Segal and Akeley, 2002].

The flexibility of this approach is severely limited to the parameters intended by the hardware designers. It basically only allows to adjusted the transformation matrixes and parameters of the Phong lighting model.

3.4.2 Vertex Programs

To allow more flexible vertex processing, second-generation GPUs support *vertex programs* [Lindholm et al., 2001]. A vertex program replaces the fixed-function vertex transformation and lighting stage by a programmable vertex processor. It operates on individual vertices, forbidding access to connectivity information or neighboring vertices. This enables a highly parallel hardware implementation.

A vertex program can consist of typically at least 128 instructions operating on 4-component vectors. Program inputs are the attributes submitted with a vertex as well as “constants”, which are parameters set by the application. The program’s output includes position, color, and texture coordinates. Operations include move, addition, multiplication, dot products, minimum/maximum, reciprocal, square root, and logarithms / exponentials. Comparison operators can be used, but no branching is supported in the original implementations (although recent GPUs start to supporting that, too). Vertex programs are written in assembly language [Lindholm et al., 2001], or using a high-level language like Cg [Mark et al., 2003] or the OpenGL shading language [Kessenich et al., 2003].

The programmability of the vertex unit allows it to perform non-traditional calculations, which makes it very apt for implementing non-photorealistic methods.

3.5 Fragment Processing

After the vertex processing stage, clipping is performed and primitives are rasterized producing fragments. Fragments are associated with attributes that are calculated by interpolating the outputs of the vertex program stage. The color of a fragment is derived in the *fragment processing* stage from these attributes before it is written into the frame buffer.

3.5.1 Textures

Textures are the main vehicle to create surface detail in real-time rendering today. Pure geometrical detail would overload even the fastest GPUs. Textures not only provide a balance between the load of vertex and fragment processing, but are also an efficient medium for artists, because painting a texture usually takes less effort than modeling. While basic texturing was already part of the first OpenGL standard (released in 1992), the capabilities have greatly been refined and extended over the last decade.

Texture Filtering

To determine the color of a texture for a given fragment, the texture coordinates associated with the fragment are used to sample the texture map at the specified location. Various filtering methods are provided. In the simplest case, just one texel is sampled, the one "nearest" to the exact texture coordinate. In "linear" filtering mode, a linear interpolation is performed from the four samples adjacent to the exact location.

An interesting filtering mode uses "mipmaps". A series of differently-sized textures is provided, and the hardware automatically chooses a level of detail that best matches the actual screen extent of the texture, to diminish sampling artifacts. Besides simply choosing a mipmap level, linear interpolation between levels is possible, too, which is called trilinear mipmapping.

The mipmapping feature was originally designed to work with prefiltered texture maps. Indeed, the OpenGL Utility Library contains functions to automatically build mipmaps (gluBuild2DMipmaps() etc.) by filtering, as does the SGIS_generate_mipmap OpenGL extension, which was promoted to the core specification of OpenGL 1.4. However, the application is free to upload any image into the individual mipmap levels, if it was generated by filtering or by other means. In conjunction with the fact that a roughly constant texture resolution is always displayed, this is a potentially very useful feature for stroke-based non-photorealism, since a stroke needs always to be displayed at roughly the same size, too.

Another filtering mode is anisotropic filtering, which takes into account the z-slope of rendered surfaces. Mipmapping is conservative, which means it chooses a level based on the minimal extent of the surface. This leads to a lower level chosen than necessary for displaying full detail if the surface is tilted (not parallel to the screen). Anisotropic filtering takes more samples in the direction of higher minification, thus allowing higher-level mipmaps to be displayed even with larger distortions. The overall look is sharpened. However, the actual sampling is implementation-dependent, so the on-screen result is less predictable than with simple trilinear filtering.

Configurable Multi-Texturing

One great enhancement of first-generation GPUs over previous graphics accelerators is the fine-grained configurability of multi-texturing. It means that not only can multiple textures be applied in one rendering pass, but that the results can be combined in a flexible way.

Multi-texturing can be configured by a variety of OpenGL extensions. The cross-vendor ARB_texture_env_combine extension allows addition, subtraction, multiplication, and interpolation of the texture of each multi-texturing stage with the fragment color, a constant color, or the output of a previous stage. Additionally, the result of a stage can be scaled by a 2.0 or 4.0. It is enhanced by ARB_texture_env_crossbar which allows to access not only the texture of the current stage, but others, too. Another enhancement is ARB_texture_env_dot3 which permits to use a dot product operation, thus enabling the evaluation of lighting models on the fragment level (known as per-pixel lighting).

These OpenGL extensions are the lowest common denominator of the features of individual graphics boards. They are universally available and have been made part of the core API in OpenGL version 1.3. By using vendor-specific extensions there are even more possibilities, at the price of portability.

Dependent Texture Reads

A special texture addressing mode introduced by second-generation GPUs is the dependent texturing. Here, the texture coordinate used for a texture lookup comes itself from previous texture lookup. That means, for example, the green and blue color components from the first texture stage are used as s, t texture coordinates for the second texture stage (see Figure 3.2 for an illustration).

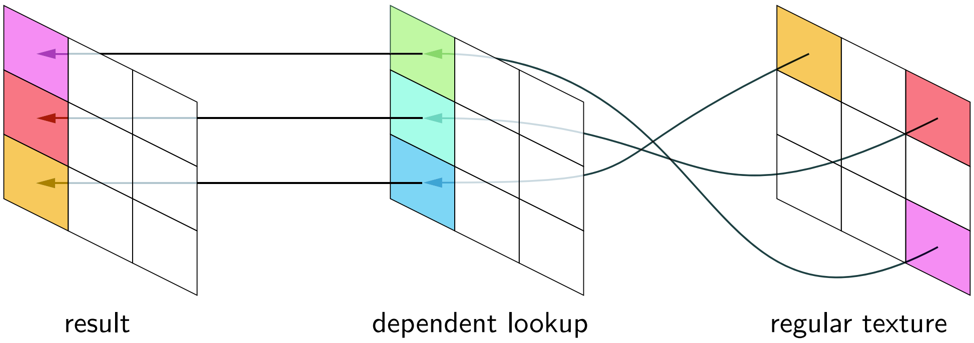


Figure 3.2: The color values in a texture stage configured for *dependent lookup* are interpreted as texture coordinates for a subsequent stage.

Unfortunately, there is no multi-vendor OpenGL extension to utilize this functionality on second-generation hardware. Vendor specific extensions providing access to dependent textures are NV_texture_shader and ATI_fragment_shader. On third-generation hardware, this is generalized by fragment programs.

3.5.2 Fragment Programs

Fragment programs bring the same level of programmability to the fragment level as vertex programs for vertices. They operate in floating point precision of at least 16 bits (typically 24 or 32) per component, which allows a lot more accuracy than the 9 or 12 bit integer operations on previous hardware. In addition to arithmetic operations there are texture sampling instructions and access to screen-space derivatives (to allow LOD techniques similar to mipmapping).

Fragment programs provide a generalization of previous techniques such as the configurable multi-texturing and dependent texturing. Furthermore, textures can be synthesized on the fly ("procedural textures").

The programming model is similar to the one mentioned for vertex shaders. Besides an assembly-language interface provided by ARB_fragment_shader, high-level languages like Cg and GLslang can be used.

3.6 Framebuffer Techniques

Colored fragments are subject to various tests, like depth test, stencil test, etc. which can discard the fragment. If not discarded, the fragment is written into the frame buffer, possibly merging it with the pixel already in there.

3.6.1 Blending

If the incoming fragment should not just replace the frame-buffer content, blending needs to be performed. The OpenGL specification lists a number of blend modes, which basically enable a linear interpolation between the color in the frame buffer and the color of the fragment, based on both the fragment's and the frame-buffer's alpha component.

3.6.2 Off-Screen Rendering

Besides rendering to the screen, off-screen areas can be used as a render target. It can be utilized for multi-pass techniques, which put the result of such a rendering pass into a texture. This enables, for example, fragment programs to use the frame buffer contents as input. Off-screen buffers are created in OpenGL using a window-system specific pbuffer extension.

3.6.3 Anti-Aliasing

There are multiple ways to realize anti-aliasing in the rendered image. *Smoothing* generates a coverage value for pixels on the edge of each rendered primitive (polygons, lines, points), which is multiplied by the fragment's alpha value. When blending is enabled, this creates smooth outlines of primitives.

With *Full Scene Anti-Aliasing* (FSAA), the scene is rendered in a higher resolution and the sub-pixels are sampled down. This *super-sampling* affects the LOD selection process in mipmap filtering, which has to be taken into account by the application if some specific behavior is needed. *Multi-sampling*, on the other hand, creates only one texture sample for all sub-pixels, which leaves the LOD intact. Most modern boards support multi-sampling in addition to super-sampling.

3.7 Graphics Hardware Capabilities

The following table will summarize features and limitations of the graphics boards used in this thesis. We think it represents the current PC stand-alone graphics market well, where NVIDIA and ATI hold a market share of more than 90 per cent in second quarter of 2003 according to Mercury Research. To accurately represent the graphics hardware market, many more vendors would have to be listed. Besides smaller vendors, notably Intel is missing which has a large overall market share due to the graphics processors integrated in its chipsets. 3dlabs is not active in the consumer market, however, it was the first vendor providing us with third-generation hardware and a high-level shader compiler in 2001. A user-contributed overview of OpenGL characteristics of many graphics boards is available at [Nuydens, 2003].

Vendor	**NVIDIA**			**ATI Technologies**			**3Dlabs**
Chip code	NV1x	NV2x	NV3x	R1xx	R2xx	R3xx	P10
Year of Introduction	1999	2001	2002	2000	2001	2002	2002
GPU Generation	1	2	3	1	2	3	3
Vertex Processing							
Vertex Shader ops	sw	128	256	sw	128	256	256
Point size range	1.0–64.0	1.0–64.0	1.0–64.0	0.5–10.0	1.0–2047.0	1.0–64.0	0.7–64.0
Point size granularity	0.125	0.125	0.125	0.125	0.125	0.125	0.125
Line width range	0.5–10.0	0.5–10.0	0.5–10.0	0.5–10.0	0.5–10.0	1.0–64.0	1.0–15.5
Line width granularity	0.125	0.125	0.125	0.125	0.125	0.125	0.25
Fragment Processing							
Textures per pass	2	4	16	3	6	8	8
Arithmetic ops	2	8	1024	3	12	64	128
Bits per component	int 9	int 9	float 32	int 9	int 12	float 24	float 32
Max. texture size	2048	2048	4096	1024	1024	2048	4096
Dependent Texturing	no	yes	yes	no	yes	yes	yes
Product Names (not exhaustive)							
	GeForce 2, 4 MX	GeForce 3, 4 Ti	GeForce FX	Radeon 7500	Radeon 8500	Radeon 9700	Wildcat VP

4 Real-Time Non-Photorealistic Rendering

This chapter will give an overview of the history and state of the art in real-time non-photorealistic rendering. There are two fundamentally different visual components in non-photorealistic images, and two basic methods of rendering them.

The two basic image elements are *surfaces* and *contours*. Surfaces are common to both photorealistic rendering and non-photorealistic rendering. They depict visible parts of objects in the scene while occluding hidden parts. Contours, on the other hand, are unique to non-photorealism and do not have a counterpart in photorealism, but rather in the visual arts. They provide visual cues about how objects are separated from each other or emphasize features on surfaces.

Rendering approaches can be classified into *analytic* and *discrete* methods. Analytic methods use geometric calculations to explicitly determine the shape of image elements before drawing them. By contrast, discrete methods uniformly operate on pixels while the final shape only emerges through the rendering process itself.

An important notion in non-photorealism is the rendering of *marks*. While in photorealism the color of each pixel is determined by a projection according to optical laws, non-photorealistic methods often use image elements that are merely guided by the projected image of an object. Contour lines are one example of marks, but marks are also used for the rendering of surfaces.

Traditional (photorealistic) real-time rendering typically combines analytic methods for geometric transformation with discrete hidden surface removal and shading. Modern graphics hardware is optimized for this combination. In particular, there is very little support for mark-based rendering, except for basic line and point drawing. Thus, special measures have to be taken to utilize graphics hardware for non-photorealistic rendering.

While many NPR systems combine specific techniques for rendering surfaces and contours, most approaches can be freely intermixed. The following discussion will therefore analyze the methods separately, even if they were originally presented in combination with one another. After general mark rendering techniques have been discussed, contour rendering techniques will be presented, followed by surface rendering techniques.

4.1 Mark Rendering

Fast rendering of marks is a challenge in non-photorealism because current graphics hardware provides only very limited direct support. Since photorealistic methods are the primary focus of three-dimensional real-time rendering today, hardware is optimized for high-quality rendering of surfaces by filling polygons after projecting them from world space to the screen.

The only primitives available on current graphics hardware that are not filled polygons are lines and points. Lines are rendered by projecting their two vertices into screen space and rasterizing a constant width line between them. The line width is limited (typically to 10 pixels). Points are rasterized as filled circles and also have limited diameter (64 pixels). Surfaces and lines are cumbersome to combine accurately because different rasterization rules are used (see Figure 4.1). However, for reasons of speed and simplicity, hardware-accelerated lines are widely used in NPR systems.

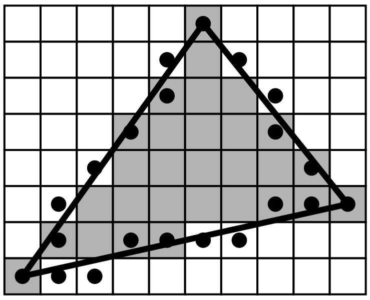


Figure 4.1: Different rasterization rules are used by graphics hardware for filled polygons and lines. Black dots indicate fragments generated by line rasterization, gray squares are pixels filled by rendering polygons. Note that line fragments are produced both inside outside of the rendered triangle.

A very simple, yet effective technique is used in Zeleznik's SKETCH system [Zeleznik et al., 1996]. Silhouettes and feature edges are drawn as multiple jittered lines, while the depth buffer handles the occlusion. The rough appearance of the resulting images that change from frame to frame hides the fact that occlusion is not always handled correctly.

Markosian et al. use a path defined by a sequence of connected edges in screen space as basis for various drawing styles. This path is perturbed by vector offsets in tangent-normal space, which also can have discontinuities. The result is drawn as line segments. A second method constructs a polygon mesh by adding a crossbar at each vertex, defining a series of quadrilaterals which are drawn with texture mapping [Markosian et al., 1997].

4.2 Contour Rendering

4.2.1 Polygon Edging

Given a polygonal model, *polygon edging* techniques draw all edges of each polygon [Herrell et al., 1995]. No distinction is made between inner or outer contours. A typical application is in a modeling package, where polygon edging is used to emphasize the polygonal structure of the model. In itself, these techniques provide little esthetic appeal, except for giving renditions a rather “technical” look. However, the approaches developed for high-quality, artifact-free polygon edging can be applied in more advanced line-rendering techniques.

One of the earliest forms of three-dimensional computer graphics is the *wire-frame* rendering. All edges connecting vertices in an object are projected onto screen-space and are drawn with a constant line width. Both projection and drawing can be accelerated by common graphics hardware. The object does not appear solid, though, because no surfaces are rendered that would hide lines.

A *hidden-line* view can be achieved by first drawing all surfaces into the depth buffer and then drawing the wire-frame rendering with depth tests enabled. However, because depth values of polygons and lines are calculated differently, “depth fighting” occurs which leads to the unwanted overwriting of some edge pixels. This can be countered to a certain extent by offsetting depth values so that polygons are always further away than the edges [Rossignac and van Emmerik, 1992]. There still remain cases where artifacts occur, for example lines poking through surfaces which are very close to one another, but for controlled scenes the offset values can usually be tweaked so no artifacts remain. The advantage of the offset method is its direct support by common graphics hardware.

For the general case, research has been ongoing. Kurt Akeley describes an algorithm using “hollow polygons” [Akeley, 1990]. This is a special rendering method on SGI PowerVision hardware that only fills those pixels of a polygon adjacent to its perimeter. Because this yields exactly the same depth values as would be generated with a filled polygon, it can be used to draw a perfect outline. Another formulation of this technique uses the special depth buffer modes found in SGI’s GT/GTX series, which can be implemented using the “reference plane” OpenGL extension found in modern workstation hardware. The reference plane can be used to force depth equality of fragments from polygon filling and line rasterization. However, none of these special features are supported by today’s common hardware.

An approach using the stencil buffer as “edging plane” that keeps track of pixels filled by a polygon was proposed by [Herrell et al., 1995]. First, a polygon is filled, marking the corresponding pixel in the stencil buffer. Next, the edges are rendered with depth comparison, but if the depth test fails although the edging plane indicates this fragment belongs to the polygon, it is let through. Finally, the stencil buffer is cleared for the next polygon.

Wang et al. observe that Herrel’s algorithm achieves good results in general, but fails in the case where not all polygons are edged or are partly hidden by others. They propose a “polygon buffer” method that uses the occluding relations between polygons [Wang et al., 1999]. Polygons are first rendered to an ID-buffer using standard depth tests. In this pass, the occluding relation (polygon A occludes B) is recorded in a sparse table. When the edges of a polygon N are drawn, its *id* is tested against the *id* in the polygon buffer (P). If $N = P$, the pixel is drawn, otherwise it is only drawn if polygon P does not occlude N according to the occlusion table. The main drawback of this method is that it does not make use of hardware acceleration (see Figure 4.2).

Figure 4.2: Three methods for polygon edging: edging-plane method (left, showing stitching and exposed rear edges), glPolygonOFfset method (middle, no stitching but still exposed rear edges), and polygon buffer method (right, no artifacts) [Wang et al., 1999].

4.2.2 Edge Selection

In most cases it is not desirable to unconditionally draw all edges in a model. Many edges in a polygonal object are just artifacts from approximating a free-form surface by polygons. If the object is adaptively tessellated, its appearance changes. Sometimes only the outer *border* should be shown, which separates the object from other objects or the background. In other cases, the edges separating front-facing parts from back-facing parts of the model are desired, which are commonly called *silhouettes* [Isenberg et al., 2003]. In addition, certain *features* like sharp valleys and ridges are often marked by lines. All these edges together form the *outline* of an object.

Borders are a subset of silhouettes, and both are view-dependent. This means they have to be recalculated for each frame. Feature edges can be automatically classified or manually selected. For automatic classification, the angle between the two polygons adjacent to an edge is determined in a pre-processing step. If the angle is larger than a certain threshold, the edge is marked as feature edge. Especially for coarsely tessellated models, the automatic classification often produces unpleasing results. In these cases edges are usually tagged as feature edges when the object is modeled. However, if the object is animated, automatic reclassification has to be performed for each frame (for example, wrinkles that occur when an object is bent).

4.2.3 Silhouette Determination

For a polygonal model, silhouettes are defined as the edges shared by both a front-facing and a back-facing polygon. For simple scenes, the straight-forward approach of testing every edge is sufficient. On modern hardware supporting vertex programs, this test can be performed entirely on the graphics board (see Section 7.2).

A memory-efficient structure for real-time rendering of silhouettes and feature lines is the "edge buffer" [Buchanan and Sousa, 2000]. For each edge, flags are stored that record front and back-facingness of the adjacent polygons (initialized to 0), as well as a bit indicating a feature edge. At run-time, all polygons are rendered first. For each polygon, its facingness is determined and the corresponding bit in the adjacent edges is inverted. When an edge is part of two front-facing polygons, the front bit will have been inverted twice, so it is 0 again. After all polygons have been rendered, the edge buffer is traversed and edges with bits that are not zero (that is, silhouettes and feature edges) are drawn.

The rapid differentiation between front-facing and back-facing regions in a model can be accelerated by special pre-computed data structures. Gooch et al. store edges in a hierarchically subdivided unit sphere [Gooch et al., 1999]. Each edge is represented as arc on the sphere spanned between its adjacent polygons' normals. To determine silhouette edges at run-time in orthogonal projection, a plane perpendicular to the viewing direction through the sphere's center is checked for intersection with the stored arcs. This selects all arcs connecting a front-facing with a back-facing polygon's normal. For increased efficiency, a cube can be used instead of a sphere [Benichou and Elber, 1999]. With perspective projection, a larger area of the sphere or cube has to be searched. This can be circumvented by storing the tangent-plane duals of surfaces in a hierarchically subdivided 4-dimensional hypercube [Hertzmann and Zorin, 2000]. Finding silhouettes is accomplished by intersecting the 4D plane that is the dual of the viewpoint with the hypercube (see Figure 4.3).

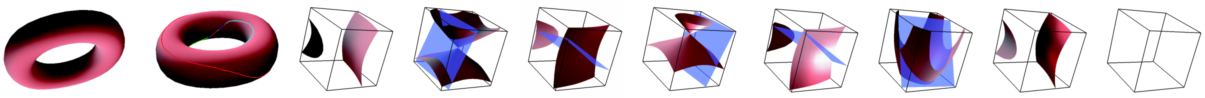


Figure 4.3: Silhouette lines under the duality map correspond to the intersection curve of a plane with the dual surface. Left: Torus shown from camera and side views. Right: The eight 3D faces of the hypercube, seven of which contain portions of the dual surface. The viewpoint dual is shown as a plane. Silhouettes occur at the intersection of the dual plane with the dual surface [Hertzmann and Zorin, 2000].

If it is acceptable that not every silhouette edge is drawn in every frame (for example, when using a sketchy rendering style), a probabilistic approach can be employed [Markosian et al., 1997]. In this method, only a stochastically chosen set of edges is tested. If a silhouette edge is found, adjacent edges are tested, too. This method is used for finding long, connected chains of silhouettes.

When subdivision surfaces are used as modeling primitive, another hierarchical method for silhouette determination can be applied. This "silhouette propagation" method determines silhouettes for the coarse mesh and propagates silhouettes down when subdividing. Thus, not all edges have to be tested (see Section 7.3.2).

Image-based methods can be used to render visible silhouettes. After filling the polygons and thus initializing the depth buffer, the model is rendered in wire-frame with wide lines, but offset away from the observer. The wire-frame lines will only be visible at discontinuities in the depth buffer, depicting the object's silhouette [Rossignac and van Emmerik, 1992]. A problem with this method is that only one half of the line is visible, because the other half is hidden "inside" the object. This can be circumvented by only drawing back-faces in wire-frame instead of all faces with a depth offset [Raskar and

Cohen, 1999]. Another method is to draw front faces in wire-frame into the stencil buffer, and rendering back-faces in wire-frame only where the stencil buffer is set [Gooch et al., 1999].

Drawing in wire-frame mode ensures a constant line thickness, but is limited by the maximum line width supported by the hardware. Filled polygons can be used instead [Raskar and Cohen, 1999]. For example, when back-facing polygons are translated towards the camera, they poke through the front-facing polygons near the silhouette. The resulting width depends on the slope of the faces and is not easily controlled. Instead of translating the back-facing polygons, they can be "fattened" depending on polygon slope and viewing distance to create a constant line width.

4.2.4 Border Rendering

Silhouette edges can be selected by examining local information only, that is, the facingness of adjacent polygons. However, an object's border cannot be determined by using local information only. In fact, the border may consist of partial edges, too. To our knowledge, there is no real-time analytic method for this problem. An image-based method for border rendering is based on the stencil buffer. The object is normally rendered first, setting the stencil buffer to 1. Next, the object is rendered again in wire-frame mode with wide lines, but only writing pixels where the stencil buffer is zero. Thus, the lines will only be visible outside the object, visualizing its border.

4.2.5 Hidden Line Removal

For most rendering styles, lines occluded by surfaces should not be visible in the final image. The image-based methods explained above solve the hidden line removal problem using the hardware depth buffer. The depth buffer can also be used to remove hidden parts for analytically selected edges. When the edges have been classified, they are simply rendered with depth test. The same problems as discussed for polygon edging arise here.

A higher-quality of hidden line removal can be achieved by rendering polygons using the hardware and then checking the visibility of edges by sampling the frame buffer along the edge [Northrup and Markosian, 2000; Isenberg et al., 2002]. Visible edges are composited on top of the rendition without further depth tests. The main advantage of this is that the resulting visible edges can be drawn in a stylized way (see Figure 4.4).

Purely analytic methods would provide for an even better quality, however they are in general too complex for real-time applications (their complexity is quadratic in the number of polygons). Markosian et al. determine a "quantitative invisibility" for each edge which is propagated to adjacent edges and changes only when an edge crosses a silhouette. For each connected chain of edges, the absolute visibility is established using special rules or by ray casting [Markosian et al., 1997].

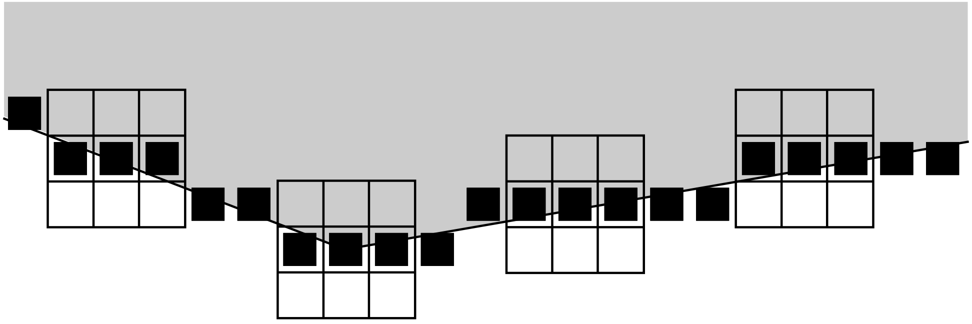


Figure 4.4: Testing the visibility of an edge by sparsely sampling the z-buffer [Isenberg et al., 2002]

4.2.6 Image-Based Outlines

A different approach to rendering contours is through the processing of rendered images, as pioneered by [Saito and Takahashi, 1990]. Only recently have graphics boards became powerful enough to implement this technique in real-time. In Mitchell et al.'s work, the scene's normals, depth values, and material *id*s are rendered into textures. Next, a 5-tap discontinuity filter is applied which compares each texel to its four adjacent neighbors. For the normal channel, a dot product between neighboring texels is performed, while for the depth and material channels, the difference is taken. These values are compared to threshold values and combined to decide if a pixel is on an edge or not, and thus to be colored black or white. In addition, a dilation filter can be performed to create wider lines [Mitchell et al., 2002].

4.3 Surface Rendering

In photorealistic rendering, surfaces are typically the only visual elements considered. Photorealism essentially tries to capture the way physical objects interact with light. The fundamental principle of light interacting with surfaces is described by the rendering equation [Kajiya, 1986]. Most importantly in our context is the strict proportionality of incident radiation to reflected light for each point on a passive (non-emmissive) surfaces. If twice as much light comes in, twice as much light will be reflected, all other factors being constant.

This strict proportionality is weakened in non-photorealistic rendering. In the most simple case, lighting is ignored completely. For example, a contour drawing with lines of constant width does not depict surface area at all, but rather only indicates the visible limits of a surface. Similarly, there are certain drawing style that just paint an object with a singular color. These styles can simply be rendered by filling the surfaces with a single color.

In most other cases, the dependency of the rendered image on the lighting is more complex. The resulting function can still be uniform across a surface, or it can be discontinuous and location-dependent in either world or image space.

4.3.1 Uniform Shading

In uniform shading, a light intensity representing the entire surface is mapped onto a shade and applied to all pixels on the surface.

Gooch et al. replaces intensity variations with color variations. This so-called “cool-to-warm shading” applies colder, bluish hues in darker areas and warmer, yellowish shades in lighter regions. The model can be implemented using the standard OpenGL lighting model by choosing special material and light parameters [Gooch et al., 1998].

Lighting in local illumination models depends to a great extent on the inclination of the surface viewed, that is, the surface normal. Instead of using the actual illumination, some researchers disregard lighting and use the surface normal directly to compute the shading. For real-time scenarios, this can be implemented by using an environment texture indexed by the normal. The environment map can be generated procedurally, hand-painted, or captured [Gooch et al., 1999; Sloan et al., 2001].

A variant of shading akin to the one used in cartoons uses exactly two tones, a lighter one and a darker one. For this, the illumination values need to be discretized. Lake et al. introduce a real-time approach using textures. The illumination is calculated at the model’s vertices. The resulting intensity is used as an index into a one-dimensional texture map. The texture contains the darker and the lighter tone. Because of the interpolation of texture coordinates across a polygon, the boundary between light and dark tones can lie inside a polygon [Lake et al., 2000]. Another approach is to split faces along the dark-light boundary and flat-fill each resulting polygon [Claes et al., 2001].

4.3.2 Shading With Implicit Marks

The uniform shading as explained above treats every pixel on the surface equally. Just like in photorealistic shading, the color of a pixel depends solely on the light that is reflected. The major difference to photorealism is that a mapping is performed on a pixel before it is displayed.

The idea of rendering *marks* introduces a concept into the rendering process that is not found in photorealistic rendering [Schofield, 1994]. It makes the rendering of a pixel dependent on its position, namely, whether it is covered by a mark or not. Indeed, there are two different ways of expressing marks. First, an image-based approach determines for each pixel if it is covered by a mark or not, and colors the pixel accordingly. Thus, the mark is defined implicitly, and we shall refer to such marks as *implicit marks*. Second, each mark can be rendered as a separate entity, coloring only a subset of all pixels. We refer to marks rendered in this fashion as *explicit marks*.

There are two major problems in implicit mark generation. The first is to achieve a constant mark density even if the objects depicted change in size on screen. The second is to vary the density of marks according to lighting.

One solution to get constant screen-space density is to actually use a screen-aligned parameterization for stroke textures. This approach is taken by [Lake et al., 2000]. It results in the so-called “shower-door” effect where objects appear to move beneath a surface attached to the screen. Many viewers find this effect disturbing in motion, although the technique works quite well in still images. To display shading information, textures are created that show an increasing number of strokes. At run-time, the model’s polygons are sorted into bins according to a lighting calculation at the vertices. Along the border between two bins, polygons are split analytically. Finally, each bin is rendered with one of the stroke textures.

To circumvent the shower-door effect, most other publications report that the texture is attached to the object’s surface. A solution to the density problem now popular is to use mipmap texture filtering, introduced in [Klein et al., 2000]. Surfaces are covered by textures that are pre-rendered in an off-line process using a mark-based renderer. This pre-process includes lighting which is not variable at run-time. Because at run-time the mark size would vary with distance, multiple renderings with differently-sized marks are stored as mipmap levels. The mipmap texture filtering hardware automatically displays appropriately-sized marks at run-time without any processing overhead on the side of the application program.

Independently of Klein et al.’s approach, the usage of mipmap filtering to attain a roughly constant stroke size on the screen was developed in the course of this thesis (see Section 6.1), and published in [Freudenberg et al., 2001b].

Various texture-based stroke shading methods have been proposed. Praun et al. introduce tonal art maps (TAMs), which capture the appearance of a hatched surface for various tones and scales [Praun et al., 2001]. An automatic hatching procedure creates a series of textures with increasing stroke density. For each density, a set of mipmap levels is created that portray the same tone. Care is taken to match strokes across mipmap and density levels to maintain frame coherency when blending from one texture to the next size (mipmap level) or tone (higher-density texture). Tones are blended by controlling weights at the vertices. The linear interpolation of Gouraud shading smoothly blends different intensity levels (see Figure 4.5).

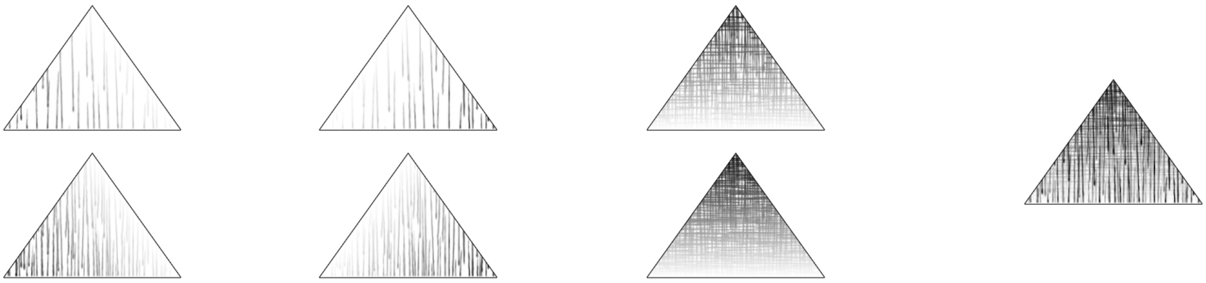


Figure 4.5: Six TAM images are blended using weights at each vertex. The result is shown to the right [Praun et al., 2001]

Finer tone control is achieved by Webb et al. using either 3D textures or TAMs that are evaluated for each pixel using special texture combining modes [Webb et al., 2002]. The first method loads a series of textures with increasing density into a 3D texture. At run-time, the third texture coordinate can be used to adjust lighting at the vertices. A similar approach with 3D textures was published by [Winnemöller and Bangay, 2003].

Webb et al.'s second method builds on the TAM method, but can control the blending at each pixel. Its main insight is that the intensity of each texel in a TAM is a piecewise constant function of the overall intensity. A texel starts out uncovered, then is eventually partly overdrawn by strokes at certain overall intensities, and finally covered completely, at which point its color does not change anymore. The thresholds (intensities at which a certain pixel is covered by stroke) and corresponding colors (the color values a texel has after being covered) are stored in separate texture channels. At run-time, texture combiners evaluate the intensity function for each pixel by comparing the actual intensity to the threshold values and so recreating the color of the pixel [Webb et al., 2002].

Veryovka also uses threshold textures in a similar way, but for a different purpose. His method is aimed at producing more color levels, like in cartoon shading where there are dark tones, mid-tones, and highlights. Unfortunately, no real-time implementation is given [Veryovka, 2002].

4.3.3 Explicit Mark Generation

In contrast to the above-mentioned techniques of implicit mark generation, there are techniques that generate each mark explicitly. While drawing each mark separately is usually slower than using texturing, it allows more specific control of the mark's appearance.

An early real-time shading technique was presented by Markosian et al.. Short strokes are attached to fixed locations on the surface of objects and only displayed in dark regions. Lighting is calculated by the dot product of surface normal and view vector, which minimizes the number of strokes (this corresponds to a light source coincident with the camera). The stroke direction is chosen perpendicular to both the view vector and surface normal by taking their cross product, which roughly aligns strokes with the object's silhouette [Markosian et al., 1997].

Elber presents a similar approach aimed at interactive rendering of free-form surfaces. A coverage of the object with strokes is computed in a preprocessing step. At run-time, strokes are selected for display based on lighting and silhouette proximity. To maintain frame coherency, the stroke length is varied continuously. It is noted that the same approach could be used when scaling or zooming objects to maintain the stroke density [Elber, 1999].

An image-based approach for maintaining stroke density was chosen by Kowalski et al.. For shading they use strokes called "graftals" which may consist of an outline and filled regions. First, a "desire image" is rendered which encodes the desired graftal density as a gray value. This can incorporate lighting or effects defined explicitly by an artist. Next a graftal is created where the desire image has the highest value. This 2D position is looked up in the 3D model via an *id* image. Thus, the 3D normal and other surface parameters can influence the graftal's style. Finally, the graftal is rendered to the screen and also subtracted from the desire image, decreasing the likelihood of placing another graftal in that position. This process is repeated by finding the next desirable graftal location until enough graftals have been drawn. To achieve some frame coherency, graftals drawn in the previous frame are tested first, before generating new graftals [Kowalski et al., 1999].

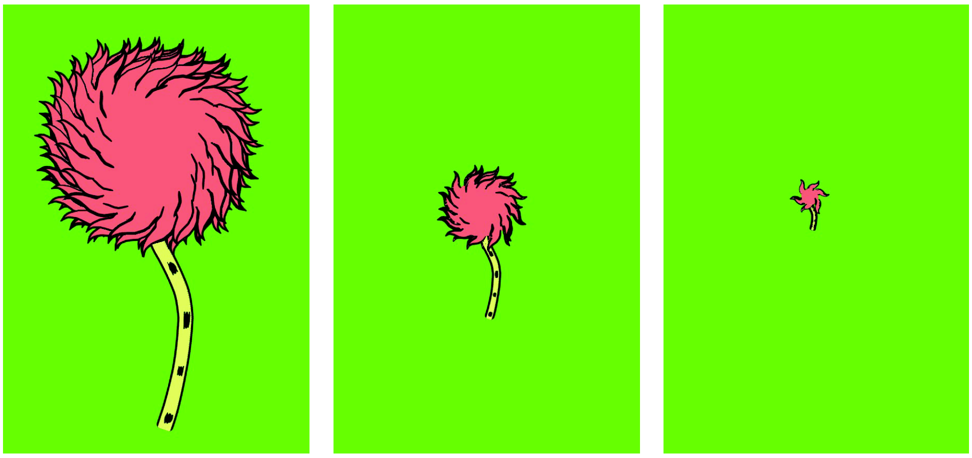


Figure 4.6: The amount of detail is varied by showing more or fewer graftals depending on the viewer distance.[Kowalski et al., 1999]

Also image-based, Secord et al.'s stippling method derives a two-dimensional probability density function from the target image. A number of random points are then distributed according to this function, using various drawing primitives like dots or short strokes in the gradient direction. Given small enough changes in the target image from frame to frame, the resulting point positions vary smoothly. To match exactly a given tone, the density function is adjusted to account for overlapping primitives [Secord et al., 2002].

To improve the frame coherency of the original graftal approach, Markosian et al. attach graftals to the surface of objects rather than generating them in the image plane. The drawing is controlled through a level-of-detail mechanism in a two-stage process. First, individual graftals are designed to be rendered at a certain screen size. The ratio of a graftal's actual screen size to its "preferred" size is used as a parameter in the definition of the drawing primitives, for example to vary length, width, or color. Second, graftals are organized hierarchically into "tufts" that control the number of graftals shown. Again, the level of detail selected depends on a tuft's preferred size. To even more reduce popping artifacts when graftals are introduced or removed, time is used as additional parameter, allowing time-dependent effects to be implemented [Markosian et al., 2000].

Yet another graftal rendering system which attaches graftals to positions on surfaces was demonstrated by Kaplan et al. Frame coherency is achieved by always drawing all graftals, and smoothly scaling their size as well as line width based on distance to the viewer and surface normal. The graftal color can be varied based on lighting. In a "line shading" variant, a fixed number of graftals is created randomly on a surface, and a fraction proportional to the intensity of lighting is drawn at run-time [Kaplan et al., 2000].

A hierarchical approach to the particle density problem was suggested by Cornish et al. They start with a densely tessellated polygonal model organized in a hierarchical structure by a vertex-merging simplification algorithm. At run-time, the particle density is adjusted to the screen area and the scene's lighting by view-dependent simplification. Particles are transformed, clipped, and lit using hardware in feedback mode. Finally, strokes are created and rendered from the now-2D particle data [Cornish et al., 2001].

For stippling, a similar hierarchy-based approach was developed by [Pastor et al., 2003]. How it can be extended to work in real time s developed later in this thesis (see Section 5.4.7).

4.4 Other systems

4.4.1 Interactive Paint Systems

While providing visual feedback at interactive rates when creating an image, painting and drawing systems typically cannot automatically recreate a different view of the subject as fast. Thus, they do not exactly fit the kind of algorithms discussed in this work.

The "Piranesi" system lets the user paint over a pre-rendered three-dimensional model. Painting tools may be sensitive to the underlying geometry, like a brush that emphasizes edges [Schofield, 1994]. In contrast, Salisbury et al.'s interactive pen-and-ink illustration system automatically fills parts of an image with prioritized stroke textures. No 3D object is involved, instead an image is taken as tone reference [Salisbury et al., 1994]. Durand et al.'s drawing tool works on images, too. While each stroke is painted by a user, tone and other parameters are adjusted automatically [Durand et al., 2001].

Other systems allow the user to freely choose the viewpoint because actual 3D geometry is used, and the drawing process is used to parameterize a model instead of just creating a single image. One such system is "WYSIWYG NPR". A user can paint strokes onto a 3D model. From these strokes, more strokes can be synthesized. Multiple levels of detail for drawing are supported. When drawing is finished, the user can freely move the camera, and the system creates a visually similar view of the scene [Kalnins et al., 2002].

4.4.2 Off-line Rendering Techniques

Many researchers have developed stroke-based shading techniques in the past. While the present thesis focuses on real-time rendering, it is worth mentioning the most influential results in off-line rendering.

The quality of the computer-generated pen-and-ink illustrations presented by [Winkenbach and Salesin, 1994] served as inspiration for our technique. They introduced prioritized stroke textures, as well as an approach to detail indication. The method did not deal with animation, however, and the approach to algorithm design is completely different if frame-coherence is a necessity. Similarly, the dependency of visual appearance on the scale of an object as examined by [Salisbury et al., 1996] is incorporated in our own implementation.

Shading three-dimensional scenes by halftoning has a long tradition in off-line rendering, like Leister's digital engravings and copper plates [Leister, 1994] or recent work like Ostromoukhov's engravings [Ostromoukhov, 1999] and Veryovka and Buchanan's halftoning works [Veryovka and Buchanan, 1999]. These methods are interesting references for possible styles. Also, to exactly reproduce tones the halftone screens can be locally equilibrated as demonstrated by [Ostromoukhov and Hersch, 1999].

Similar effects can be achieved by procedural texturing as described by [Johnston, 1998]. See Section 6.7 for our real-time implementation of this idea.

4.4.3 Commercial Applications

The use of non-photorealistic shading techniques is not yet very widespread in real-time applications. However, one commercial system which incorporates such functionality is the "Demon" software package by Archaeoptics Ltd., which is used to create illustrations from archaeological range-scan data [Archeoptics Ltd., 2001]. The details of the rendering process are not published, but several rendering styles are possible (see Figure 4.7).

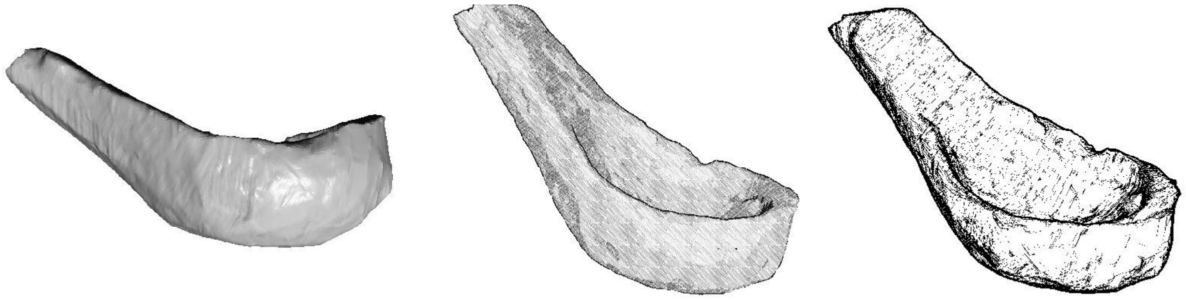


Figure 4.7: Archaeological illustrations created in real-time using Archaeoptics Ltd. "Demon" tool. Several styles are possible, including "Pencil Sketch" (left) and "Ink" (right). Taken from [Archeoptics Ltd., 2001].

One field of applications seeing increasing use of NPR techniques is computer games. However, only cartoon-like rendering styles have been realized to date. One prominent example of this is Pseudo Interactive's "Cel Damage". For shading, a discrete lighting model is used. Black object silhouettes are rendered as extruded backfaces [Provost, 2003]. An example of the normal game graphic can be seen in Figure 4.8. The game also has a hidden rendering mode which creates a sketch-like black-on-white look using an edge detection filter on the standard textures in texture space. Filtering in texture space has the advantage to fade out detail in the distance due to mipmap filtering (see Figure 4.9).

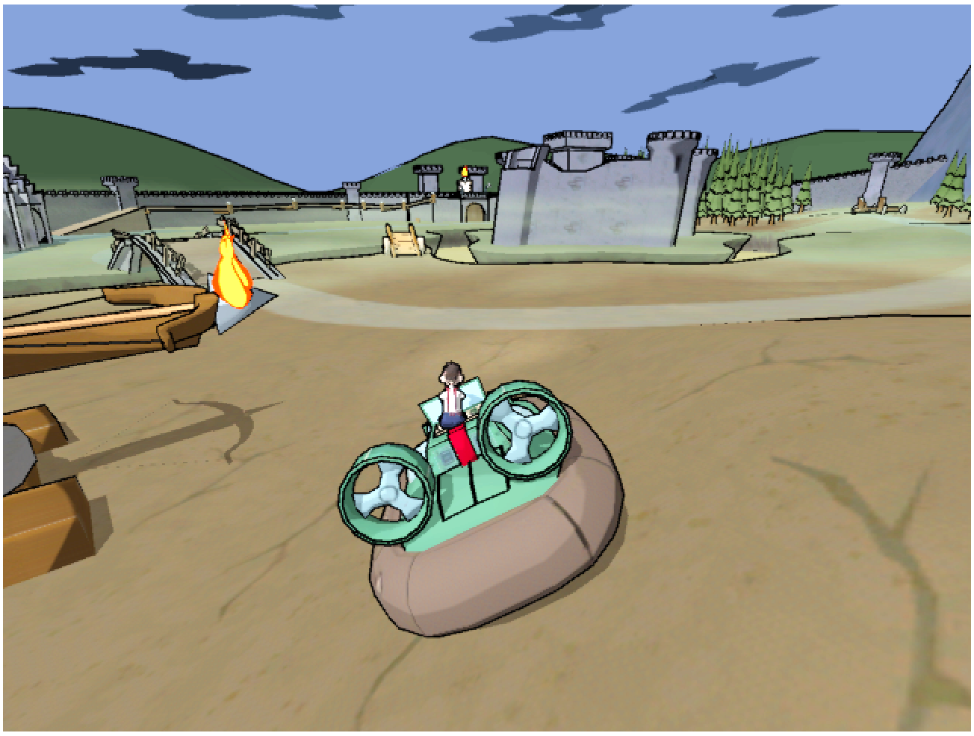


Figure 4.8: Video game "Cel Damage". Black outlines and discrete shading are used to create a cartoon style (courtesy of Guillaume Provost, Pseudo Interactive Inc.)

One step further to capturing the likeness of comics is the game "XIII" [Ubi Soft, 2003], which is based on a popular comic book series [Vance and van Hamme, 1984] (recall Figure 2.7). Small comic-strip insets are shown, and even onomatopoeia (words denoting sounds) are displayed (see Figure 4.10). The rendering, however, is mostly traditional. Background objects are rendered photorealistically, although the textures have a painterly style. A cartoonish style with discrete lighting and black contours is used only on characters and some objects.

The reasons why not more rendering styles problems have been implemented in commercial games are not only of technical nature, as this quote from a game developer shows in response to the question why there are so few NPR techniques used in current games:

> To make NPR look really good, you really need to have the whole art pipeline commit to it and use it – artists, designers, coders. Since NPR is still regarded as a research topic, few people are prepared to commit an entire game style to it – what if it doesn't work? I'm sure we'll see more with time, and not just the cell-shading style. [Forsyth, 2003]

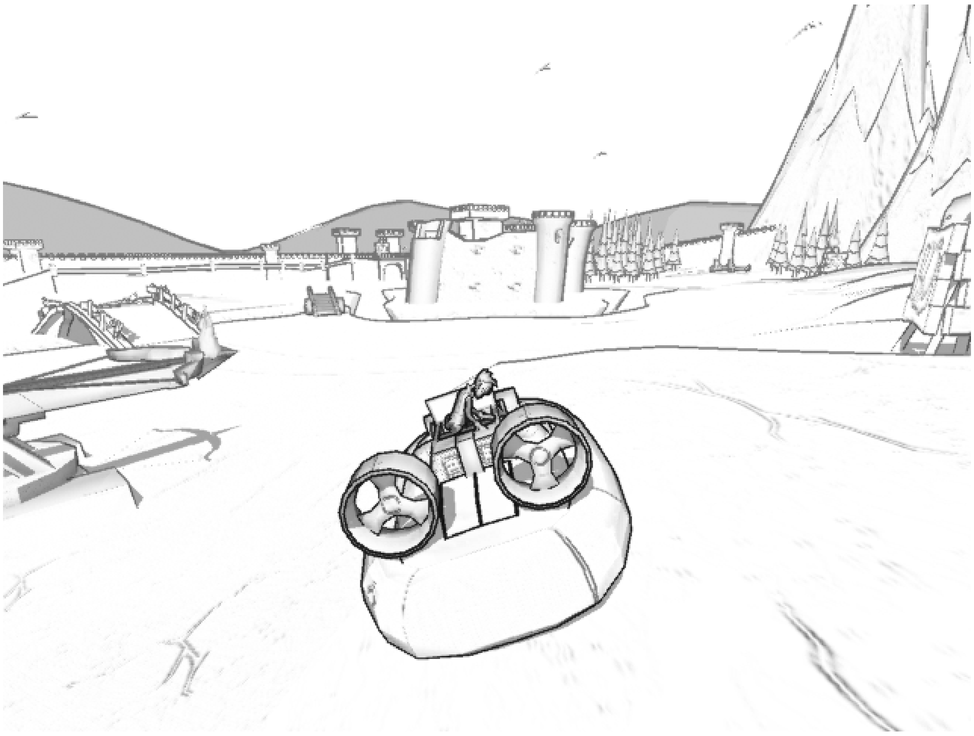


Figure 4.9: Video game "Cel Damage", sketch rendering mode (courtesy of Guillaume Provost, Pseudo Interactive Inc.)

Figure 4.10: Video game "XIII". Outlines are displayed on characters, the textures have a stylized look. Discrete lighting is used, as well as visual language from comic books [Ubi Soft, 2003]

4.5 Summary

Many of the systems discussed in this chapter employ features of current graphics hardware. However, while there are quite sophisticated techniques for contour rendering, most shading techniques rely heavily on CPU processing. In the next chapters, techniques will be developed that are designed with reducing CPU load in mind, and that are tuned for hardware acceleration.

5 Shading with Explicit Mark Generation

The analysis in the previous chapters revealed three major points that, taken together, will lead to a new concept for an interactive NPR system. The first is the observation that in traditional illustration styles like pen-and-ink or engravings, shading is depicted by individual drawing elements. The arrangement, number, and style of these elements both characterize the surface structure and depict the three-dimensional shape of an object through intensity changes. In our method, we will be creating individual texture elements, too, that can indicate structured surfaces and vary in changing lighting conditions.

The second point is the conclusion that 3D particle-based NPR methods are especially well suited for solving the frame-coherency problem, which is one of the major obstacles to smooth non-photorealistic animation. On the other hand, approaches that mimic the traditional drawing process by working in the two-dimensional image plane most closely can approximate the visual style of hand drawings. The new approach will incorporate both image-plane operations and three-dimensional particles.

Finally, we have observed the wide availability of immensely powerful graphics rendering hardware. However, it is optimized towards photo-realistic rendering techniques. Our new rendering approach must take advantage of this environment. The increasing programmability of the hardware will be key to achieving this goal.

This chapter will first outline the new concept of stroke-based halftoning with explicitly created strokes. A tool is discussed which allows to interactively create stroke textures which are then applied to objects. Finally a real-time implementation of the stroke rendering process is demonstrated.

5.1 Concept

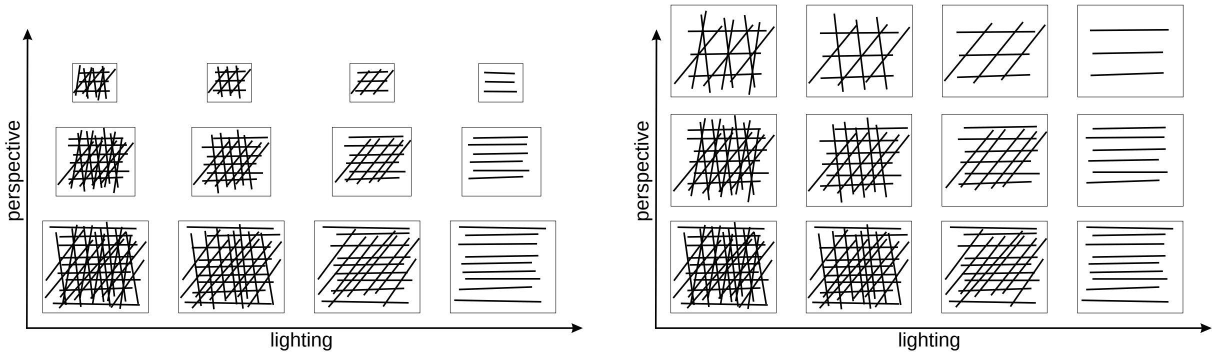
To fully exploit the graphics hardware it must be fed a constant stream of geometric data. To minimize CPU workload and CPU-GPU data transfer, the geometry should be transfered to the graphics card only once, rather than in each frame. The geometry data itself will be static, and rendered for each frame. Vertex programs will be used to modify the geometry on-the-fly while rendering. This trades increased processing power for reduced bandwidth, which is justified by the observation that calculation speed is getting faster more quickly than memory transfer speed.

While various schemes to split the work between the CPU and the GPU can be conceived, we decided to explore the most extreme path, namely, let the GPU do all the work. The

CPU is This poses interesting design challenges for our approach, even if the overall performance might benefit from a more balanced distribution of the work load.

If the graphics board is going to do all the work, it means it has to use three-dimensional data to begin with. However, as has been discussed before, the effects we are trying to implement operate mainly in two-dimensional image space.

Consider, for example, Figure 5.1(a). In this image a style is used that uses a varying number of strokes to depict shading. The net result should be that for a given intensity value, a certain percentage of the image area should be covered by strokes. However, as the strokes are defined in 3D, their spacing actually depends on the perspective size of the surface. If the object's screen size gets smaller, perhaps due to it moving further away or being tilted or just scaled, the number of strokes must decrease, too. This means that in object space, the lines will be spaced further apart, even though in the image plane their density remains constant (see Figure 5.1(b)).



(a) image space

(b) object space

Figure 5.1: In this hatching style, intensity should by varied according to lighting by using fewer layers of parallel strokes. When the perspective size changes, the structure in the image plane needs to be maintained (a). In object space, however, the spacing of strokes varies with the perspective size (b).

Hence when operating in 3D, we have to anticipate what the actual rendering in 2D will look like, and counteract this. However, due to the pipelined nature of the graphics hardware, it is very inefficient, if not outright impossible, to directly obtain 2D image information in the rendering process. For example, to estimate the actual density of particles in the image in order to decide if more particles need to be drawn, there would have to be a way to access information about neighboring particles when a certain particle is processed.

5.1.1 Particle Density

Instead, we estimate the particle density using a more macroscopic approach. This is possible because we know beforehand how the other particles will be placed, namely, with our own algorithm. When we start with an even distribution of particles in 3D, we know how the distribution will change after the projection in 2D. For a rigid surface covered with particles, the only parameters that influence the particle density in the image are the distance of the object from the viewer, and its angle with the viewing direction. The absolute density of course depends on the actual image resolution and screen size of the object. But the relative change in density can be estimated even without that knowledge. When the distance doubles, the projected area is only a quarter as large, meaning an increase in the density of particles by a factor of four. Likewise, when the surface is not aligned parallel to the image plane, its projected area (and hence particle density) is proportional to the cosine of the angle between the surface and the image plane.

By combining distance and inclination we now have a common factor we will call *perspective scale*. It can be determined for any point on a surface given the distance and normal vector of the surface relative to the viewer position and viewing direction. Thus, by storing the position and surface normal with each particle, the perspective scale can be determined using only local information. The position has to be stored with the particle anyway, the only additional information necessary is the normal vector. This yields a very lightweight addition in terms of memory requirements and bandwidth consumption.

5.1.2 Particle Distribution

But even though we now have found a measure of how dense the particles are in the image plane, we still need a way to vary their number to achieve an even distribution. If we did this on the CPU, we could just create as many particles as needed and send these to the GPU for rendering. However, GPUs currently do not provide this sort of functionality. A GPU cannot create geometry, it only sequentially processes the vertices as transmitted by the GPU, or from a vertex array. Because we do not want to burden the CPU, we have to generate the particles in advance and store them in a vertex array.

When rendering, only selected particles may be actually drawn to achieve the goal of uniform particle density. One possible way to do this would be to limit the number of vertices used from the buffer. When the request to render something from a vertex array is issued by the GPU, it can influence how many primitives are drawn by giving a range in the vertex array. This means that only a range of particles can be culled, namely, the contiguous regions at the beginning and end of the vertex array.

Even when not considering that the calculation of how many primitives to draw would have to be executed by the CPU, this approach would be too inflexible. It would require an ordering of particles in a way that an even thinning of particle distribution is achieved by shortening the rendered vertex range. There are two fundamental problems with this. Firstly, the thinning process is not viewer-independent. Since for efficiency we need to make the batch size quite large, the particles in this batch will cover a larger region of the surface (if not the entire surface). But which particles need to be culled first depends on the orientation of this surface region, so there is not one correct order for all viewing directions. Secondly, we do not discard particles for perspective scale only. As explained before, when portraying a certain surface intensity, particles need to be culled according to different rules than those meant for counteracting perspective scale (recall Figure 5.1).

5.1.3 Thresholds for Stroke-Based Halftoning

Hence we need another scheme for selecting primitives from the draw buffer. From a phenomenological point of view, what we want to achieve is that the number of particles varies depending on both lighting and perspective. Since the graphics hardware and our analysis above require an independent processing of each particle, to realize this effect we propose a procedural approach. This procedure determines, for a given particle, if it should be drawn or not. It has to be in a form that can be implemented as a vertex program on the GPU.

We start by observing that each particle is either drawn, or not. There are two events that can cause a particle that was displayed at a point in time to no longer be displayed at a later point in time: either the intensity is too light, or the perspective scale gets too small (e.g., distance too large). So for a given particle a decision has to be made whether it should be drawn based on the average surface intensity at the particle's position and the surface's perspective foreshortening.

We propose to store a separate threshold value for intensity and perspective with each particle. Such values will then determine the order in which particles are displayed, in particular as the associated parameter changes. These values will depend on the desired drawing style that is implemented with the texture. For example, there might be a style that does not wish to distinguish between lighting-dependent and distance-dependent pruning of strokes. In contrast, to implement the hatching style shown in Figure 5.1, one would assign increasing intensity threshold values to each layer of strokes, while the perspective thresholds would be assigned independently of the stroke layer. This leads to thinning out strokes from all intensity levels when the perspective changes, but removal of one level after another with increasing intensity.

5.1.4 Frame-Coherence and the Principle of Local Change

To solve the problem of frame-coherence, we introduce the *principle of local change*. If some change must be performed in response to a varying environment, keep the changes local and to a minimum. For example, if there is a long object which gets bent in the middle (think an arm and elbow), the surface area changes near the joint. The distribution of particles attached to the surface will get less dense on the outer side, while becoming denser in the inside. There are many ways to re-establish a continuous particle distribution. The whole object might be covered with particles again, but this would result in a non-frame coherent animation. Or, a relaxation process could be employed, but this could ultimately alter the position of all particles.

By applying the principle of local change, we try to keep as many particles as possible in their current positions. In regions where the density gets too high, particles are removed, leaving just enough particles to ensure the desired density. Particles are added in places that are too sparse, inserting new particles between the ones already there. This methods ensures that there is no perceived "sliding" of the textured surface, which would appear like some skin stretch across the object. This is similar to the shower-door effect, which appears like a skin attached to the screen under which objects move.

5.1.5 Drawing Dots

Now that we have found a way to generate and cull particles, we will look into actually drawing them. A relatively simple method is to just display each particle as a dot. This can be implemented by using the point rasterizing mode in OpenGL [Segal and Akeley, 2002, p. 66]. Depending on the graphics board, this mode can display anti-aliased dots of varying sizes. Although only points of size 1.0 are guaranteed to be supported by OpenGL (that is, exactly one pixel without anti-aliasing), current GPUs usually support anti-aliased points 1.0 to 64.0 pixels wide in steps of 0.125 pixels, giving 512 discrete steps (see Section 3.7).

A problem still is how to disable the drawing of a point when the vertex program determined that it should be culled. There is currently no direct way to discard a drawing primitive from within a primitive. However, we investigated several possible solutions to this. The vertex program can, for example, transform the vertex position to lie outside the clip volume. This causes the primitive to be clipped. It should be a relatively efficient method as no fragments are generated for clipped primitives which would have to be passed down the pipeline. Alternatively, the point could be translated in z so that the resulting fragments are discarded by the depth test. Another possibility is to set the point size to zero. However, this violates the OpenGL specification and might lead to undefined results. The fragment color could be set to the background color, but this would cause overdrawing artifacts when particles overlap. A better way is setting the alpha value for the fragment color and enabling the alpha-test, which discards fragments below a certain threshold. This latter approach also has the advantage that with alpha-blending and gradually changing alpha values, a smooth transition can be achieved. This yields a better frame coherence than the sudden introduction or removal of dots in the course of an animation.

Directly displaying the particles as dots creates a real-time "stippling" style. A secondary problem is to create an even distribution of particles for stippling on a surface. In joint work with Oscar Meruvia (see [Pastor et al., 2003]) we adapted his particle generation approach to generate the threshold values for the real-time scheme presented here, which is explained in more detail in Section 5.4.7.

5.1.6 Drawing Lines

For creating a real-time “pen-and-ink” style, we need to draw lines, not just dots. The idea for realizing this is to treat the particles as reference points for the lines in the image. The simplest way here is to use the OpenGL line rasterization mode to draw lines with a constant width in image space.

This method shares many characteristics with the point rasterization mode described above. A line width of zero is illegal, which can be increased in steps of typically 0.125 pixels up to a maximal width, typically at least 10 pixels, depending on the graphics hardware (see Section 3.7).

A similar technique for culling lines can be used as was for points. However, it has to be ensured that to disable a line, both vertices must agree on this decision. If, for example, only one vertex is moved outside the clipping volume, while the other remains inside, a line would still be drawn from the visible vertex to the one outside. The problem is not as severe with alpha blending or alpha testing. However, because of the color interpolation, only a part of the line will be visible, which may or may not be desirable.

A more severe restriction with lines is that their width cannot be modified in a vertex program. Hence the line width cannot be used to achieve a continuous change in intensity. To circumvent this, and to lift the restriction of a limited maximal line width, lines can be drawn as four-sided polygons (quads) instead. Polygons have the advantage of allowing a zero width, so the implementation does not have to resort to fragment processing techniques like alpha blending or alpha testing. However, to create a quad that appears as a line of specific screen-space width, it is necessary to re-implement the line drawing from scratch in a vertex program.

First of all, since programmability of current GPUs is limited to modifying vertices (in contrast to creating vertices), four vertices have to be specified for each line drawn as quad, instead of two. Each end of the line is submitted twice. However, both vertices at the end of a line have to be moved in a opposite direction to span the polygon. The direction in which to move the points depends on the orientation of the line in screen space, as well as the aspect ratio of the view port. The distance to offset the vertices depends on the actual screen resolution. This process will be covered in more detail in Section 5.4.2.

While this approach of drawing line segments as quads allows a greater flexibility, it is also less efficient compared to the simple drawing of lines. Not only are twice as many vertices needed, but the calculations themselves are more complicated. A useful addition to OpenGL addressing this problem would be to allow the modification of line width in a vertex program.

The last possibility of shading primitives, besides points and lines, is polygons. Their position and orientation is fully determined by their vertices, no screen-space operations to maintain a certain size are necessary or even possible. Still, a vertex program needs to be employed to adjust the scale of the polygon depending on the desired intensity.

5.1.7 Hidden Line Removal

One aspect not yet mentioned in this discussion is the hidden line removal. Analytic methods are obviously not implementable as vertex program because they would require access to more than one drawing primitive at a time. The standard method for hidden surface removal implemented in graphics hardware uses a depth buffer. To utilize it, fragments produced by rasterizing the points, lines, or other shading primitives must have correct depth values relative to the surface definition of the object. Then hidden surface removal can be implemented by a two pass method. In the first pass, the depth buffer is initialized by rendering the object surface. Color writes can be disabled for this pass, which may speed up this process on certain hardware. In the second pass the shading primitives are rendered with depth information, and color write. The fragment depth test ensures that only visible parts of the primitives are shown.

5.1.8 A Parameterization Model for Drawing Primitives

To depict shading by varying the area covered by drawing primitives we need a way to parameterize these primitives. By changing their size, more of the background is revealed. This influences the ratio of foreground pixels to background pixels and consequently leads to a different perceived intensity.

Basically, a primitive can be scaled in one or two dimensions. Scaling in one dimension will linearly change its area, while two-dimensional scaling will result in a quadratic proportionality. Specifying the transformation can be cumbersome. The axis for a one-dimensional scaling operation usually is not perpendicular to the coordinate axes. Similarly, the point of convergence for a two-dimensional scaling usually does not coincide with the origin of the coordinate system.

We therefore propose a more flexible model for specifying these scaling operations. As can be seen in Figure 5.2, there exist many variations in how to interpolate the shape of even a single triangle. To easily specify these methods in a unified fashion, we use a target position for each vertex. When the interpolation factor (derived from the target intensity) changes, a new vertex position is calculated as the weighted sum of the source and target positions.

This model of specification is less restrictive than just specifying a scaling axis and factor. However, it does not guarantee a strict proportionality between the interpolation factor and the covered area. In Figure 5.2 it can be seen that the same method can yield both linear and quadratic dependencies. However, if the target positions are not either the same (which would create a quadratic proportionality) or both collinear and span the same extent as the original figure (linear ratio), the dependency might be more complicated.

Note that the line or point of convergence does not necessarily have to lie inside the triangle. Therefore, multiple triangles can be made to shrink towards one common point. This way, polygons can be used as drawing primitives, too. An example of rendering a

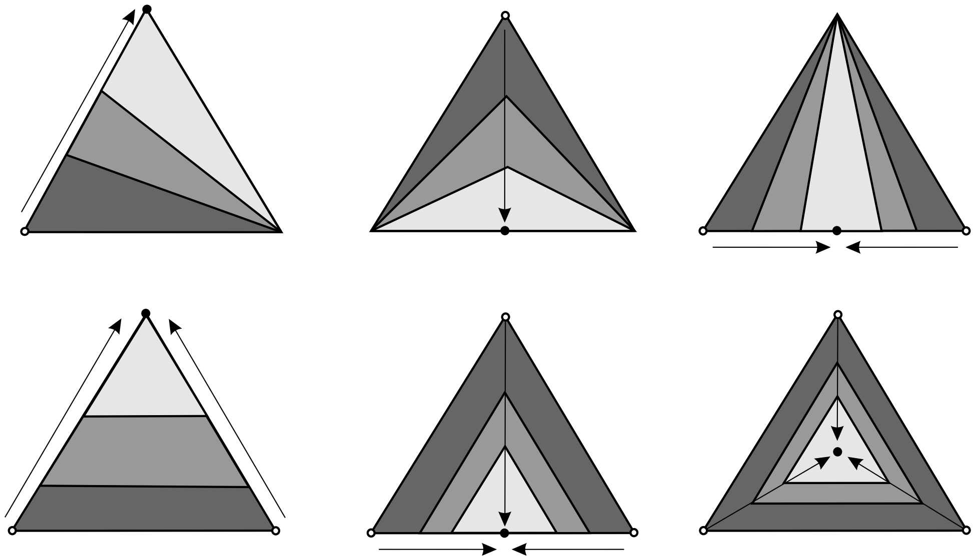
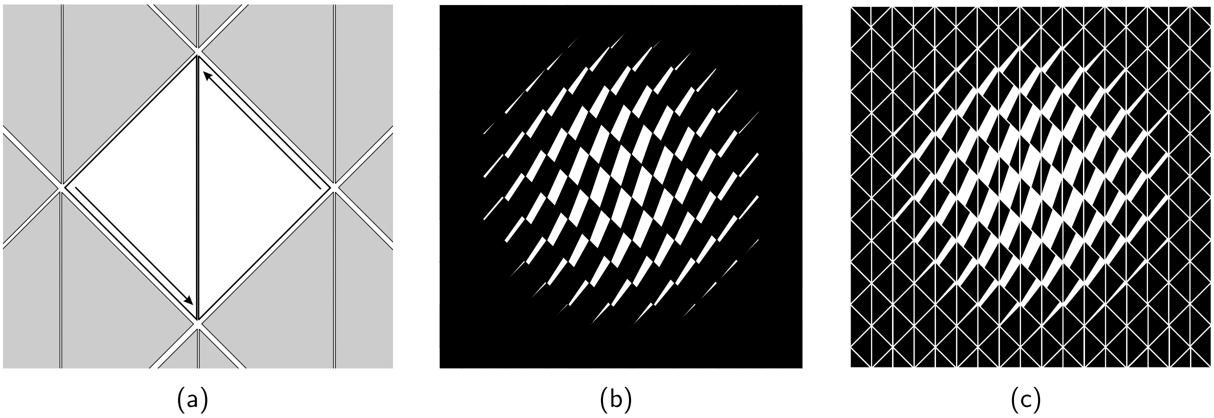


Figure 5.2: Scaling a triangle to depict intensity. As the area of the triangle decreases, a larger area of the background becomes visible. Linearly interpolating one, two, or all vertex positions results in a proportional change of the triangle area. If the triangle converges to an edge, the dependency is linear (upper row), if the limit is a point, the area changes quadratically (lower row).



(a) (b) (c)

Figure 5.3: Shading with a scaled quadrangle pattern composed of two triangles. The scheme used is portrayed in (a), the result for an intensity value of 0.5 with a radial falloff can be seen in (b). To make the pattern more obvious, white edges were added for each triangle in (c).

surface with a pattern of parameterized quadrangles (made up of two triangles) is shown in Figure 5.3.

5.2 Texture Creation

While the primitive interpolation method introduced above is quite powerful, it has two major drawbacks. For one, its run-time implementation is quite complicated and not well-suited for a pure hardware implementation (see Section 5.3.3 for a prototypical solution). The more severe drawback, however, is that to create textures for this approach requires a rather non-intuitive handling.

A more promising approach is to base the creation of surface textures on a process with which the prospective content creator is already familiar. Since we are aiming at a look that is inspired by hand drawings, giving the content author a drawing tool seems to be a promising alternative to explore.

There are a number of drawing systems for creating specific kinds of images or models in non-photorealistic rendering (recall Section 4.4.1). However, none of them qualify as a tool for authoring textures suitable for application to game worlds. They focus on creating images of 3D scenery, or they are annotating models to describe their look. Our intention instead is to have a tool that plays a similar role like the bitmap editing programs used to author bitmap textures. The main characteristic of these programs is that the texture itself is treated independently from the model it is later applied to.

Using a 2D drawing tool also enables a vector-based representation of drawing primitives. This reduces the algorithmic complexity to a considerable degree, because one has to deal only with curves on surfaces, rather than 3D surfaces itself. A stroke can be represented by its main axis (the path) and a width. From this, the original stroke can be reconstructed and similar strokes can be synthesized.

Recall the constraint developed earlier (Section 5.1) that when the texture is shown on a surface, its appearance should vary according to perspective size and lighting (recall Figure 5.1). This effect of different display parameters must be controllable in the drawing tool. In particular, it must be possible to adjust which strokes should be shown if the texture covers a specific area of the screen, as well as the ability to specify a certain order in which strokes should be accumulated to create a given density, and hence intensity.

Manually adjusting the priority of each individual stroke would be rather cumbersome. Instead, we found that for an initial priority assignment, the original sequence of drawing strokes is indeed a good choice. Especially if the artist keeps in mind that the original drawing order will be the basis for further tweaking, and therefore systematically adds strokes, in many cases this initial guess is already sufficient.

Otherwise, the editing features of the texture drawing tool have to be employed. Grouping strokes in layers helps in this task, because in many drawing styles individual groups of strokes are used in specific ways. Again, the user might choose to assign layers to the strokes immediately when drawing (draw some strokes, select next layer, continue to draw, and so on). Since in each layer an artist typically adds strokes in a linear fashion, like from left to right or top to bottom, a randomization function is provided which randomly permutes the order of stroke inside the layer. This makes the texture strokes appear more even when animated, as new strokes are added or disappear distributed all over the surface, instead of linearly in one region only.

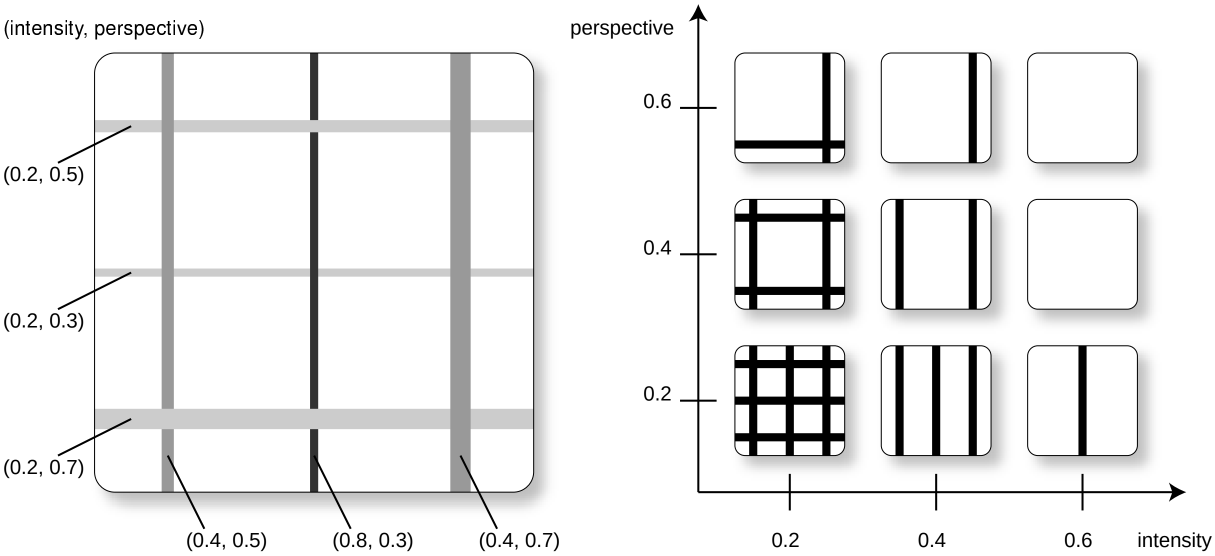


Figure 5.4: Example for stroke culling by priority values. Each stroke is assigned a pair of priorities (*intensity*, *perspective*), as shown on the left (for demonstration purposes, the intensity was mapped to a gray value, the perspective size to the line width). When rendering, reference values are computed for the actual perspective size and intensity. Only when both priority values for a stroke match or exceed the reference values, the stroke is drawn. This is demonstrated in the right diagram for different values of intensity and perspective.

Hence what the user adjusts is not the stroke priority directly, but rather the order in which strokes appear. This also aids in spreading priority values across the [0,1] range, because additional strokes are only appended to the list. The actual calculation of priorities is carried out when the texture is saved in a file.

Priority values are assigned based on the lengths of the strokes. For each layer, the sum of the lengths of the strokes is calculated, which roughly corresponds to the contribution of this layer to the total area covered by strokes. The range of priority values for a layer is the fraction of total stroke length this layer contributes. Each stroke in a layer is randomly assigned a value in this range.

This method of assigning stroke priorities does note take into account the actual coverage of area by strokes, which does not only depend on stroke length, but also on line width and overdraw. If a more accurate representation of gray values is desired, an equalization can be performed. The algorithm essentially remains the same as the length-based approach, but instead of measuring the fraction of stroke length, the fraction of area covered by strokes is considered. Analytically calculating this percentage of area would be prohibitively expensive. However, it can be accomplished with reasonable accuracy by rendering the strokes to a pixel matrix and counting the number of pixels newly covered by each stroke.

5.2.1 Drawing Tool

The drawing tool was implemented using the GLUI toolkit. It presents the user with a drawing window and an options window. The interaction is mouse-based. However, test users found the program easier to use with a stylus. We arranged this by setting up a Wacom Cintiq panel with the stylus in mouse emulation mode (see Figure 5.5). This means no pressure data is available. However, this is sufficient for our purposes because currently, the tool does not allow to modify the line width.

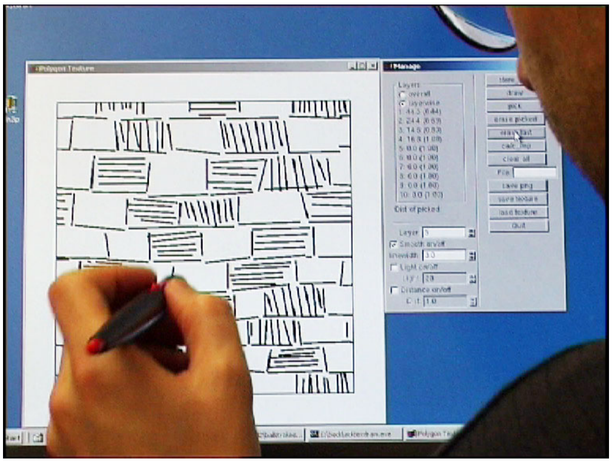


Figure 5.5: A user is drawing a texture with a pen on a touch sensitive screen.

When a stroke is drawn, it gets added to the currently selected layer. The layer has to be explicitly changed by the user. When drawing a stroke that extends outside the limits of the square, it is automatically "wrapped around". This ensures that the texture will be seamlessly tileable. In edit mode, the user can select strokes and modify them, for example by moving the endpoints or adjusting the layer.

Lighting priorities are assigned automatically based on the layer information. Perspective priorities are either set to match the lighting priorities or are distributed randomly over all strokes.

The user can verify the appearance of the texture under varying conditions by applying an intensity threshold (only strokes above that threshold are displayed) or a distance threshold. Both values can be adjusted independently, for example to examine the effect of lighting changes for distant objects.

Internally, stroke records are stored in a list. Each element corresponds to one stroke and contains the following attributes:

- start vertex (u, v)
- end vertex (u, v)
- lighting priority
- perspective priority

- layer number

Stroke textures are stored externally in a simple text-based file format. Each line describes one stroke with six floats and one integer, separated by spaces:

Field	1	2	3	4	5	6	7
Type	float	float	float	float	float	float	integer
Content	start u	start v	end u	end v	light priority	persp. priority	layer

The file is named *filename*.stx (for stroke texture). Additionally, the strokes are rendered to a square bitmap and stored in PNG format as *filename*.png. This bitmap texture is used later to aid in assigning texture coordinates.

5.3 Texture Application

Before 3D objects can be rendered with the stroke textures described above, the textures have to be “applied” to the model. Their position, orientation, and scale have to be specified. Also, some pre-processing has to be performed to actually transform the textures from two-dimensional texture space into the three-dimensional object space.

5.3.1 Comparison to Traditional Texture Mapping

In a conventional bitmap texture, texture elements (texels) are contiguously stored in a fixed, equally-spaced grid. At each grid point, exactly one texel is stored. This allows random access to individual texels. Given a texture coordinate, a simple lookup can be performed by calculating the memory index of that texel, and retrieving the content of this memory cell. This has constant cost. Because of this simplicity, today’s GPUs can fetch billions of texels per second.

A vector texture, on the other hand, does not have these properties. There is no grid, and not even a one-to-one correspondence between texture coordinates and texture elements. Stroke textures are sparse, rather than contiguous, because there can be regions not covered by any stroke. Moreover, since strokes can overlap, there may be more that one stroke per texture coordinate.

In traditional textured rendering, for each frame, every triangle in the model is rendered in sequence. For each triangle, the texture coordinates at the vertices are used to determine the part of the texture that is to be transformed to the screen. Using a forward-differencing scheme, the texture coordinates are interpolated for each pixel. Then the corresponding texel is fetched and used to generate the fragment that gets stored in the frame buffer.

If that same method would be applied for stroke textures, it would mean that for each frame, for each triangle in turn, the strokes covering that triangle would have to be determined, clipped, transformed to screen space, and drawn. To determine the strokes, an intersection test with the triangle in texture space would have to be performed for each stroke, since there is no direct indexing from the coordinates alone. This could be sped up by using a spatial partitioning scheme on the strokes. In this case, strokes partially covering the triangle need to be clipped so that no parts outside the triangle are affected. Finally, the clipped strokes are projected and drawn.

Obviously this is a very time-consuming process. It is not feasible to do this in each frame on the CPU, and impossible to do on the GPU. Fortunately, most of this can be offloaded to a preprocessing step. If at run-time only strokes already clipped need to be transformed and drawn, this task can easily be implemented by the GPU. Since we chose to use a fixed texture mapping in object space, the application and clipping process can be performed before the actual rendering.

In conventional bitmap texturing it is common to repeat a texture on an object by “wrapping”. Texture coordinates are assigned that cover a wider range than the interval $[0, 1]$. When rendering, only the fractional part of the texture coordinates are used for addressing texels, so that the same texels will be fetched for different parts of the surface. Also, a texture can be reused on the same object by mapping a texture to different parts of the object. In fact, “unique” texture coordinates are the exception, rather than the norm. Even separate objects can share textures.

This texture reuse increases efficiency in multiple ways. Less memory is necessary to store the textures. The upload is faster because of the smaller size. Caching schemes are more effective with repeated accesses.

Unfortunately, this kind of texture sharing and reuse is not possible in the vector scheme, since all texture elements are transformed to their object-space position. If a texture is repeatedly applied on a surface or on multiple objects, the texture elements are duplicated and transformed to their respective positions. Even if a scheme was found to reuse the texture elements (perhaps by adjusting a transformation matrix for each instance of the texture), the clipping is most likely different from one instance to the next, so it could not be reused.

An advantage of unique texturing is that textures can be modified independently. If a texture is altered in a shared scenario, the change occurs in all instances, which usually is not desired. For example, if one wall is painted by drawing into the texture map, the paint would appear on all recurrences of that wall texture. In the context of stroke textures this means that every instance of a texture might be different from the others. This could be used to, for example, randomly jitter the stroke positions to make the repeated application of textures less obvious.

5.3.2 Surface Parameterization

Despite the various differences between traditional texture mapping and out stroke textures, the basic principle is the same. A parameterization of the surface is used to map texture elements from texture space to object space. The parameterization for polygonal models is usually given as texture coordinates for each vertex.

We use a conventional modeling tool (3D Studio Max) for creating the parameterization. While an automatic surface parameterization like that proposed by [Hertzmann and Zorin, 2000] may be suited for regular hatching, the rather non-homogeneous nature of our textures is better left to manual control.

Since the modeler cannot cope with analytic textures, the bitmap texture as produced by the texture drawing tool is imported into the modeler to visually aid the texturing process. The resulting parameterization (uv coordinates for each vertex) is stored along with the polygonal model. We use the ASE (ASCII Scene Export) format to export the model from 3D Studio. Besides the geometric information and texture coordinates, an ASE file also contains material information, including the filenames of textures. When reading the file, the bitmap file names are converted into vector file names by replacing the ".png" extension with ".stx".

5.3.3 Parameterizable Polygon Clipping

To render a geometric texture on the surface of an object it is necessary to clip the drawing primitives to each surface primitive. Since the drawing primitives change their size according to lighting, the clipping actually would have to be performed in every frame. Because this analytic clipping process would be prohibitively expensive, we seek a parameterizable clipping operation—the result should be a polygon that behaves under changing lighting conditions exactly as the clipped part of the original polygon. We next discuss a method to create such a parameterized clipped polygon.

Suppose a texture triangle $\Delta M = \Delta M_1 M_2 M_3$ should be clipped against a model triangle $\Delta D = \Delta D_1 D_2 D_3$, as depicted in Figure 5.6. The size of ΔM is varied by replacing M_3 with a point M_p that moves linearly from M_2 to M_3, scaling the triangle from 0 % to 100 %. The intersection of two triangles is at most a hexagon, with vertices P_1 to P_6. Some of these vertices will need to move according to the current scale of the triangle.

We start by computing $\alpha_i = \angle M_2 M_1 P_i$ that is the angle between edge $\overline{M_1 M_2}$ and P_i (in Figure 5.6, $\alpha_1 = \alpha_2 = 0°$ and $\alpha_5 = \alpha_6 = \alpha$). We also store the length $l_i = |\overline{M_1 P_i}|$ and the angle $\gamma_i = \angle M_1 P_i P_{i+1}$ between this edge and the next polygon vertex. At last, we store the lengths $l_1 = |\overline{M_1 M_2}|$ and $l_2 = |\overline{M_2 M_3}|$ as well as the angle $\beta = \angle M_1 M_2 M_3$. From this information, the polygon can be reconstructed at run-time.

For drawing the polygon according to a reference value $p \in [0, 1]$ we first determine the new triangle vertex M_p that shrinks the area of $\Delta M_1 M_2 M_p$ by the factor p. For the new triangle the angle $\alpha_p = \angle M_2 M_1 M_p$ is calculated by the SAS theorem:

$$l_p = |\overline{M_1 M_p}| = p l_2$$

$$\alpha_p = \sin^{-1}\left(\frac{l_p \sin\beta}{\sqrt{l_p^2 + l_1^2 - 2 l_p l_1 \cos\beta}}\right)$$

Now every vertex of the polygon is processed in turn. For the ith vertex, if $\alpha_i < \alpha_p$, the vertex is not interpolated but drawn as is. If, however, $\alpha_i > \alpha_p$, the position has to be

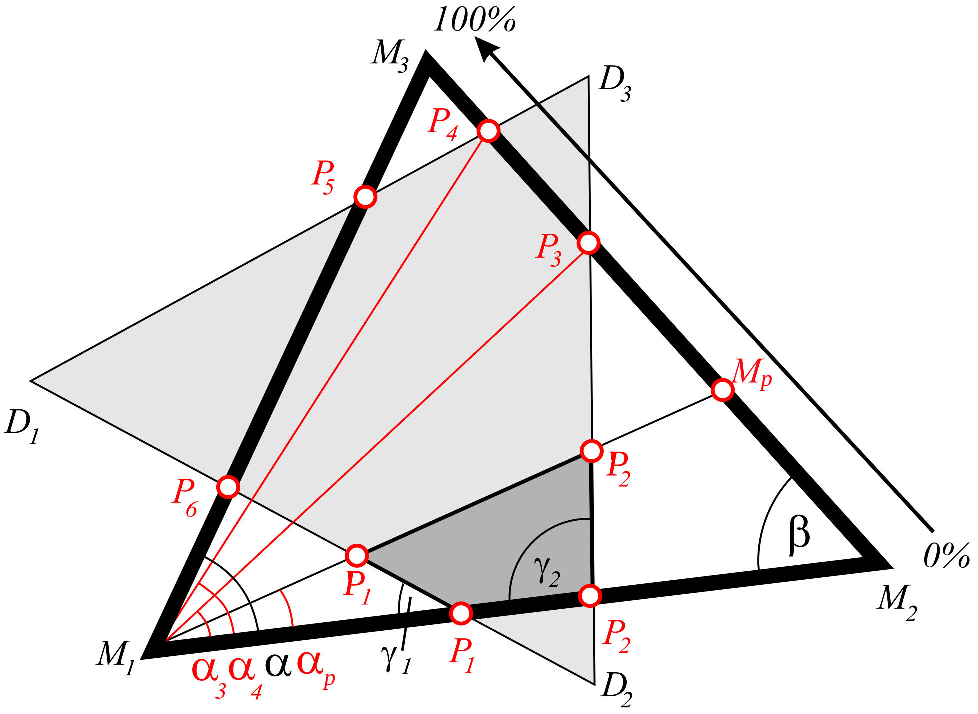


Figure 5.6: Analytic clipping of triangle $\Delta M_1 M_2 M_3$ against $\Delta D_1 D_2 D_3$ so that the resulting polygon $P_{1\ldots6}$ behaves just like the original triangle when it is reduced in size by moving M_p between M_2 and M_3.

recalculated. The same test is performed for the vertexes predecessor and successor. If both $\alpha_{i-1} > \alpha_p$ and $\alpha_{i+1} > \alpha_p$, the interpolation creates two new points, otherwise only one.

This process is rather expensive. Even if it were optimized, a problem that still remains is that the number of vertices varies depending on the actual lighting values. Since vertices cannot be created or removed in vertex programs, this prevents an efficient GPU implementation.

Instead of trying to clip variably-sized polygons on the surface of objects we instead will generate lines on the surface of objects and generate variably-sized polygons from them at run-time.

5.3.4 Line Clipping

The strokes defined in the texture drawing tool need to be transformed into the object's coordinate system (see Figure 5.7). For all triangles sharing a texture (material), the range of texture coordinates at the triangles' vertices is determined. If the range is larger than one, strokes in the texture have to be tiled across the surface. Each stroke in the texture (and its tiled copies) is tested against each triangle in 2D texture space. If it overlaps a triangle, it is analytically clipped against the three edges of the triangle.

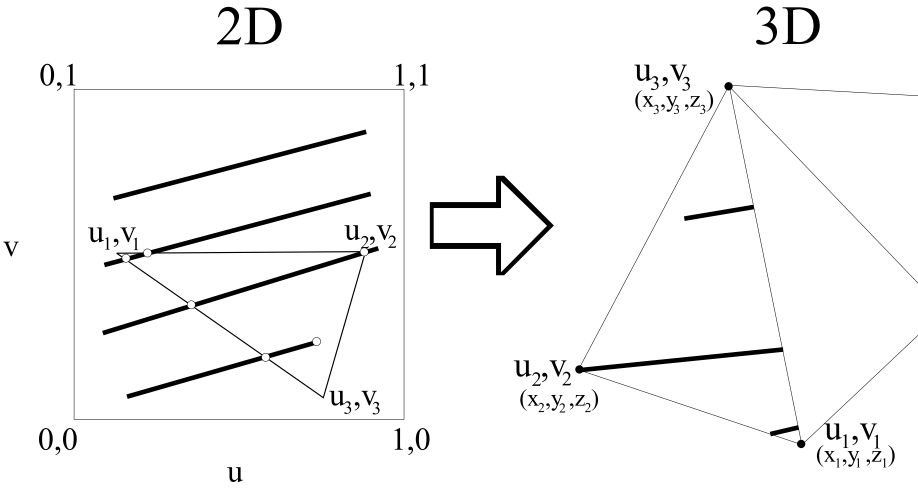


Figure 5.7: Application and clipping of lines via barycentric coordinates.

The clipped segments are then transformed from 2D texture space into 3D object space. For each triangle, the coordinates of its vertices are known in both texture space (u, v) and in object space (x, y, z). A mapping needs to be performed that takes the relative position of the segment's endpoints with respect to (u, v) and creates 3D points establishing the same relation to the (x, y, z) coordinates.

This mapping is performed using barycentric coordinates as intermediate representation. First, the barycentric coordinates (t_1, t_2, t_3) for both endpoints are determined and normalized so that $t_1+t_2+t_3 = 1$. This yields a coordinate-system independent representation for the positions. Then, to determine the corresponding 3D position, the triangle's vertex positions are weighted with the barycentric coordinates $\sum_{i=1}^{3} t_i(x_i, y_i, z_i)$. The result is a list of line segments with endpoints having 3D coordinates in object space. Also, the priority values from the original stroke in the texture are stored with each segment.

Note that the relation between segments and surface triangles needs to be maintained. This can be accomplished by storing the triangle *id* with each segment. Since there tend to be many segments per triangle (depending on the density of tessellation) it can be more efficient to store lists of segments per triangle. The triangle-segment relation will be used in rendering, for example to access the surface normal, or to cull all segments if their corresponding triangle is invisible.

Field	1	2	3	4	5
Type	float[3]	float[3]	float	float	integer
Content	start position	end position	light priority	persp. priority	polygon id

Of course, these lists of strokes can also be stored per object. This is necessary if, for example, the objects are going to be animated and thus have to be transformed separately.

5.3.5 Indication Mapping

In drawings, artists sometimes only *indicate* the texture of objects in certain regions, rather than minutely filling in detail for the whole area. This also reduces visual clutter while still hinting at the surface quality [Strothotte and Strothotte, 1997, p. 277]. Computer-generated line drawings, too, benefit from this technique [Winkenbach and Salesin, 1994].

We support indication by using a gray-scale texture map on the model's surface. Darker values mean that more detail should be drawn, while lighter values mean to use less detail. However, with current hardware a texture map cannot be accessed in the vertex program. This would be needed to take the in indication value into account. Instead, we perform a preprocessing step to incorporate the indication map into the stroke texture.

The indication map is usually defined with unique texture coordinates, so each part of the model can be independently indicated. However, this is no precondition for our approach.

When the strokes from the vector texture are applied to the model, the indication value p_i is looked up in the indication map for the midpoint of the stroke. This value is used to modify the lighting priority p_l value of the stroke. We define a value of $p_i = 0.5$ (mid-gray) to not change the priority. The new priority then is calculated by $p_l' = p_l + c_i(0.5 - p_i)$ where c_i is a constant regulating the influence of the indication map.

The result is that stroke priority values are lowered in regions that should show less detail according to the indication map, while priorities are increased where indication is wanted. Since this is done at preprocessing time, no run-time penalty is acquired.

5.4 Rendering with Vertex Programs

The result of the preprocessing that applied a stroke texture to the surface is a set of stroke segments in 3D object space, and the original surface polygons are retained, as well. The camera is modeled with a modeling and viewing matrix specifying the object positions and viewing direction, and a projection matrix defining the perspective projection, as usual in real-time rendering. Thus, we reuse the standard OpenGL viewing pipeline. In addition, a light source can be incorporated.

The basic rendering algorithm proceeds as follows. For each frame, the frame buffer and depth buffer are cleared. Then, for each object, the transformation is set, and the object's surface is rendered with enabled depth test and depth writes. Color buffer writes can be disabled. This initializes the depth buffer so subsequently painted strokes will be correctly hidden if behind a surface. When all objects have been rendered, color writes are enabled, while depth writes may be disabled. Now the strokes are rendered, using the same transformation as their corresponding objects. A vertex program is employed in this pass.

This vertex program forms the core of the rendering algorithm. Most paramount, it selectively removes strokes to create the desired density of drawing marks and thus depict a certain intensity. It takes into account lighting and compensates for changes in density caused by a changing perspective scale of the object. In addition, the vertex program creates the actual drawing primitives in screen space from their object space stroke description.

5.4.1 Data Layout

The stroke data as defined in Section 5.3.4 is not yet in a format suitable for efficient hardware rendering. Vertex arrays have to be set up in such a way that the GPU can continually pull vertices from the arrays. A severe restriction in geometry processing on current GPUs is that each vertex is processed separately. This is necessary to prevent dependencies which would hamper a parallel processing of vertices, which is one of the keys to the efficiency of vertex processing. Thus, the stroke data must be converted from the CPU-centric layout into a GPU-suited layout.

This layout of the vertex array strongly depends on the actual rendering primitives. If line primitives are used, two vertices are used per stroke. If, in contrast, quads are employed, four vertices per stroke have to be provided. Since each stroke is not connected to the next, more efficient representations like poly-lines or quad-strips cannot be used.[1]

Since each vertex is processed independently by the graphics hardware, the data has to be replicated in all 2 or 4 vertices comprising a stroke. For example, the data associated with each vertex in the line-based scheme (two vertices per stroke) is at least the following:

- P_o vertex position in object coordinates
- N_o normal (for lighting and slope correction)
- p_p perspective priority
- p_l lighting priority

Here, P_o is different for each pair of vertices comprising a stroke, while N_o, p_p, and p_l are identical. Since the extent of the line is automatically determined by the hardware, the screen-space position P_s of the vertex, which is calculated in the vertex program, coincides with the projection of the P_o (see Figure 5.8).

In the quad-based scheme, both vertices at one end of the stroke have the same P_o, otherwise the structure is identical. In addition, the direction of the stroke must be provided:

- D_o stroke direction

---

[1] If a stroke was clipped, the segments are connected so a strip might be employed. However, the normal of their respective model faces is different.

Figure 5.8: Two vertices are needed to render a stroke in the line-based drawing scheme (shown separately here). The object-space vertex position P_o is different for each vertex, while the normal N_o, and the priorities p_p and p_l are replicated (not shown). In contrast to the quad-based scheme (see Figure 5.9) the screen-space position P_s is just the projected object-space position P_o.

However, to distinguish between the two vertices at one end, we negate D_o for one them. The screen-space position P_s will be determined by projecting P_o to screen space and offsetting it perpendicularly to the projected stroke direction by the desired line width (see Figure 5.9).

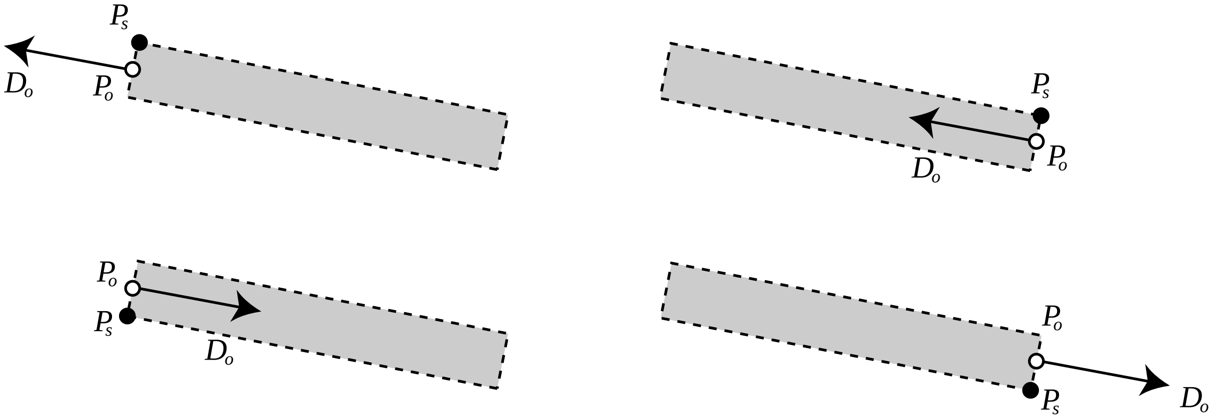


Figure 5.9: In the quad-based drawing scheme, four vertices are needed per stroke (shown separately). The object-space vertex position P_o is the same for the two vertices at one end, but the stroke direction D_o is negated for one of them. As in the line-based scheme (Figure 5.8), the normal N_o, and the priorities p_p and p_l are identical in each vertex, and therefore not shown. The screen-space position P_s is offset from the projected object-space position orthogonally to the projected stroke direction.

Using Cg notation, the structure for a vertex used in the line-based scheme is as follows:

```
struct LineVertex {
  float4 Position : POSITION;
  float4 Normal : NORMAL;
  float4 Priority : ATTR3;
};
```

This uses one Priority vector for both p_p and p_l. They are accessed as Priority.x and Priority.y, respectively. For even more space saving, if the position's w component is not utilized, the priorities could be stored as Position.w and Normal.w. Discarding the Priority field would save 33 % memory. Due to the complex vertex program, however, this would probably not result in a speed increase.

For the quad-based drawing scheme, we use this structure:

```
struct QuadVertex {
  float4 Position : POSITION;
  float4 Normal : NORMAL;
  float4 Priority : ATTR3;
  float4 Direction : ATTR4;
};
```

The addition of one `float4` might not seem significant. However, twice as many vertices are needed, since each stroke is created from four vertices instead of two. This increases the memory requirements to 267 % of the space needed for the line-based scheme.

5.4.2 Drawing Primitives

From the vertex arrays, as outlined in the previous section, the primitives are drawn. There are two basic schemes. The first is a line-based scheme, which uses two vertices per stroke to draw a stroke as line using native hardware line drawing primitives. The second one forms strokes as quad from four vertices.

Native Line Drawing

The simplest and most efficient way of drawing lines is to use the line drawing capability provided by the graphics hardware. Two vertices define a stroke and the line is automatically drawn with a constant width. The vertex program must merely transform the position of each vertex from object coordinates P_o to screen coordinates P_s using the combined model/view/projection matrix $\mathbf{M}_p$:

$$P_s = \mathbf{M}_p \, P_o$$

The line width can be adjusted by a global setting (calling `glLineWidth()` under OpenGL). If line smoothing is enabled, the width can also have non-integral values. However, most implementations only support discrete steps for line width. Even more severe is the limitation to a maximal line width. Some current systems limit this to 10 pixels, which might be to low for some purposes (for example, if running on a display device with more than 200 dpi, 10 pixels amount to a physical line width of 1.3 mm, or less then 3 pt).

Quad-Based Line Drawing

If more freedom is wanted in controlling the appearance of the strokes, a quad must be constructed manually. This enables features such as wider lines, per-stroke variation of the line width, or textured strokes.

Four vertices are passed in, which are attributed in a way to be able to construct a quad as needed. There is no way to run a different vertex program for the individual vertices, every vertex is processes exactly like the others. That is why the position P_o and direction D_o are assigned for the vertices as described in the previous section (recall Figure 5.9).

The position P_o of each vertex is transformed to screen coordinates P_s. Then it is translated by the line width w perpendicular to the transformed stroke direction D_s. This translation needs to take into account the aspect ratio of the viewport, and the perspective divide that is performed after the vertex program.

5.4.3 Density Control

The major purpose of the vertex program is to adapt the density of strokes to the screen area covered by an object. Since a fixed number of strokes is attached to the object's surface, drawing all of them would result in a too dense rendition when the object is too small on the screen. This is because strokes are drawn with a line width independent of the object's distance. Strokes not only get more dense when an object is moved away from the viewer, but also when the surface is tilted. Thus, density control needs to account for both changes in distance as well as inclination.

Each stroke is processed independently. Therefore, the actual density, that is, the number of strokes per given area, cannot be measured in the vertex program. Instead, it uses the known dependency of density on the distance and inclination of the surface. From these attributes a threshold value is derived, that in combination with the stroke's priority value is used to cull a stroke. Only when the priority exceeds the threshold, the stroke should be displayed.

Distance Threshold

To account for changes in distance, a minimal distance d is introduced. If the object is nearer than this distance, all strokes are displayed (if not removed by other thresholds). Starting at distance d, strokes are removed to maintain the density. The calculation of distance is done in camera coordinates. Since in this coordinate system the camera is at the origin, the distance d_P of a vertex to the camera is just the length of its position vector in camera coordinates:

$$\begin{aligned} P &= \mathbf{M}_v\, P_o \\ d_P &= |P| \\ t_d &= \frac{d_P - d}{d_P} = 1 - \frac{d}{|P|} \end{aligned}$$

Here, $\mathbf{M}_v$ is the combined model and view matrix which transforms vertices from object space via world space to camera space. When the distance d_P is smaller than the minimum distance d, the resulting threshold t_d will be less than 0, meaning all strokes are displayed. As the distance gets larger than d, the resulting distance threshold t_d gets proportionally larger. If, for example, $d_P = 4d$, the threshold will be 0.75, so that only a quarter of the strokes are displayed (assuming a linear distribution of priorities).

Slope Threshold

The influence of inclination is derived from the angle between the surface normal N and the view vector V. The visible area is proportional to the cosine of that angle, which can be expressed by the dot product:

$$\begin{aligned} N &= \mathbf{M}_v^{-1^T} N_o \\ V &= -\frac{P}{|P|} \\ t_s &= 1 - |N \cdot V| \;=\; 1 - \frac{|N \cdot P|}{|P|} \end{aligned}$$

The slope threshold t_s will be 0 if the surface directly faces the viewer, meaning all strokes are displayed. The more tilted the surface is, the higher the threshold gets, reaching one at 90 degrees (that is, the surface is viewed edge-on).

Perspective Threshold

The combined perspective threshold t_p which is to be compared to the perspective priority p_p depends on both the distance and slope thresholds. Because we defined the threshold as the fraction of strokes that are removed, rather than the fraction of strokes that are shown, we have to subtract the thresholds from 1 first:

$$\begin{aligned} 1 - t_p &= (1 - t_d)(1 - t_s) \\ t_p &= 1 - (1 - t_d)(1 - t_s) \\ &= 1 - \frac{d}{|P|} \frac{|N \cdot P|}{|P|} \\ &= 1 - \frac{d}{|P|^2} \, |N \cdot P| \end{aligned}$$

Fortunately, the length $|P|$ is squared, which eliminates an expensive square root calculation ($|P|^2 = P \cdot P$). The minimal distance d will be passed in as a uniform parameter, while P and N are the transformed vertex attributes.

5.4.4 Lighting

Thinning out strokes to preserve a certain density independent of the screen size of an object is necessary to portrait a certain fixed intensity. To depict varying lighting conditions, we want to change that intensity. This means we need to adapt the number of strokes to the lighting conditions.

As explained above, a lighting priority value p_l is stored with each stroke. We calculate a lighting threshold value t_l, similar to the perspective threshold value t_p, which is used to eliminate strokes to create the impression of a varying lighting contribution.

We use a very simple lighting model. While more complex lighting models could be employed, they only would result in a different target intensity that is to be depicted. Since we focus on the depiction of an intensity, rather than how that intensity is determined, a simple diffuse lighting model with a directional light source is sufficient.[2]

$$t_l = \max(0, L \cdot N)$$

This sets the lighting threshold t_l to 0 when the surface is perpendicular to the light direction, causing all strokes to be displayed. When the surface is facing the light, the threshold will be 1. This eliminates all strokes with priorities less than 1. Normally, these are all strokes. However, if an indication map is used, priorities can exceed 1, which leads to strokes being displayed even in areas of full brightness. Of course, this is exactly the purpose of indication. Note that this formula assumes dark strokes on a light background. If light strokes were used instead, the proportionality would have to be reversed.

5.4.5 Primitive Culling and Frame Coherence

Once the thresholds are calculated and compared to the priorities, and it is determined that a stroke should be culled, the question remains how to actually prevent the stroke from being displayed. Current GPUs do not offer the functionality of directly discarding a drawing primitive in the vertex processing unit. Therefore, other methods of not showing a stroke, even though its vertices are present, have to be found.

Various approaches are conceivable. There are several stages in the graphics pipeline that can discard primitives or fragments:

- *View volume clipping*: The primitive could be transformed to lie outside the viewing volume, for example by translating it behind the far clipping plane.
- *Triangle setup*: Back-face culling could be enabled. The order of vertices would have to be changed (which is not possible) or they would have to be transformed so the geometric order is reversed.
- *Rasterization*: A primitive of zero area does not produce any fragments. If vertices are transformed to create a degenerate quad, for example by applying a zero line width, it would be invisible. However, when using primitive lines, a width of zero is not allowed by the OpenGL specification.
- *Fragment operations*: The alpha test or depth test can be used to discard fragments. For this, the alpha or depth values would have to be modified in the vertex program.

---

[2] Besides, as Chapter 2 showed, lighting in a drawing is rarely required to be photorealistic anyway.

- *Frame buffer blending*: By enabling the blend function and setting the alpha value to zero, the values in the frame buffer can be preserved.

The earlier in the pipeline the rejection happens, the more efficient this process is. Only the previous stages need to work on all primitives, the latter only process primitives that actually contribute to the image. This means that view volume clipping would be most efficient, and frame buffer blending is the least efficient technique.

Another consideration is frame coherence. By culling a primitive that was visible in the previously rendered frame, or enabling a primitive that was not shown before, a sudden change in the image is introduced. This creates a visual distraction known as "popping". Instead, it is preferable to introduce or discard a primitive *smoothly.*

However, not all of the above stages support a smooth transition in this sense. Basically, only the zero-width method and blending approach allow to continuously adjust the visibility of a stroke.

Since the line width of primitive lines cannot be set to zero, we use alpha blending for the line-based scheme. In the quad-based scheme, however, the line width is only used to calculate the positions of quad vertices, so here a line width adjustment is possible. The alpha value is conditionally set based on the thresholds and priorities in the vertex program:

$$\alpha = \begin{cases} 0 & : \quad p_p \geq t_p \quad \text{or} \quad p_l \geq t_l \\ 1 & : \quad \text{otherwise} \end{cases}$$

When a smooth transition is desired, the alpha value is derived as follows:

$$\alpha = 1 - \min(t_p - p_p, t_l - p_l)$$

Similar to the alpha value in the line-based scheme, the line width is adjusted in the quad-based scheme. The alpha value is derived as above, and the line width w is multiplied by the alpha value:

$$w' = \alpha w$$

This new line width value is used instead of the original, as outlined in the line drawing section.

5.4.6 Optimization

There are a number of possibilities to further optimize the explicit rendering process. For example, strokes need only be drawn if the original model's face is visible. This can be tested using an occlusion query. Also, for solid models, strokes attached to back-facing polygons can be omitted. Depending on how many strokes are applied per face in the model, faces should be grouped in clusters by adjacency and face normal. Rather than testing each face individually, the clusters are tested for backfacingness or occlusion and the corresponding strokes may be skipped.

5.4.7 Stippling

Up to now, only stroke-based renditions have been discussed. In contrast, stippling uses dots as basic drawing primitive. The stipples should be distributed on the screen in an irregular way to prevent the emergence of distracting patterns. In contrast to stroke-based rendering styles, the stipples are placed automatically, rather than in a drawing program.

The stipple rendering process is similar to the stroke rendering process described before. Stipples are attached like particles on an objects surface, represented as a single vertex. Each stipple is assigned a priority value, at run-time this priority is compared to a dynamically calculated threshold. Only if the threshold is exceeded, the stipple is drawn.

Stipple Creation

The creation of stipples on a geometric model and assignment of priority values was developed in joint work with Oscar Meruvia [Pastor et al., 2003]. His off-line stipple rendering system creates a hierarchy of stipple points that is used to determine which stipples to discard if the density gets too high. The distance between neighboring points is used to determine up to which hierarchy level stipples should be drawn.

For the purpose of real-time rendering in this work, we derive threshold values from the hierarchical structure. This reduces the decision whether to draw a stipple or not from a neighbor-distance check (which is not possible in a vertex program since neighbors can not be accessed) to a simple comparison of a threshold value and a priority value. The position, and priority of a stipple is stored along with the surface normal of the underlying surface as one vertex.

Point Drawing

Just like for the strokes, several rendering methods for stipples exist. The simplest, and most efficient one is to use native hardware point rendering mode. This requires only one vertex per stipple. The point size can be directly set in the vertex program. However, as with line width, it cannot be 0, so alpha blending has to be used as well to discard stipples.

There is a maximal size for hardware point rendering as well. If this is a limitation, quad-based stipples can be rendered similarly to the quad-based line drawing scheme. It is somewhat simpler because no stroke direction needs to be taken into account.

5.5 Examples

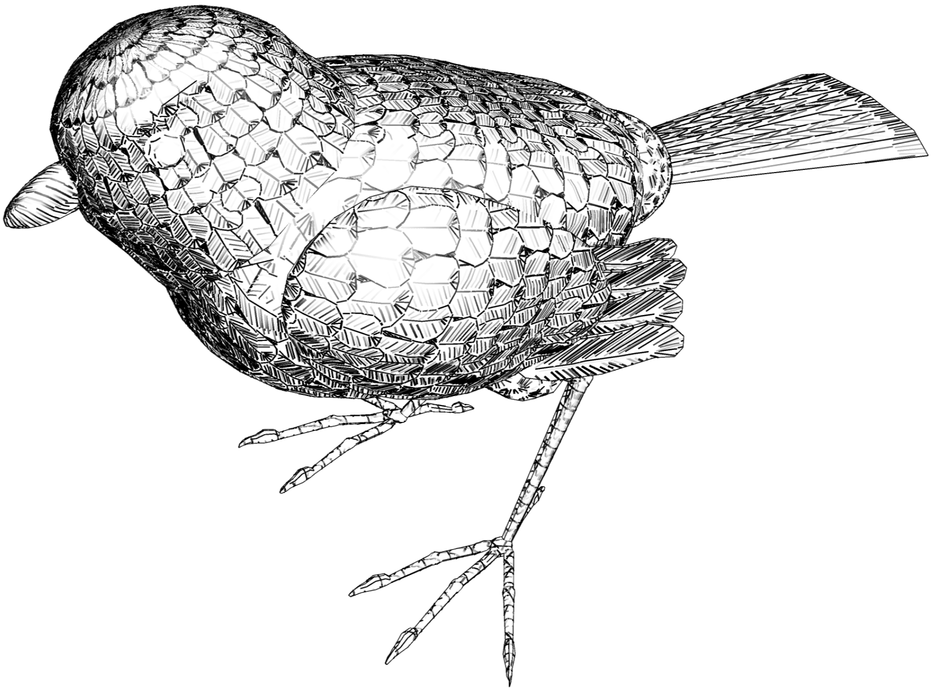


Figure 5.10: Bird, ca. 2,000 faces, rendered using more than 80,000 strokes. Note the smooth blending of strokes.

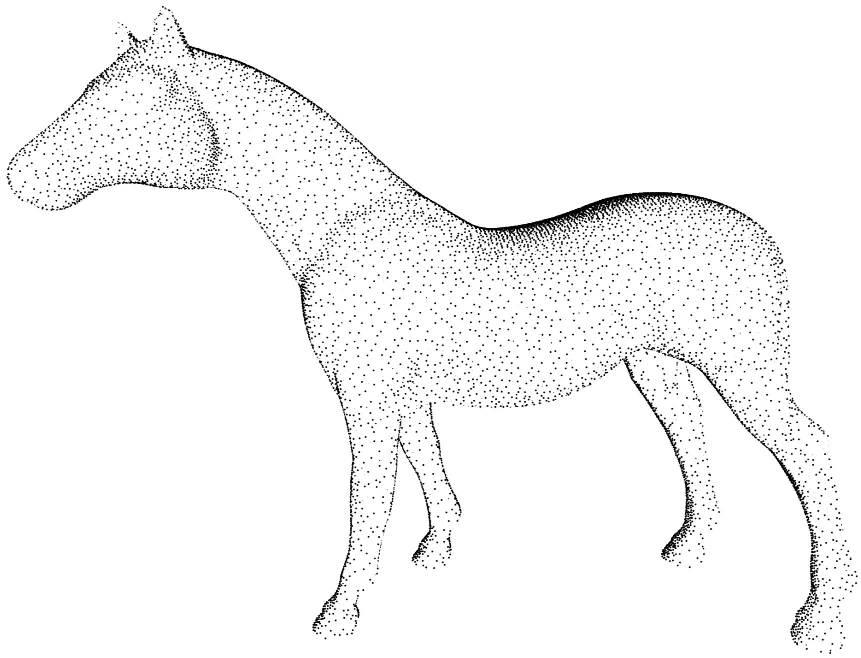


Figure 5.11: Horse, 90,000 faces, rendered with 70,000 stipples at 44 frames per second on a GeForce 4.

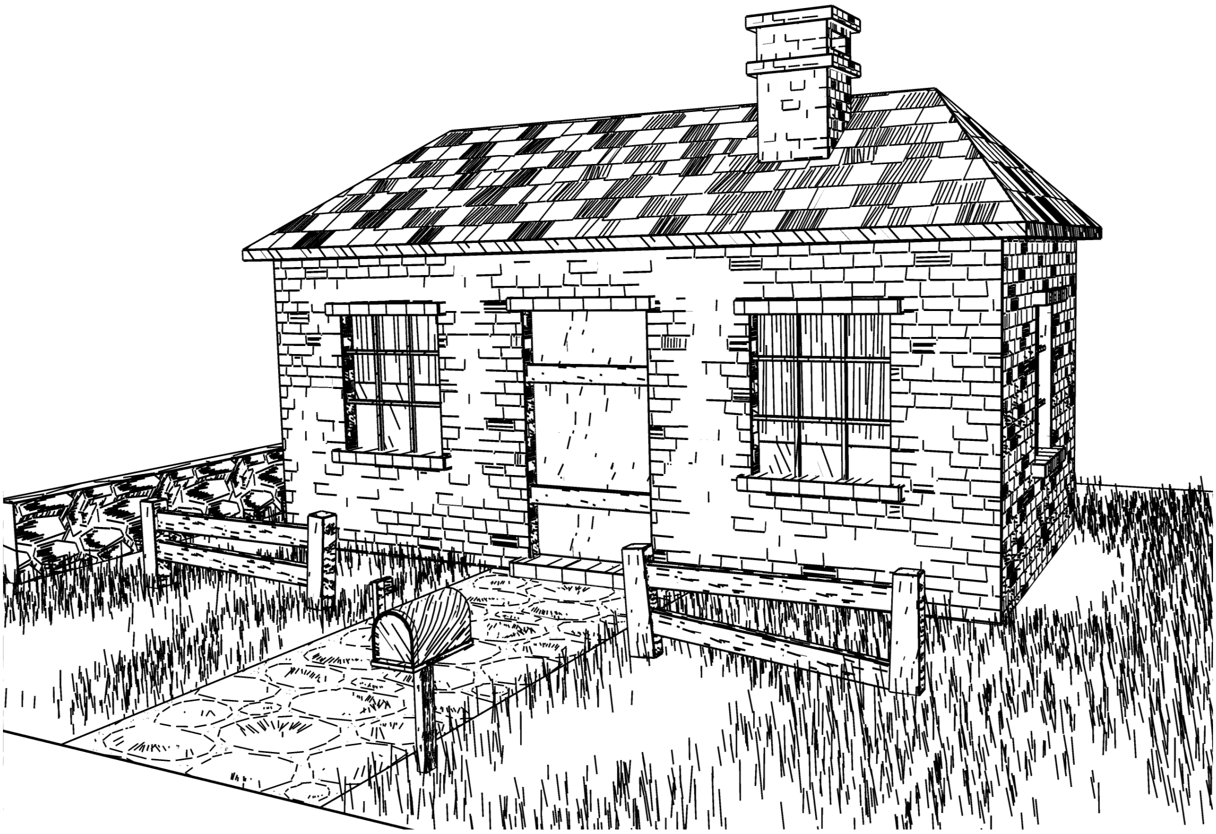


Figure 5.12: House, 90,000 strokes, rendered at 38 frames per second on GeForce 4 (modeled after [Winkenbach and Salesin, 1994])

6 Shading with Implicit Mark Generation

In Chapter 5, explicit mark generation techniques for the shading of surfaces have been discussed. While they exhibit an exceptional level of control over the placement and appearance of individual marks, they suffer from the complexity that linearly depends on the number of strokes. Also, the drawing of disconnected primitives is not well-suited for current graphics accelerators, because if no vertices are shared, the vertex cache is effectively bypassed.

This chapter will present techniques that overcome these inefficiencies, albeit at the price of flexibility. The basic approach will make use of bitmap textures that *implicitly* contain information about where to place strokes. Only when rendering the surfaces with these texture maps the strokes emerge on a pixel-by-pixel basis.

The decision to use bitmap texture maps is mostly motivated by their ubiquitous use in current graphics acceleration schemes. Manufacturers strive for achieving a good balance of geometry calculations and pixel processing. While historically the emphasis has been on geometry processing, recent products (announced in 2002) push pixel processing capabilities much further forward. While in this thesis the main focus lies on the efficient exploitation of mainstream graphics accelerators, Section 6.7 will give a glimpse on what can be achieved with even more processing power at the pixel level.

There are a multitude of problems that will be approached in the following sections. Section 6.1 discusses the use of mipmap filtering for maintaining a consistent mark density even when surfaces are scaled or moved in distance. Stroke-based shading with halftone textures is introduced in Section 6.2, while Section 6.3 describes how the halftone textures are constructed. To provide more control and accuracy, techniques for lighting strokes independently and making the lighting apply to a whole stroke are shown in Sections 6.5 and 6.6. A discussion of advantages and shortcomings of texture-based stroke shading concludes this chapter.

6.1 LOD via Mipmapping

Surface detail in 3D computer games is added via texture mapping. Structures painted into a texture are intended to become smaller with increasing distance. In the case of a pen-and-ink hatching texture, the small structures cause serious moirée patterns. In motion, this produces even worse visual artifacts than in a still image, which prevents frame-coherence.

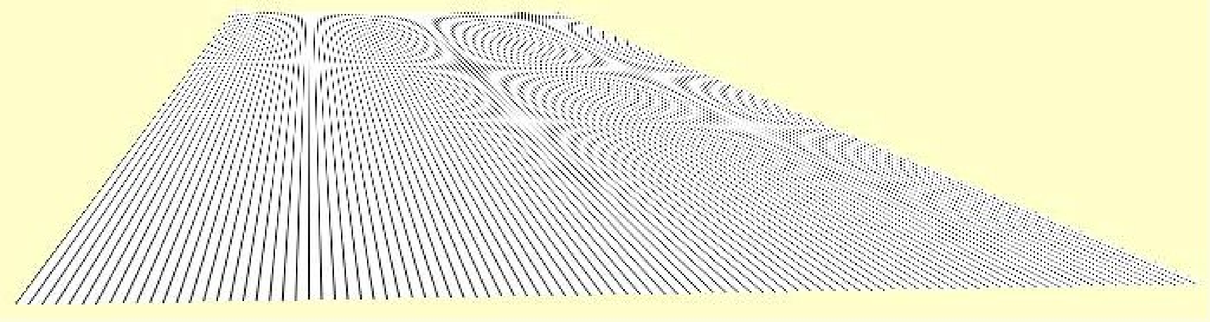


Figure 6.1: Unfiltered hatching textures cause moirée patterns.

Figure 6.1 shows a single tilted rectangle covered with a repeated black-and-white pattern choosing the nearest texel's color for each pixel. The texture scale is adjusted so that in the front, the texture is drawn at its original size, while in the back, it is minified due to perspective foreshortening.

The usual antidote for such problems is mipmapping [Williams, 1983]. Mipmaps consist of a series of textures decreasing in size. The graphics hardware automatically chooses a mipmap level so no texture minification occurs, on a per-pixel basis. The mipmap levels are constructed by scaling the original texture down in a preprocessing step, applying more or less sophisticated filtering techniques. Unfortunately, the normal filtering process applied to the large white areas and relatively thin black lines of a hatching texture causes the lines to vanish into gray soon (Figure 6.2). All visible surface structure is lost.

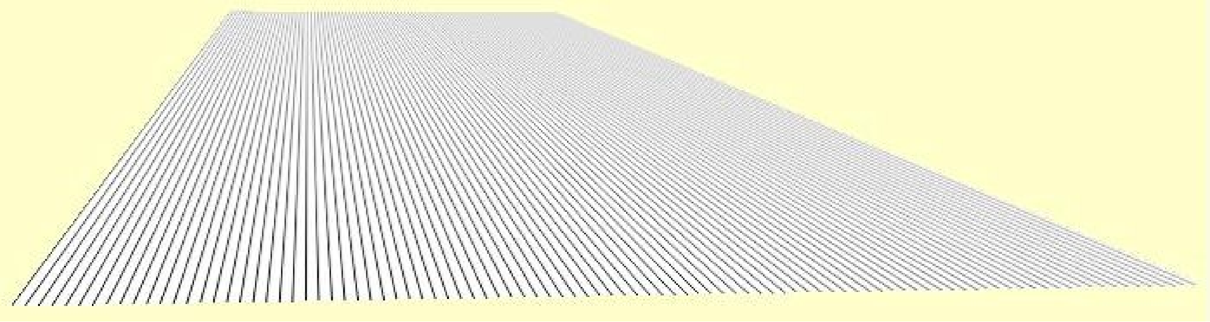
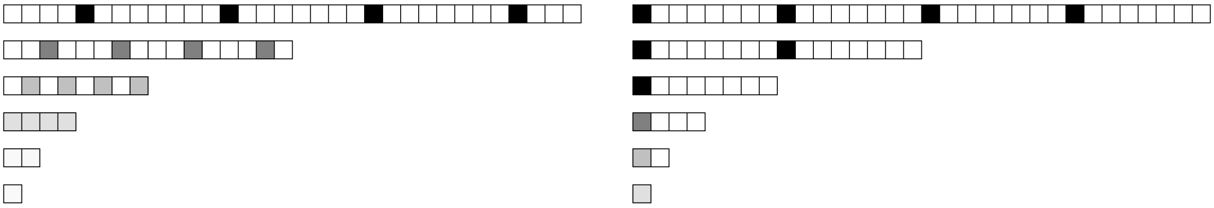


Figure 6.2: Normal mipmap filtering fades hatch lines into light gray.

So, at a first glance, textures seem to be unsuited for implementing pen-and-ink detail. But fortunately, the graphics hardware allows us to set each mipmap level independently. It does not care whether the individual mipmap levels really are “nicely” filtered-down versions of the original image. We would need to construct the mipmap levels in a way that maintains the desired gray-level by spatially distributing black ink on the white background instead of locally raising the gray-level of the texels.



(a) Normal mipmap filtering by averaging (b) Hatch-maps

Figure 6.3: One-dimensional textures used for parallel hatching.

6.1.1 One-dimensional hatchmaps

This is indeed possible. The key idea is that mipmapping essentially displays textures at constant size in screen-space. That means, a one-texel-wide line drawn into each mipmap level will be displayed roughly as a one-pixel-wide line on the screen, independently of the actual object size.

Figure 6.3 demonstrates the process of constructing one-dimensional *hatch-map* levels that are used as hatching primitive. On the left, the normal mipmap filtering scheme is demonstrated. As can be seen, the lines are black and well-separated in the top level, but they become lighter and closer to each other in the lower levels. On the right, the hatch-map construction scheme is shown. It uses equally-spaced black hatch lines for all mipmap levels. Only in the very lowest levels that are smaller than the desired on-screen line spacing, gray values are used to maintain the tone. Another possibility would be to leave them white, which means that no detail will be shown in the distance.

Texture coordinates for game objects are usually assigned interactively in a modeling tool. Hatch-map coordinates are in no way different, except that only one coordinate actually gets used. An object needs to be entirely covered by a repeating texture for hatching. In practice, we use a checker-board pattern in the modeling program, while at run-time it is substituted by the hatch-map.

For hatch-maps to work, the texture must be scaled so that it is always minified, because the mipmapping hardware only allows to specify minification textures. On the other hand, to cover a wide depth range, we need many mipmap levels. We adjust the texture scale for an object while modeling, so that the texture appears at original size when the object is at its closest distance to the viewer. The result of applying the constructed hatch-map is shown in Figure 6.4.

It can be seen that our hatch-map approach works as expected. Less lines are used in the distance, each line is clearly distinguishable. However, though the over-all tone is maintained, it is not continuous. There is an abrupt switch of hatch-map levels, causing discontinuities in line-width and hatching density.

To overcome this limitations we use tri-linear texture filtering. This is a special texture hardware mode that smoothly blends texels from two adjacent hatch-map levels. Since hatch-map levels are constructed such that lines in one level, when displayed, are spaced

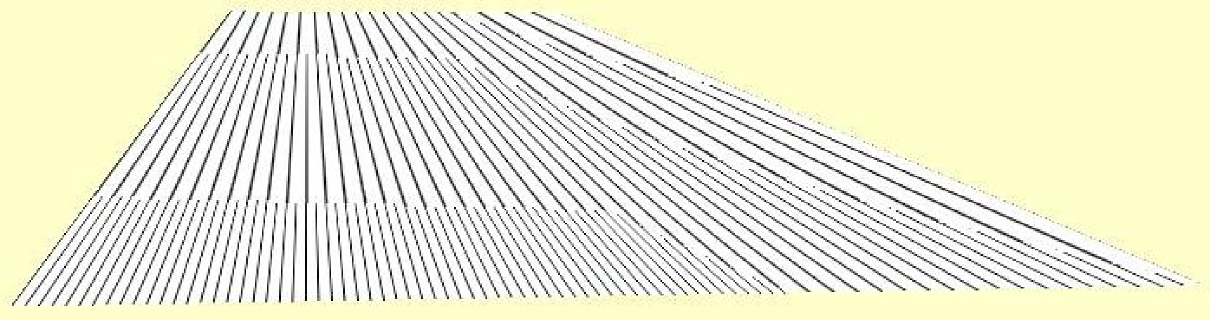


Figure 6.4: Hatch-maps in bi-linear texture filtering mode. Four hatch-map levels can be distinguished.

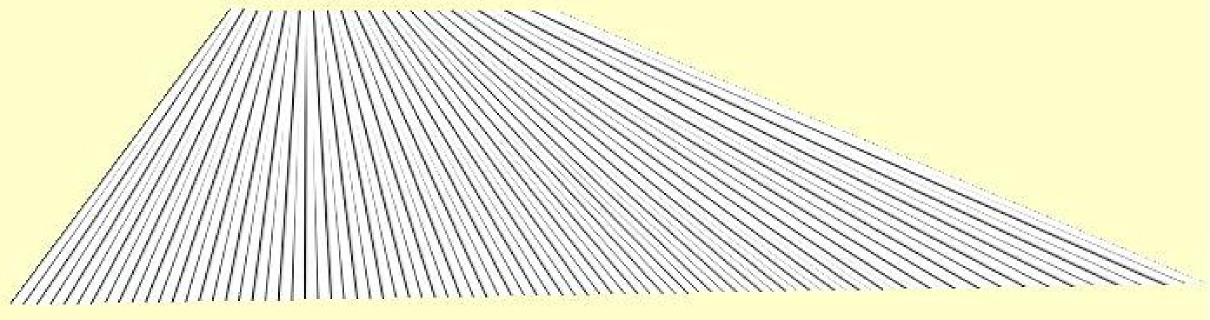


Figure 6.5: Hatch-maps in tri-linear texture filtering mode. Strokes are faded out smoothly.

exactly twice as wide as in the previous level, the lines fit exactly in-between. This creates the highly desirable effect of smoothly thinning-out hatch lines while maintaining the overall brightness of the image (Figure 6.5). This is the key for the frame-coherence inherent to our approach.

To give an example, assuming the desired hatch line spacing on screen is 8 pixels, the lower 3 mipmap levels of widths 1, 2, and 4 cannot be used. Therefore, a 2048×1 texture gives an usable depth range of $1 : 2^{11-3} = 1 : 256$. The texture could be even larger than that, because, fortunately, one-dimensional textures do not use much texture memory. However, most graphics hardware limits the maximal texture extent (see Section 3.7, p. 29). Also, to extend the depth range, the texture scale could be dynamically varied depending on the object's distance to the viewer. Another possibility is to generate the texture procedurally (Section 6.7).

6.1.2 Two-dimensional Ink Maps

While maintaining a constant stroke width and spacing for parallel hatching is a rather mechanical process, adding other detail to surfaces is artistically challenging. We therefore now construct hand-drawn *ink-maps* that utilize the same hardware-mipmapping technique like the one-dimensional hatch-maps, but this time in two dimensions.

The desired surface structure is manually drawn into a texture map. When this texture is displayed with normal average mipmap filtering, similar problems to the hatch-map problems mentioned above arise. The moirée effect is not quite as disturbing because of the larger structure size, but again, worse in motion. Of course, the drawing rapidly blends into the white background when mipmapping is applied.

Automatically adjusting the line spacing as we did for the hatch-maps is not possible, because the lines are not evenly spaced. We therefore for now try to maintain the stroke width only, without considering the change in tone. Unfortunately, our initial attempts at automatic filtering gave unsatisfying results (see Figure 6.6).

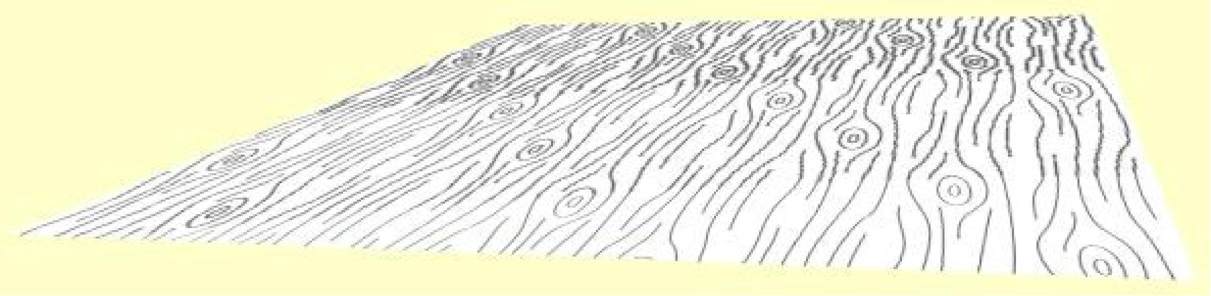


Figure 6.6: Ink-map levels automatically constructed by minimum filtering.

We used a filter that selects the darkest texel out of the four parent texels, instead of the average intensity. While the stroke width is indeed roughly maintained, the lines become fuzzy. Although more sophisticated filtering methods might provide better results, this direction of bitmap-based filtering does not seem too promising.

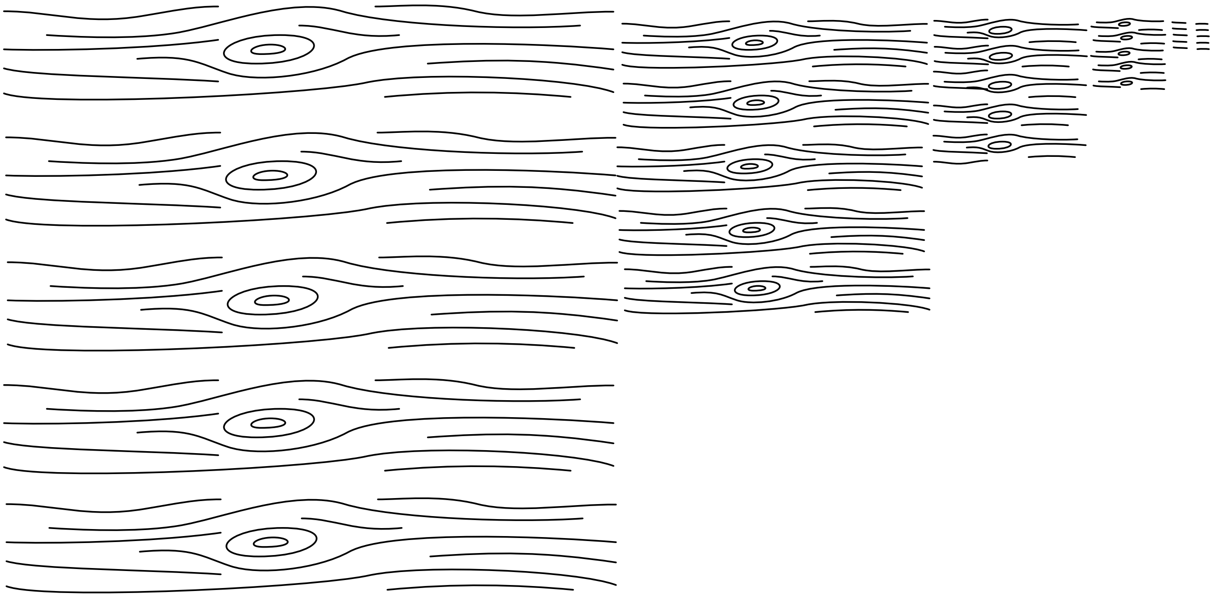


Figure 6.7: Ink-map example.

Instead, we now manually construct a series of mipmap levels for the ink-maps. This also allows an artist more expressiveness than any automatic method can provide, because the artist can control exactly how an ink-map should look like in the distance. The ink-map was drawn again, this time in a vector graphics application. Down-scaling the drawing by 50% maintained the line width. In the lower scale levels, lines were gradually removed until no lines remained. An example of an ink-map constructed like this is shown in Figure 6.7, while Figures 6.8 and 6.9 show renderings of a rectangle with the ink-map applied.

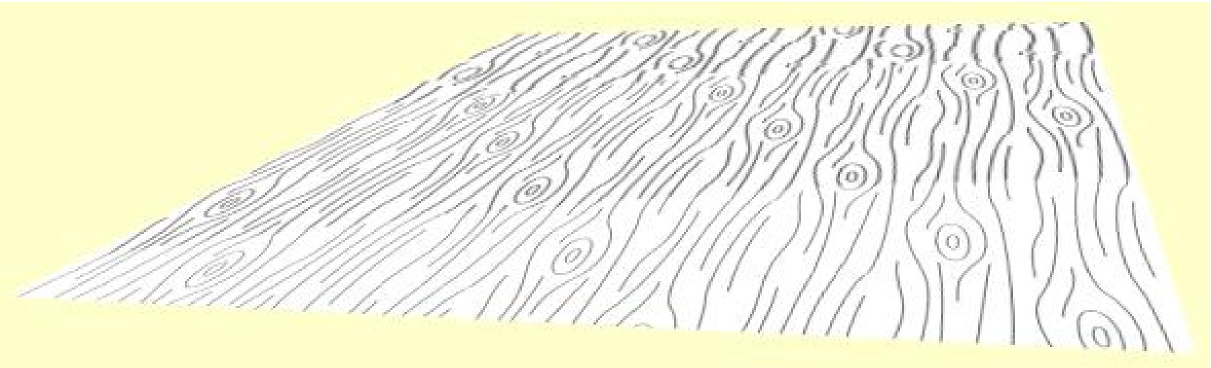


Figure 6.8: Rendered levels in a manually constructed ink-map.

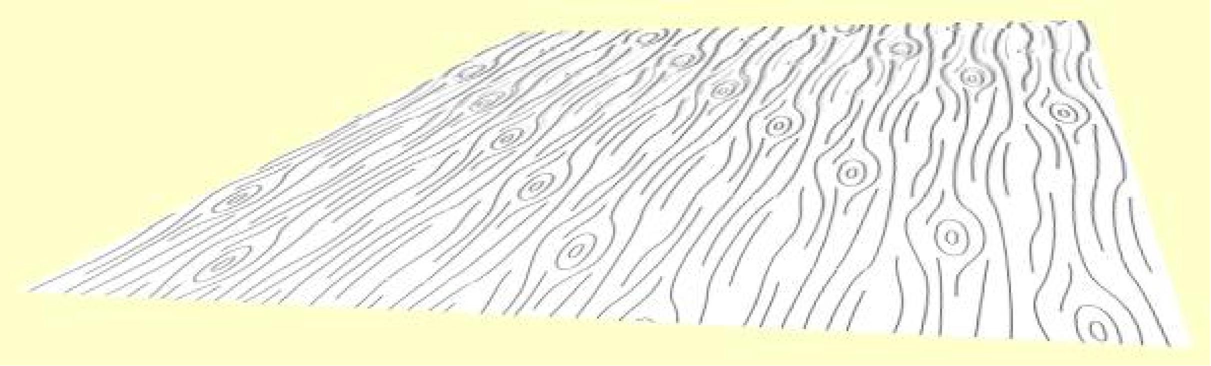


Figure 6.9: Tri-linear filtered ink-map. Note how in the background lines are faded out to preserve the overall tone.

With carefully crafted ink-maps, a smooth animation can be achieved. Frame-coherence is maintained by using tri-linear filtering which blends different ink-map levels into each other. First acceptance tests indicated that the gradual appearance of detail as the viewer approaches an object is a valuable feature of our illustration style.

6.1.3 Quality Considerations

In the case of a very steep viewing angle, anisotropic texture filtering (enabled via the texture_filter_anisotropic OpenGL extension) noticeably improves the image quality. Another factor for better image quality is enabling full scene anti-aliasing. This increases the virtual screen resolution, which causes the wrong mipmap level to be chosen. We compensate this by biasing the level computation using the texture_lod_bias OpenGL extension.

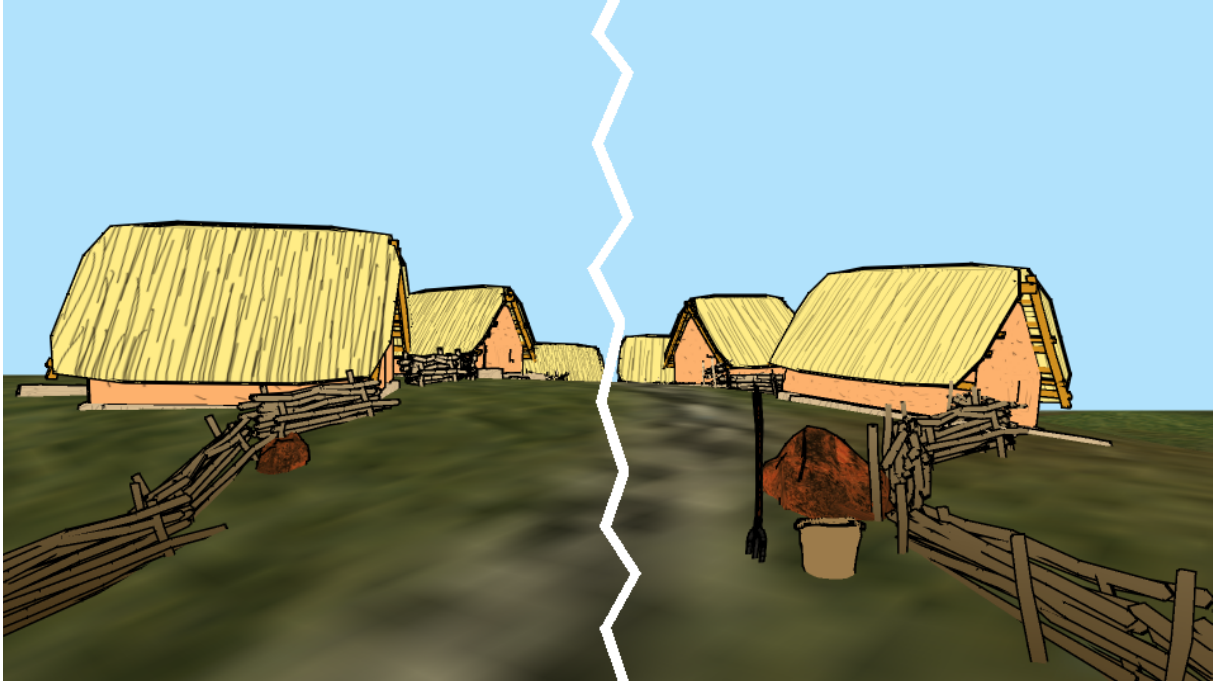


Figure 6.10: Example scene rendered with hatch-maps. In the left half, normal mipmaps are used. On the right, hatchmaps are employed.

6.2 Shading

We have seen in the previous section how a constant apparent lighting intensity can be achieved with hatching strokes, while still preserving detail when an object changes its size on the screen. By choosing a different texture, the apparent intensity can be changed for the whole object. However, no actual shading information can be conveyed because the texture is fixed.

6.2.1 Combining Surface Color and Strokes

One possibility to create an impression of shading with ink-maps is to use conventional interpolated shading for the surface and superimpose the ink-map on top. The shading can be done by using standard vertex lighting as provided by OpenGL. The ink-map is applied as before, but the texture environment is configured to *modulate* the interpolated color, instead of replacing it. Since the ink-map is black at strokes and white otherwise, it will not alter the surface color in between strokes. The strokes will appear black or fade into the surface color.

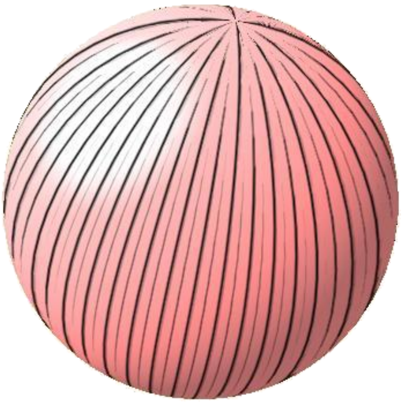


Figure 6.11: Conventional shading combined with inkmaps.

An example of this approach is shown in Figure 6.11. The strokes are visible best when there is a high contrast between the surface and the strokes. Therefore, we chose unsaturated, light colors for surface shading. To give the surface a relatively light appearance even on the side opposite to the light, we add an emissive term to the material definition. For the example, an ambient and diffuse color of (1.0, 0.5, 0.5) was used, the emissive color was set to (0.5, 0.3, 0.3). An advantage of this method is that it will run on any OpenGL accelerator, as it uses only standard OpenGL functionality.

6.2.2 Halftoning

While the combined strokes and shaded surfaces convey a great sense of shape and lighting, they are not stroke-based shading techniques in the sense of our definition. The surface intensity is depicted by varying the overall brightness, rather than the ratio of dark to light regions. Thus, the rendering style is not quite what we aimed for.

To use stroke-based shading with textures, the appearance, width, or number of strokes must be dynamically varied with the lighting intensity. As discussed by [Lake et al., 2000], one way to do this is to create several textures of different brightness and switch between them depending on the current intensity. An improvement on this, which was developed independently of our approach presented next, is to use linear interpolation between textures of different brightness [Praun et al., 2001]. The weights for each texture level are calculated at the vertices, and smoothly blended across the surface.

Both of these approaches use different textures that contain stroke patterns for specific intensity levels. However, there is a certain amount of redundancy in this representation. Both Lake et al. and Praun et al. carefully put all the strokes that are shown in a lighter level into all darker levels, too, to ensure frame coherence. A more efficient way would be if the texture changed its appearance according to lighting, rather than requiring a new texture for this.

There is a technique based on images that does something very similar: digital halftoning. It is used to create a black-and-white representation of a continuous-tone image using a threshold image (the halftone screen), usually because printing processes only support one ink [Ulichney, 1987]. The threshold image is constant and usually replicated to match the size of the input image, whereas the input image to be halftoned varies in intensity. For each pixel in the input image, the input value is compared to the threshold value from the halftone screen. Depending on whether the threshold was exceeded or not, a white or black pixel is generated in the output image. The resulting image is black and white and exhibits varying intensity by varying amounts of black pixels.

Our basic idea, and one of the major contributions of this work, is to use this idea of halftoning and apply it to real-time rendering. We will create halftone screens as textures on object surfaces, and use thresholding operations accelerated by the graphics hardware to create spatially shaded renderings from this.

There is an important difference in the goals of traditional halftoning and this work. Halftoning seeks to reproduce the tone of a given image as closely as possible. This is expressed by the tonal reproduction curve of halftoning algorithms, which should be as linear as possible, and faithfully portrait even small changes in the tone. Also, it should not depend on the local properties of the halftone pattern, which would lead to visual artifacts. In contrast, in this work the visual artifacts (strokes) created by the halftoning pattern are valued themselves, while the linearity of tone reproduction is of minor interest. Also, since the target tone is determined by lighting, it rarely exhibits high-frequent detail that needs to be conveyed. However, the fidelity of our method solely depends on the actual halftone screens that are used. With properly equilibrated screens, a more linear tone-reproduction is achievable (see Section 6.3.4).

Two schemes for real-time halftoning were implemented. A binary threshold scheme will be discussed in Section 6.2.3, followed by the smooth threshold scheme in Section 6.2.3. A variety of rendering styles can be implemented with halftoning by constructing specific halftone screens, as explained in Section 6.3.

6.2.3 Binary Threshold Scheme

The first scheme for halftoning will mimic the classic halftoning process. A binary choice of generating a black or white pixel is made. The choice depends the comparison of the target intensity value for a pixel with the value of the halftone screen at that pixel.

Add discussion of why an "inverted" halftone screen was chosen. Traditionally, higher screen values (which display as white when viewed as image) mean a higher ink priority (black ink)

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 Sections 6.4 or 6.5 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
```

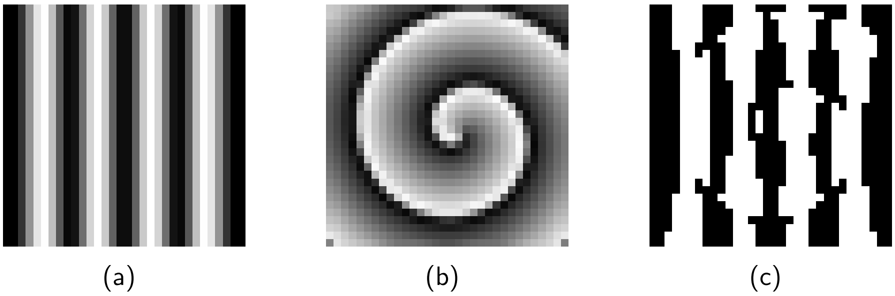

(a) (b) (c)

Figure 6.12: Thresholding a stroke map (a) against a light value (b) results in a halftoned rendering (c) that exhibits strong aliasing artifacts.

so we can use an implementation like

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

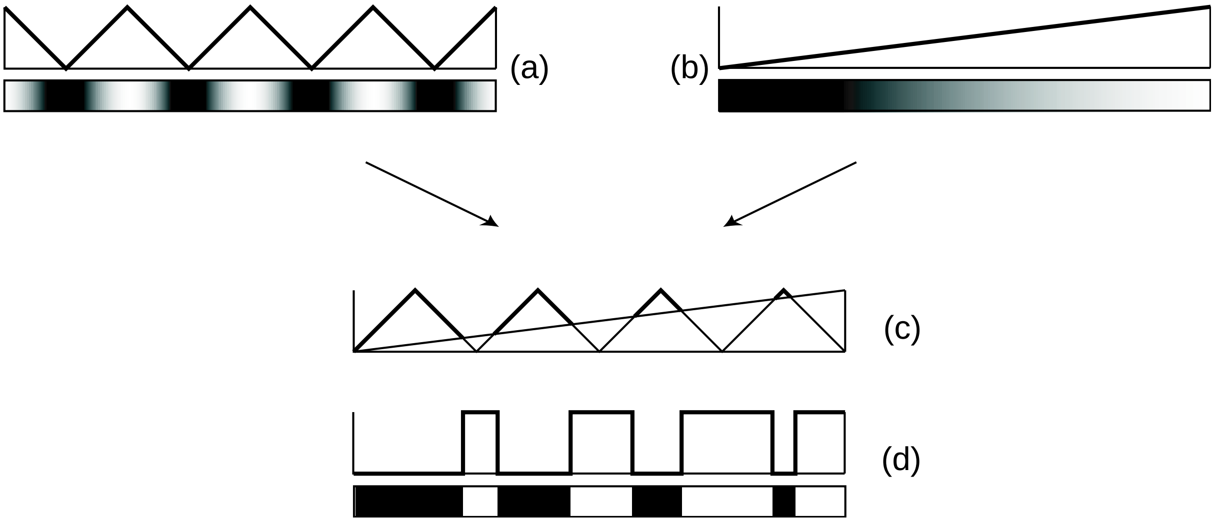


Figure 6.13: For the binary threshold scheme, a halftone screen (a) is compared to a target intensity (b). If the halftone value is greater than the threshold (c), a black fragment is generated, otherwise a white fragment (d).

This works indeed well and fast. With 8 bit halftone screens 256 lighting levels can be represented. However, the binary approach 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.

6.2.4 Smooth Threshold Scheme

To overcome the deficiencies described in section 6.2.3 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.

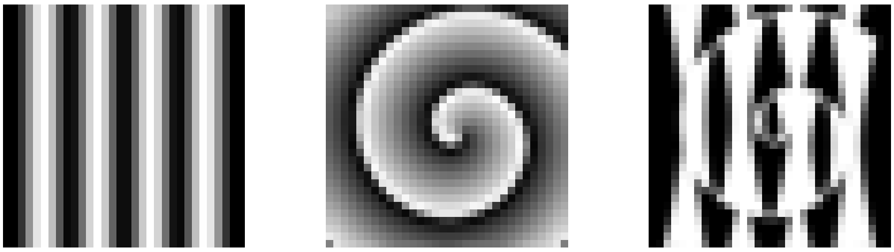

(a) (b) (c)

Figure 6.14: In the smooth threshold scheme, using the same stroke map (a) with the same lighting (b) results in a higher-quality halftoned rendering (c).

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 6.15). The result is clamped by the hardware to the 0–1 range. Applying this function instead of a binary threshold results in an anti-aliased image, as illustrated in Figure 6.14(c).

6.2.5 Discussion

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.

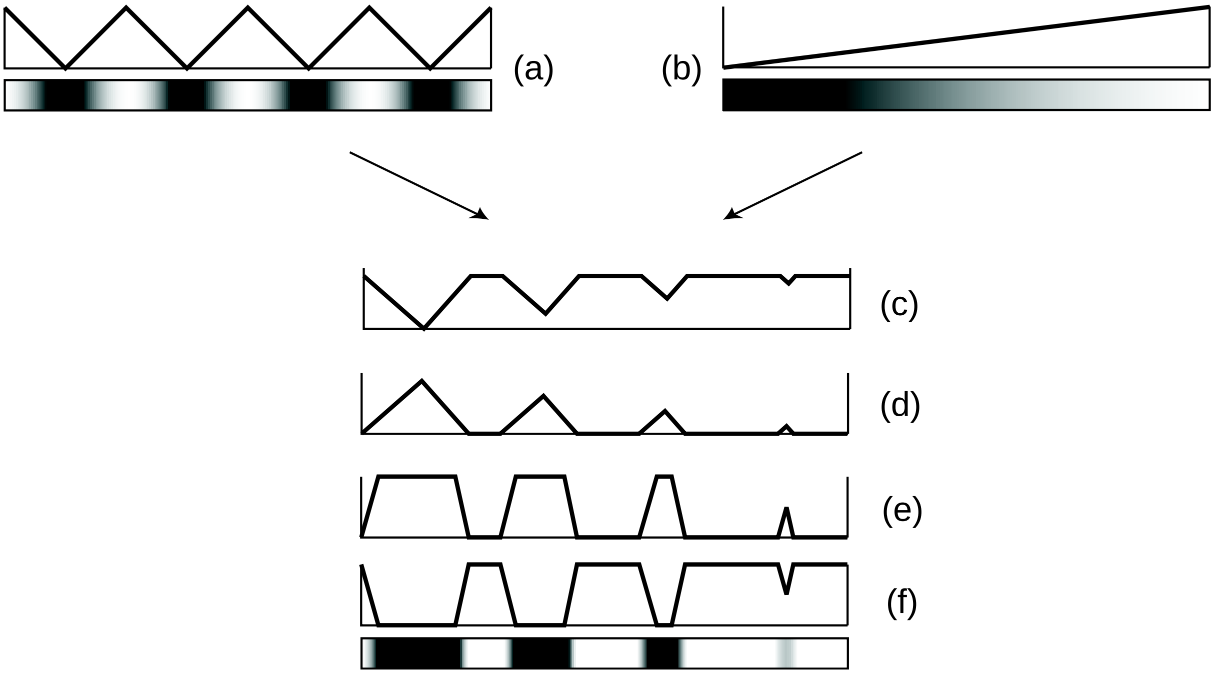


Figure 6.15: The smooth threshold scheme takes the halftone texture (a) and lighting intensity (b) and calculates the sum of both (c). This sum is inverted (d), scaled by a constant (e), and inverted again (f).

6.3 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. There are variations, like stipple dots instead of lines, but basically either the number or the size of drawing primitives is changed, or both. We show some methods for how to construct halftone screens for these styles now, and then discuss the implications for the tonal fidelity of the rendition.

6.3.1 Procedural Screens

The most basic principle, and the one used in most related work, is to construct the textures used for shading procedurally. For example, [Lake et al., 2000] use a few hand-drawn sample strokes that are replicated in a random fashion to create textures of differing intensity. Similarly, [Praun et al., 2001] generate random strokes analytically. They apply a heuristic to ensure an even distribution of strokes. To maintain coherence between mipmap-levels and levels of darkness, all strokes generated for one level are re-used in all larger or darker levels.

Given a procedure that is able to construct textures of differing intensity, we can apply the schemes for halftone-screen construction outlined in the next sections. For example, a procedurally generated halftone screen was used to create the engraving-like appearance of Figure 6.16. However, in the scope of this work we focus on textures created by artists. This is necessary because we do not limit the drawing style to hatching with its more or less regular patterns.

Figure 6.16: An engraving-like appearance with lighting-dependent line widths can be implemented using a procedurally generated halftone screen. Here, a simple one-dimensional screen with a triangular pattern like shown in Figure 6.15 was employed.

6.3.2 Bitmap-Based Screens

One method of creating halftone screens would be to draw them directly as gray-scale image. In fact, any known approach to construct halftone screens could be used, promising candidates would include [Ostromoukhov and Hersch, 1995] or [Veryovka and Buchanan, 2000]. However, the layer-based method presented here is well-suited for manually creating these screens.

The input to the halftone screen encoding process is a set of bitmaps of independent stroke layers. These are drawn in a painting package using black ink and anti-aliased strokes on a transparent background. There are no particular restrictions for the application used, we employed Photo Shop with a digital tablet.

The layers should be drawn with the final appearance of the texture under various lighting conditions in mind. The first layer should contain only few strokes that are most important for the visual appearance. This layer will always be displayed except for the very lightest regions. Subsequent layers use strokes that darken the image, usually either by adding detail, or by hatching in the very darkest layers.

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 6.17.

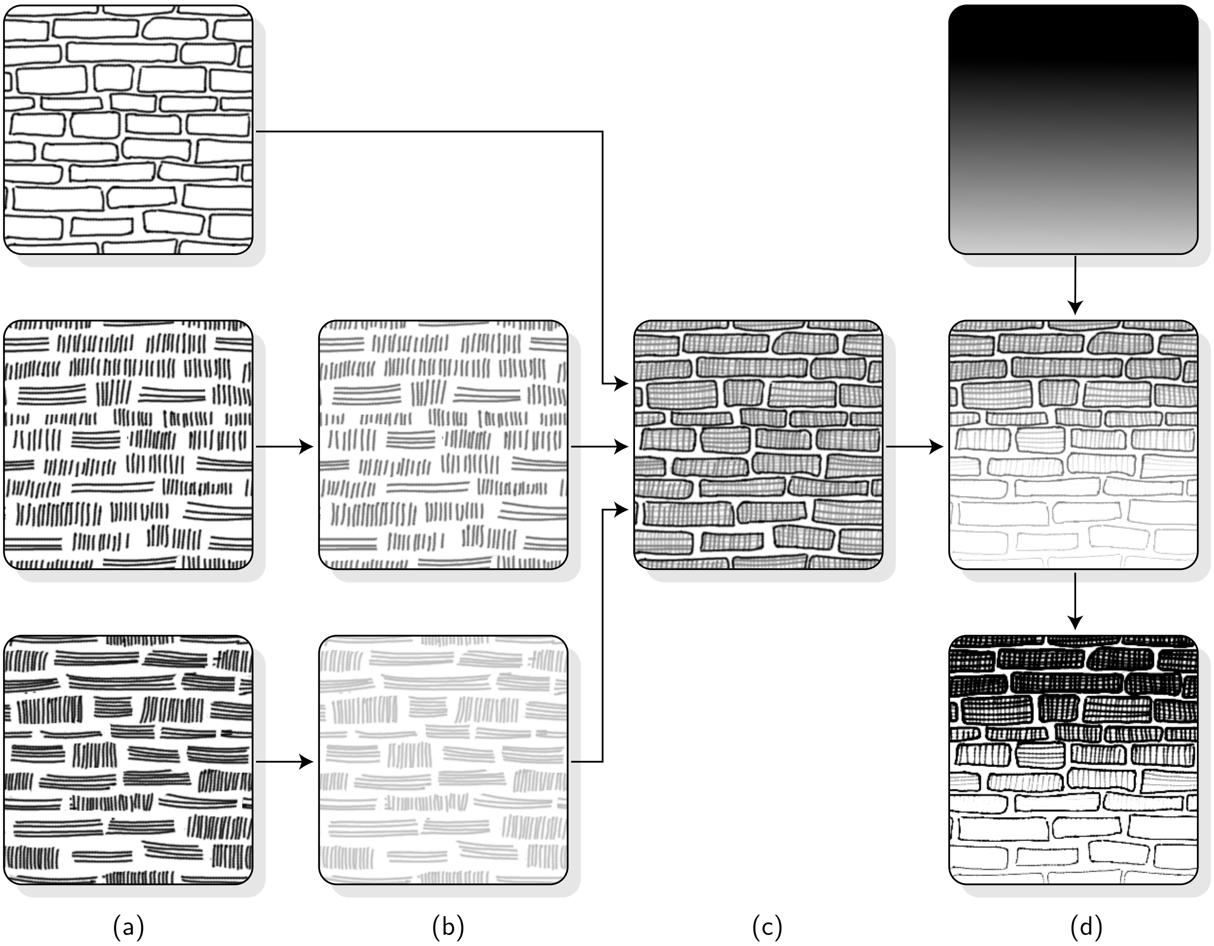


Figure 6.17: Strokemap Scheme. A stroke map is created from a number of hand-painted bitmap layers (a). The colors in each successive layer are lightened (b) and composited into one strokemap (c). At run-time, the strokemap and lighting intensity are added and scaled towards black (d).

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 contradictory, it should become obvious by examining Figure 6.17.

6.3.3 Vector-Based Screens

The bitmap-based scheme works fine for single halftone screens. However, when multiple screens need to be created, as is the case for mipmap levels, it is very hard to accurately draw the same stroke in different sizes of a texture. But this is mandatory to maintain coherency when an object undergoes an animation that involves multiple mipmap levels.

To ensure coherence, we use analytic stroke descriptions instead of bitmaps as input to the screen creation process. The Texture Drawing Tool presented in Section 5.2.1 was extended to generate bitmap screens.

A series of mipmaps is generated and stored. To create smaller images, the distance priority value is used to select the strokes shown in each mipmap level. Inside each level, the lighting priority of each stroke is represented as a gray value, with black meaning highest priority. See Figure 6.18 for an example map.

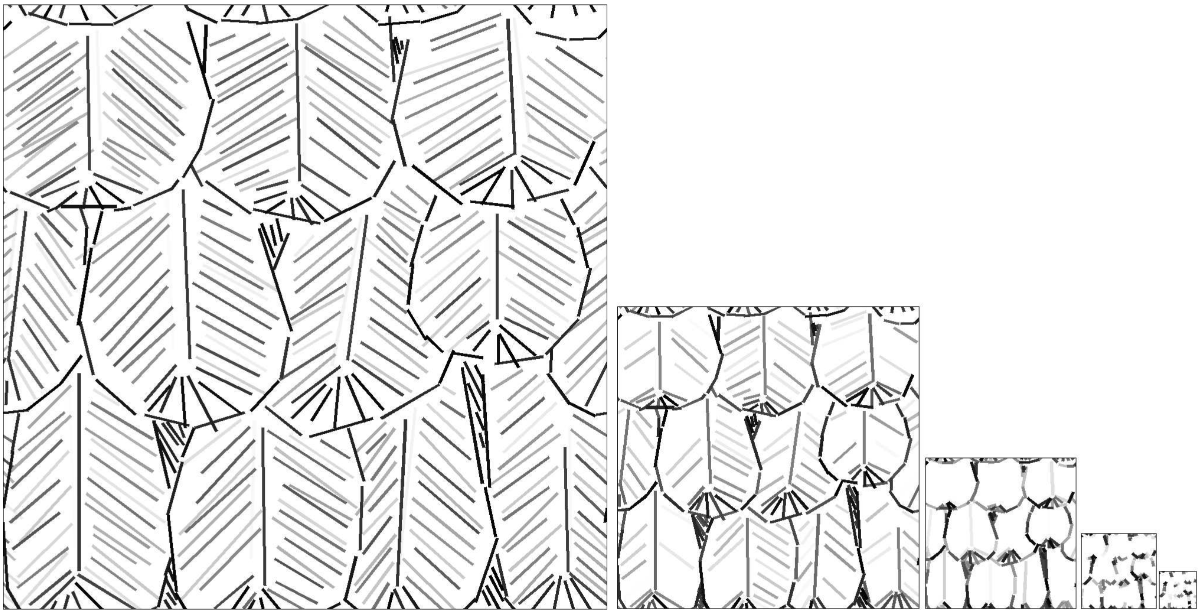


Figure 6.18: Example for a halftone mipmap sequence generated from analytic stroke descriptions. Lower levels are just white and not shown.

6.3.4 Fidelity of Tone Reproduction

Except for the width-modulated procedural screens discussed above, the sparseness of the other stroke-based screens leads to a rather irregular tone reproduction. This is fine for our application of non-photorealistic rendering, and can even be exploited by an artist to adjust the tonal reproduction by modifying the halftone screen at will. But if tonal fidelity is a goal, a more linear tone reproduction curve should be used. This can be achieved by equilibrating the halftone screen, for example using the method described in [Ostromoukhov and Hersch, 1999]. Also, if small local changes in the target intensity need to be conveyed, a finer pattern with smaller strokes can provide the necessary spatial resolution.

6.4 Indicating Detail

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 and Salesin, 1994] and it adds greatly to the non-technical look of these kind of renderings.

For the bitmap-texture scheme, we implement 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. When rendering, this value is added to the target intensity, before applying the threshold operation as explained above. An example indication map is shown in Figure 6.19.

Figure 6.19: Indication maps used for the house in Figure 6.21.

The implementation of this is straight-forward and works even on first-generation GPUs. The first texture combining stage adds the lighting and the indication value, the second stage carries out the smooth thresholding. The result of this technique can be seen in Figure 6.21.

A slightly more complex alternative to this formulation is to treat the indication map as an upper bound on the intensity a surface may exhibit. This only influences light regions, whereas the addition of a signed value as described before even darkens dark regions. This can be implemented by using the minimum of the lighting value and the indication value as threshold. However, to calculate a minimum, two additional texture combining stages are necessary and the use of vendor-dependent extensions (for example, using the CND statement in GL_ATI_fragment_shader or the MUX operation in GL_NV_register_combiners).

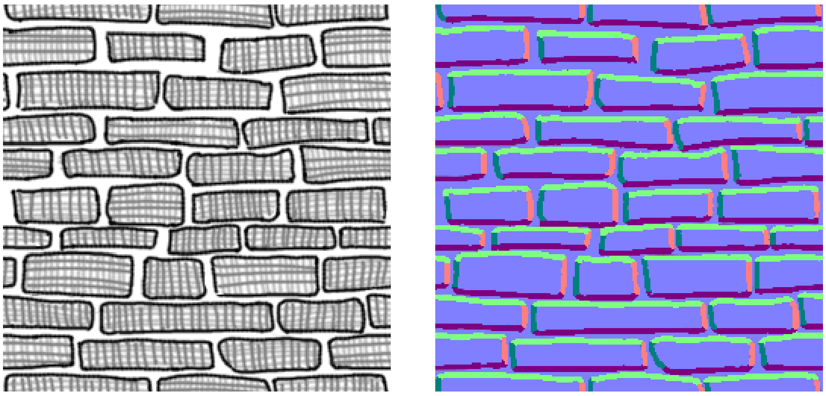


Figure 6.20: Stroke map and corresponding normal map for stroke lighting. The result can be seen in Figure 6.21.

6.5 Individual Stroke Lighting

In the explicit mark generation scheme, every stroke can be lit independently because it has an individual normal vector. Since we use per-pixel arithmetic to evaluate the threshold function, 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.

In Figure 6.20, a hand-drawn normal map is shown. Normals are stored in tangent space so $(x, y, z) = (0, 0, 1)$ is perpendicular to the surface and encoded as colors (light blue in this case). The upper part of the outline is drawn in light green which is equivalent to the y axis, and so on. The light vector is transformed into tangent space using a vertex program and encoded using the same coloring scheme. The texture combiners are configured to take the dot product of the normal sampled from the normal map and the interpolated light vector, which yields a very simple but effective diffuse lighting model. The lighting value thusly calculated is fed to the thresholding as before.

The result of individual stroke lighting can be observed on the brick outlines in Figure 6.21: 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 6.21 can only be viewed on GeForce3 or equivalent hardware.

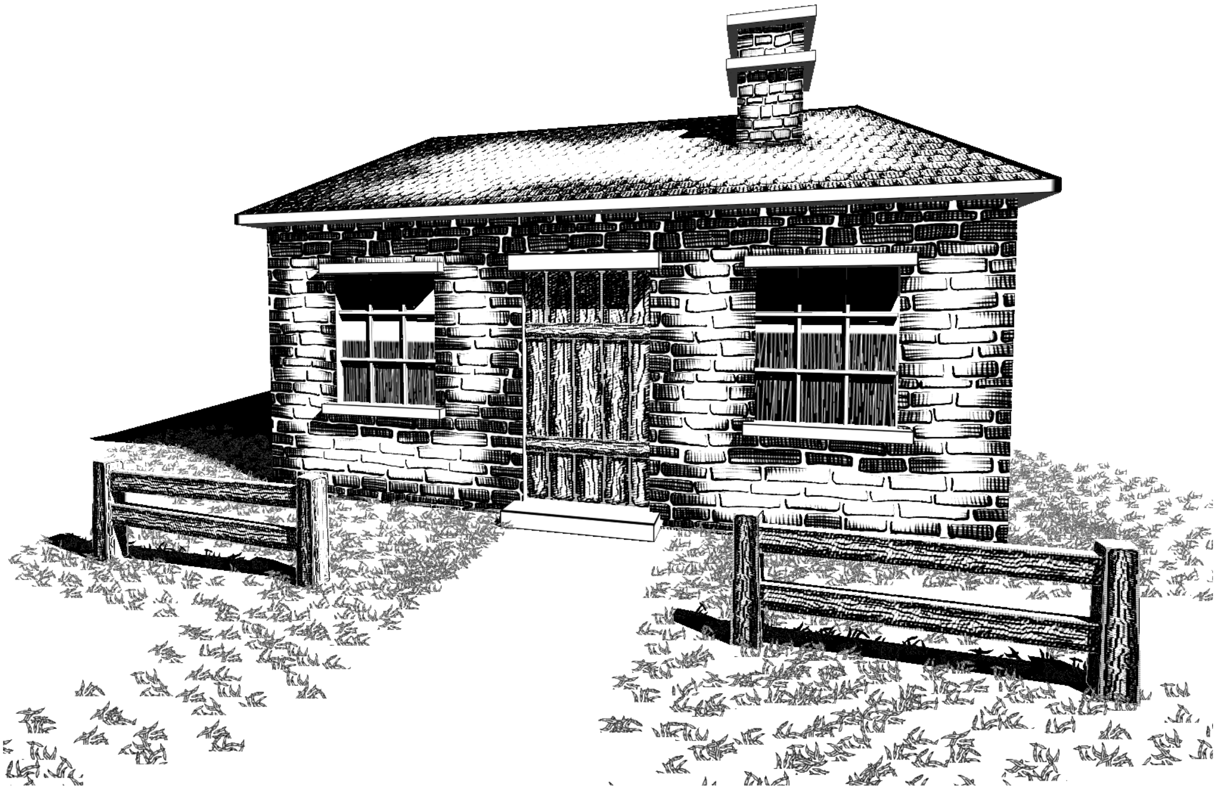


Figure 6.21: Smooth thresholding, indication mapping, individual stroke lighting, shadows, particle grass

6.6 Lightmap Warping

When the lighting smoothly varies across a surface, a mismatch between explicit and implicit halftoning arises. This is because in the explicit method, either a whole stroke is drawn or nothing at all. In the implicit scheme, partial strokes are displayed when one part of the stroke is in a lighter area then the rest.

To make all pixels in a stroke behave the same way, they need to share a single threshold value. They must not use the threshold at their location but rather that at a reference location for the stroke. We implement this by introducing a *warp texture* that replicates the lighting from one point on the surface to all texels covered by a stroke (see Figure 6.22). For each texel, the warp map contains the coordinate of the reference texel encoded as color. The texturing hardware is configured for dependent texturing[1]. All texels in a stroke use the same color-coded texture coordinate, so they will reference the same texel in the light map. Since both the threshold and the lighting value is the same for all texels in the stroke, they will behave identically.

The warp map is created along with the halftone map from the analytic stroke descriptions using the Texture Drawing Tool.

---

[1] Dependent texturing: see Section 3.5.1

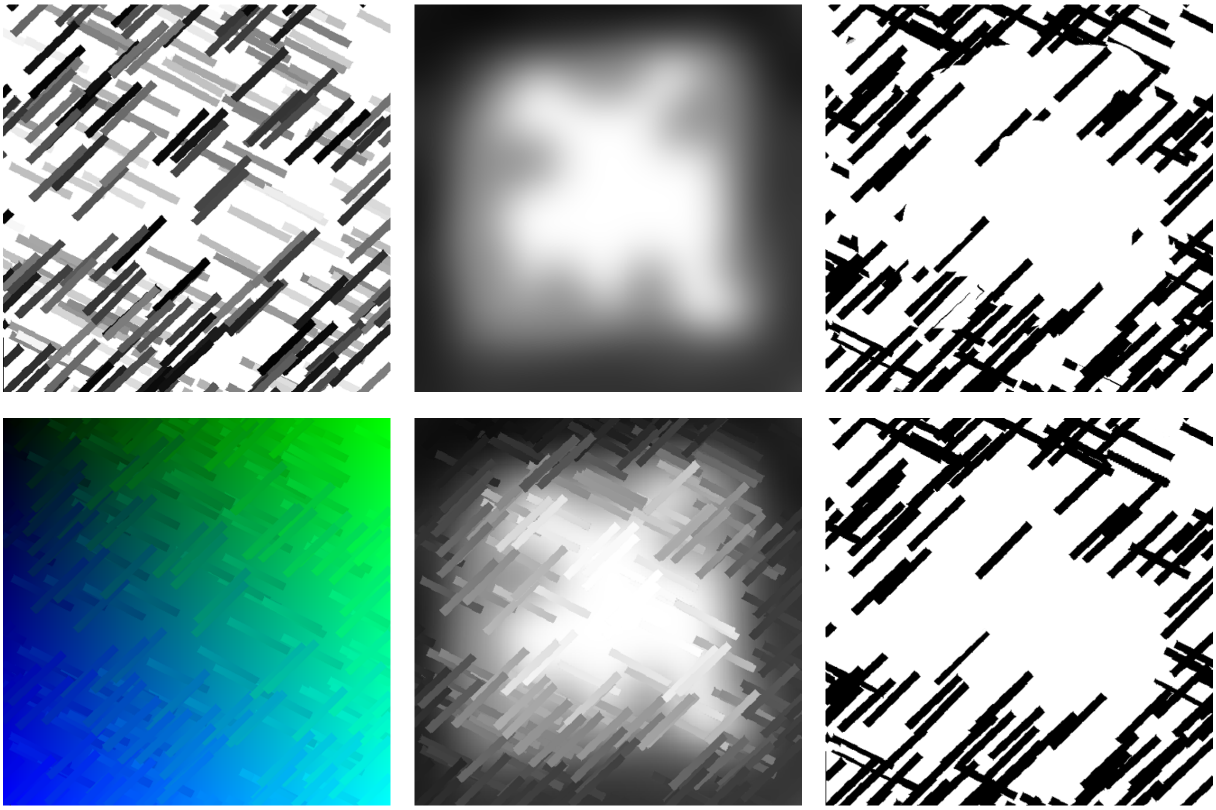


Figure 6.22: Upper row from left to right: halftone map, light map, result showing partial strokes. Lower row: warp map, adjusted light map, resulting whole strokes.

6.6.1 Lightmap Creation

For this technique to work, the lighting needs to be put into a texture. This texture must use the same parameterization as is used for the halftone map, although it may have a different resolution.

For static lighting this is no problem, as the lightmap can be computed in a preprocessing step. This is a standard procedure in many computer games. However, for dynamic lighting the lightmap has to be updated every frame. We use a software renderer for this, which is built into the Shark3D engine.

It would be nicer to leverage the rendering hardware for this task. However, we don't know of any hardware-accelerated approach that can map lighting information as seen from a single viewpoint onto a given scene with different parameterization. Possibly one could use dependent texturing to re-route shading, or "unwrap" the surfaces. this remains an area for future investigation.

Another possibility would be to use per-pixel lighting. With this, all the variables of the lighting model (typically the normal, viewing, and light direction) would need to be looked up from the reference location, and evaluated at the current pixel.

6.6.2 Tiling

Without lightmap warping, the halftone map can be replicated over a surface. This is not possible with the warp map, however. The warp map contains absolute coordinates for the lookup, so if the warp map was replicated, all tiles would refer to the same location on the surface, which is obviously wrong.

We therefore cannot allow tiling of the warp map texture. Instead, the geometry is split into tiles, with each tile having its own lightmap.

What would be needed to handle tiling without splitting geometry is a dependent lookup with relative coordinates, that is, relative to the tiled warpmap coordinates. This is beyond the capabilities of second-generation hardware. It could possibly be implemented with fragment shaders (third-generation hardware).

6.7 Procedural Texturing

The methods described in the sections above use a bitmap texture as primary source of variation on the surface of an object. The textures contain information about where strokes or other image elements should emerge in the rendering process. They are created externally, either procedurally in a preprocessing step or drawn by hand. A surface parameterization associates the texture with specific locations on the surface.

An alternative to bitmap textures for creating attributes that vary on a surface are procedural shaders. As with bitmap textures, a surface parameterization is employed. Instead of looking up the texture's value in the bitmap using the parameters for each point on the surface, a procedurally defined function is evaluated that depends on these parameters. This technique has extensively been in use in off-line rendering, most notably in RenderMan [Upstill, 1990; Apodaca and Gritz, 1999].

One of the main advantages of procedural shading over bitmap textures is their resolution independence. A bitmap texture, however large, only contains a fixed amount of detail. When zooming in on a bitmap-textured surface, there will be a point where no more detail is shown. In contrast, a procedural description can provide virtually unlimited detail, as, for example, fractals prove.

The current draft of the OpenGL 2.0 specification contains a shading language [Kessenich et al., 2003]. It allows both vertex shaders and fragment shaders to be specified in high-level language similar to C.

6.7.1 Vertex and Fragment Shaders

Vertex shaders operate on the vertices of the geometry. They replace the traditional fixed-function graphics pipeline stages of transform and lighting with a user-defined program.

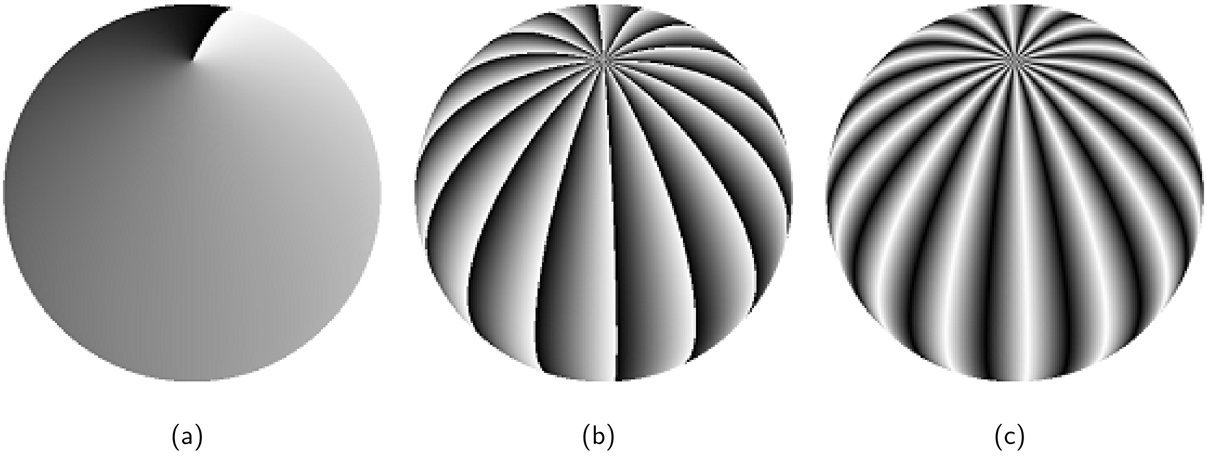

(a) (b) (c)

Figure 6.23: The base for procedural hatching is a parameter varying uniformly across the surface (a). Using a modulus function on this parameter creates a sawtooth wave (b), which is transformed into a symmetrical triangular wave (c).

The input consists of parameters and per-vertex attributes, the output is a vertex with attributes that are then interpolated across the primitive.

A fragment shader is evaluated for each fragment. It substitutes a program for the hard-wired texturing and color calculations. Interpolated attributes and shader parameters are used with arithmetic operations or texture lookups to determine the fragment color, which is subsequently blended into the frame buffer.

Procedural surface shaders [Olano et al., 2002, p. 7] can be used to implement hatching. This was demonstrated for off-line rendering with RenderMan shaders by [Johnston, 1998]. In this section we will develop a real-time implementation of parallel hatching based on Johnston's ideas using programmable shaders with a prototypical implementation of the OpenGL 2.0 shading language on a 3Dlabs WildCat VP board. It was demonstrated at the SIGGRAPH 2002 Exhibition in San Antonio.

The base for a procedural shader is a parameterization, which is usually given as individual values at each vertex of a surface and interpolated across a primitive. Alternatively, a parameterization can be derived from the vertex position, which can be used for solid texturing (using model coordinates) or for screen-space texturing (using screen coordinates).

The model used as example for Figures 6.23(a) to 6.28(a) is a fairly high tessellated sphere with latitude-longitude mapping texture coordinates. It was generated with the standard OpenGL utility quadric functions:

```
/* main program snippet */
GLUquadric *q = gluNewQuadric();
gluQuadricNormals(q, GLU_SMOOTH);
gluQuadricTexture(q, GL_TRUE);
gluSphere(q, 0.6f, 128, 128);
```

The sphere was chosen for simplicity. It demonstrates well the problem of different parameterization scale, as the longitude is reduced towards the poles. For another model rendered with the exact same technique refer to Figure 6.29. The high tessellation is necessary to better approximate surface derivatives. Due to linear interpolation of vertex attributes, the derivatives are constant for a primitive in current graphics accelerators.

In this example, we will use the s texture coordinate (longitude) as primary parameter. It runs from 0.0 to 1.0 around the sphere. A vertex shader puts this texture coordinate into the scalar interpolator "v", and transforms the vertex position into screen space:

```
// vertex shader
varying float v;
void main(void)
{
  v = gl_MultiTexCoord0.s;
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

All variable names prefixed with "gl_" are predefined mappings that correspond to the traditional OpenGL conventions. In our example, "gl_Vertex" and "gl_MultiTexCoord0" are vertex attributes, "gl_ModelViewProjectionMatrix" maps to current OpenGL state, and the "gl_Position" output variable refers to the homogeneous vertex position.

The fragment shader operates on the interpolated **varying** parameters written by the vertex shader. In our example, "v" is the sole connection between both shaders. For demonstration purposes, this value is mapped to a gray value, where 0.0 is displayed as black and 1.0 is shown as white (see Figure 6.23(a)). The complete fragment shader achieving this mapping is shown here:

```
// fragment shader
varying float v;
void main (void)
{
  gl_FragColor = vec4(v, v, v, 1.0);
}
```

Once vertex shader and fragment shader are "bound", they form a shading program that is applied to the geometry drawn thereafter. Note that we omit the complete fragment shader definition and mapping of variables to the fragment color in the following examples.

6.7.2 Generating Hatching Strokes

The first step in creating hatching lines is to generate stripes on the surface. While a sine function could be used to make a periodic variation on the surface, it is rather expensive. Another periodic function is the modulus operation, which is much cheaper to calculate. For a modulus of 1.0 it is equivalent to taking the fractional part of a number. In our

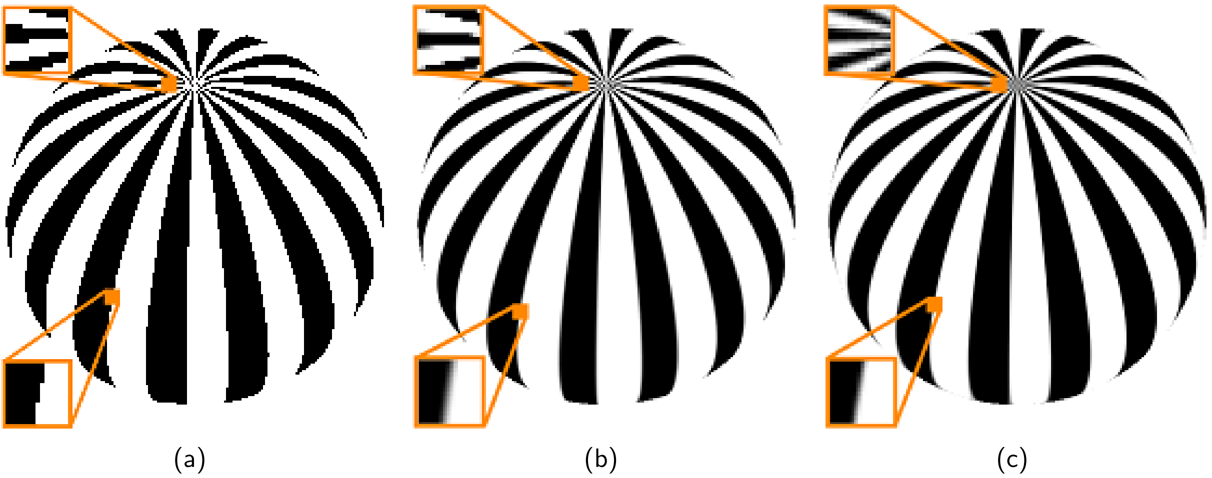

(a) (b) (c)

Figure 6.24: Applying a threshold to the triangular wave creates black strokes, which suffer from aliasing artifacts (a). Smoothing the strokes with a fixed-width smoothstep function introduces bluring in some regions, while not effectively removing aliasing in other areas (b). If the width of the step function is adjusted adaptively with respect to the degree of change of the underlying function, the edges of stripes are smoothly anti-aliased everywhere (c).

example, we construct a sawtooth function by taking the fractional part of a multiple of the parameterization (see Figure 6.23(b)). The frequency of the resulting wave can be adjusted by manipulating the scale, which is fixed at 16.0 here:

```
float sawtooth = fract(v * 16.0);
```

The sawtooth function, in contrast to a sine wave, is not symmetrical. But we can construct a triangular wave from the sawtooth wave by biasing it towards negative values and taking the absolute value (see Figure 6.23(c)):

```
float triangle = abs(2.0 * sawtooth - 1.0);
```

To create black-and-white hatching strokes, a step function is applied to the triangle wave. Varying the threshold value (here 0.5) will create wider or thinner strokes:

```
float square = step(0.5, triangle);
```

A problem with this approach is aliasing. Since the step function creates only black and white pixels, no intermediate gray values are available to smooth the result, as can be observed in Figure 6.24(a).

6.7.3 Anti-Aliasing

There are multiple ways to anti-alias a procedural shader [Olano et al., 2002, p. 100]. One simple solution would be super sampling, that is, the shader is evaluated multiple times for several positions inside each pixel. Besides being very expensive computation-wise, this only shifts aliasing artifacts to higher frequencies. A better method is analytical filtering. Here, the step function is convolved with a filter kernel. The built-in `smoothstep()` function provides such a smooth transition by allowing a minimal and a maximal threshold value, in-between of which the result will vary smoothly. For example, we can use a smooth filter which is 0.2 wide:

```
float square = smoothstep(0.4, 0.6, triangle);
```

As can be seen in Figure 6.24(b), this indeed creates smooth edges for the stripes. However, due to the fixed width of the smooth step function in the parameter domain, its width is not constant in screen space. This results in the edges being smeared over several pixels at the sphere's equator, while aliasing still occurs near the pole. The same effect would occur when moving the sphere in space or scaling it. When it is farther away, the projected filter width is much smaller than a pixel, leading to aliasing again. When zooming in on the object, the stripes get blurred because the filter now spans multiple pixels.

Because samples are taken for each pixel, the filter's support should be approximately one pixel wide in screen coordinates. It is necessary to "back-project" the screen-space values of the function to be filtered into the original function domain. For this purpose, the pixel shaders can provide the screen-space derivative of a function with respect to the screen x and y coordinates, that is, the approximate difference in function value from one pixel to the next.

The derivative of the parameter `v` with respect to x is given by the function `dPdx(v)`, in the y direction it is `dPdy(v)`. The derivatives of the sphere are visualized in Figure 6.25(a). A color mapping was chosen that maps `dPdx(v)` to the red color component and `dPdy(v)` to green. Because the absolute values of the derivatives are very small, they were scaled by a constant to fill the visible color range. To account for negative slopes, an offset of 0.5 was added to the scaled derivative. As can be seen in the image, the derivatives are not arithmetically exact (they would have to be at least symmetrical on a sphere), but are rather rough approximations. This is of no consequence in our context, but it means that the actual results might differ between different hardware implementations.

In order to adjust the filter width of the `smoothstep()` function, we calculate the magnitude `dp` of the gradient of `v` using the euclidean metric (visualized in Figure 6.25(b)):

```
float dp = length(vec2(dPdx(v), dPdy(v)));
```

When this value is used to adjust the thresholds for the `smoothstep()` function, the anti-aliasing quality is much more homogeneous across the sphere's surface (see Figure 6.24(c)). Only at the poles aliasing still happens. This is because in these regions there are too many strokes per pixel, that is, the feature (stroke) frequency is higher than the sampling (pixel) frequency. A usual band-limiting anti-aliasing approach would have to remove the high-frequency features altogether, resulting in a flat gray area. However, we want to still be able to discern individual hatching strokes even in dense regions, so we will instead adjust the number of strokes.

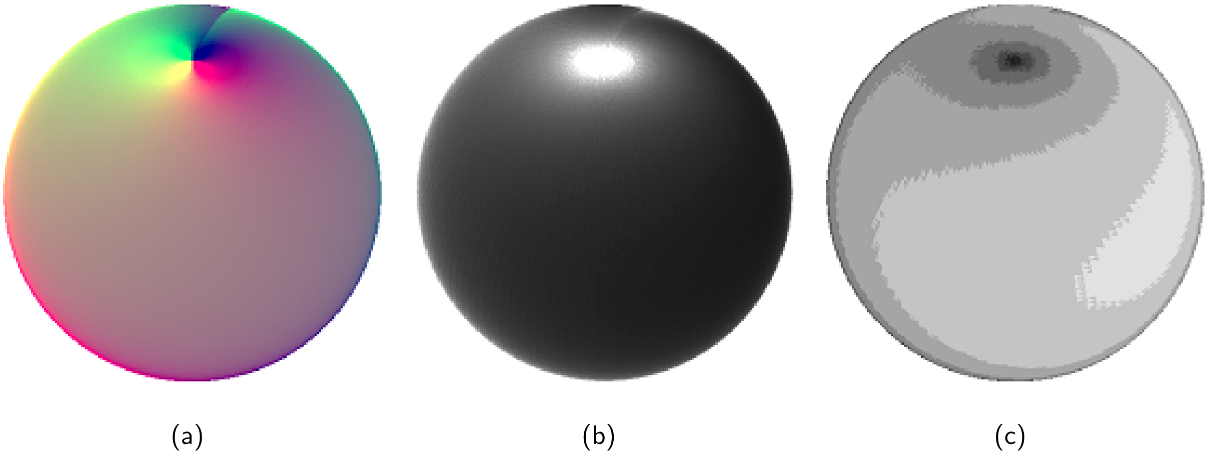

(a) (b) (c)

Figure 6.25: To determine the degree of change, the screen-relative derivatives dx and dy of a variable are provided by the hardware (a, color-coded). The length of this gradient vector (b) is used to adjust the width of the smooth step function. The integral logarithm of this length (c) can be used to adjust line spacing (see next figure).

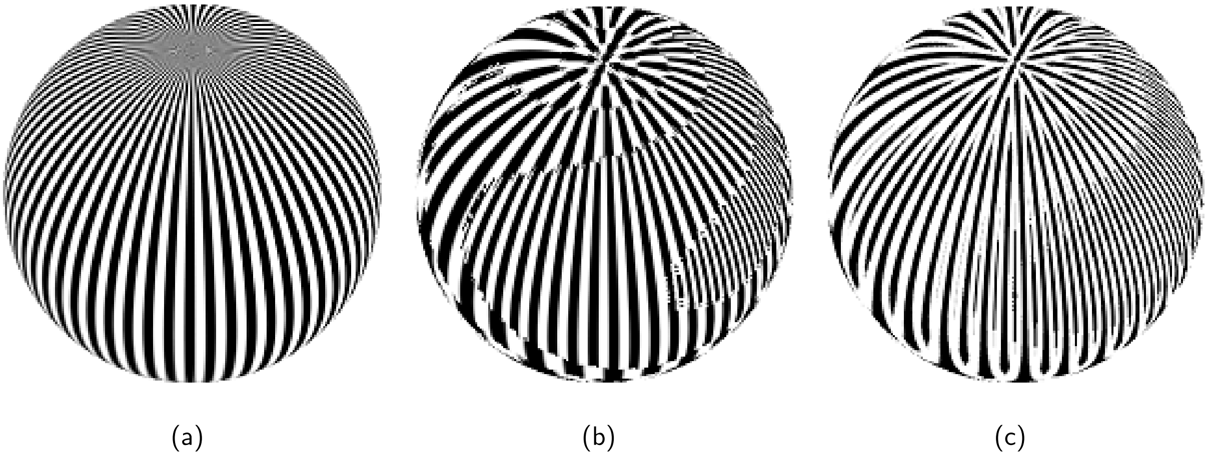

(a) (b) (c)

Figure 6.26: To create a constant line width for hatching, the frequency of lines needs to be adjusted. This is achieved by using the integral logarithm of the gradient (see previous figure) instead of a constant frequency (a), creating lines of roughly equal width in screen space (b). By interpolating between two integral frequencies the line endings can be tapered (c).

6.7.4 Stroke Density Adjustment

In order to adjust the number of strokes on a surface, we can modify the frequency of the sawtooth wave by using a higher or lower multiplier. In Figure 6.26(a), the frequency has been raised compared to the examples before. While it looks reasonable in the lower part where the lines are nicely spaced, again the lines get too close to each other near the rim and towards the pole.

Fortunately, we do have a measurement on how dense the lines actually get on the screen: the derivatives again. What is needed is to adjust the stripes' frequency based on the derivative. To remove every second line, the frequency has to be halved, to create a line between two others, the frequency must be doubled. So the derivative needs to be discretized into logarithmic chunks. To get a higher frequency when the derivative is smaller, the logarithm is negated:

```
float logdp = −log2(dp);
float ilogdp = floor(logdp);
float frequency = exp2(ilogdp);
```

The frequency is visualized in Figure 6.25(c). Again, the asymmetry stems from numeric inaccuracies in the calculation of derivatives but is more pronounced due to the logarithmic scale. Using this frequency in the calculation of the sawtooth wave gives an image that indeed exhibits strokes of roughly constant density everywhere on the surface (Figure 6.26(b)):

```
float sawtooth = fract(v ∗ 16.0 ∗ frequency);
```

It is interesting to note that because now the screen-space frequency of the shown pattern is constant, no adjustment of the step function's filter width is needed anymore. Fixed thresholds for the `smoothstep()` function are sufficient when the frequency adjustment is used.

The resulting rendering exhibits artifacts similar to the sharp borders between mipmap levels shown Figure 6.4. The transition from one stroke frequency to the next is too abrupt. We can adopt a similar solution to this problem. Instead of linear filtering between adjacent mipmap levels, we linearly interpolate between two frequencies of the triangle wave. The fractional part of `logdp` serves as weight, it is 0.0 at one frequency and increases smoothly towards 1.0, when the doubled frequency will be chosen. A triangular wave t of double the frequency can be generated by $|2t-1|$. To interpolate between t and $|2t-1|$ based on the transition weight w, we use the relation $(1-w)t+w|2t-1| = |(1+w)t-w|$:

```
float transition = logdp − ilogdp;
triangle = abs((1.0 + transition) ∗ triangle − transition);
```

As shown in Figure 6.26(c), the resulting stroke quality is surprisingly good. Especially at the poles, lines are smoothly faded in with tapered ends. The blurriness in the sphere's lower part can be attributed to the insufficient accuracy of the derivatives. Except for using more accurate hardware there is not much that can be done to counteract this.

6.7.5 Lighting

A constant density of equally wide lines creates an overall constant perceived intensity. By varying the density or the line width, the perceived intensity can be adjusted to reflect lighting. In Figure 6.27(a) a very simple diffuse lighting model has been added to the vertex shader:

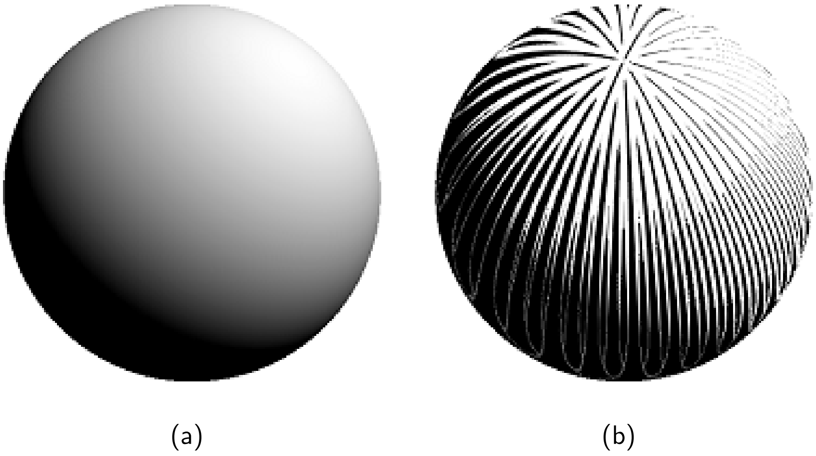


Figure 6.27: To make the illustration reflect lighting (a), the intensity is used in the shader to modify the line width by offsetting the threshold (b).

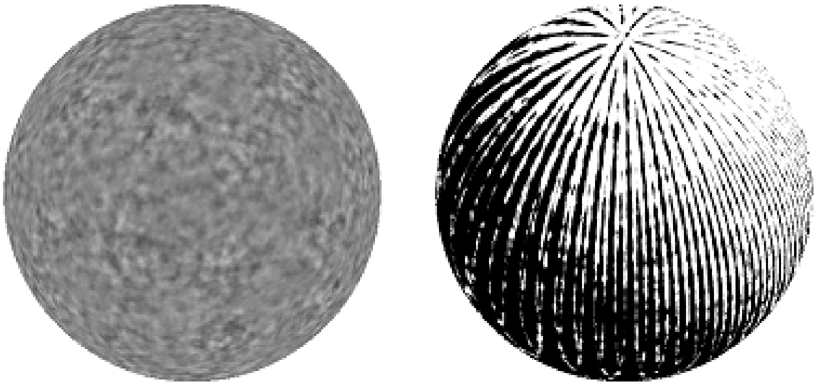
```
varying float v;
varying float lightIntensity;
void main(void)
{
  vec3 P = vec3(gl_ModelViewMatrix * gl_Vertex);
  vec3 N = normalize(gl_NormalMatrix * gl_Normal);
  vec3 L = normalize(gl_LightSource[0].position - P);
  lightIntensity = max(dot(L, N), 0.0);
  v = gl_MultiTexCoord0.s;
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

The `lightIntensity` value is now used in the fragment shader to bias the step function's threshold values. In lighter regions, the threshold is lowered so the black hatching strokes get thinner. In unlit regions, by the same principle, more pixels become dark because they fall under the higher threshold, producing wider lines. The result is shown in Figure 6.27(b):

```
float edge0 = clamp(lightIntensity - 0.2, 0.0, 1.0);
float edge1 = clamp(lightIntensity, 0.0, 1.0);
float square = 1.0 - smoothstep(edge0, edge1, triangle);
```

6.7.6 Noise

To make the hatching look less perfect and technical, the lines can be made less straight or of unsteady thickness. This can be accomplished by using a noise function indexed by the three-dimensional fragment position. Such a noise function, as provided by the fragment shader, is shown in Figure 6.28(a).



(a) (b)

Figure 6.28: Threedimensional Perlin noise can be used to create a less technical look while still maintaining frame coherency (a). It is used here to bias the threshold function, which creates an uneven line width (b).

Various parameters used in the hatching shader can be made noisy to achieve different looks. For Figure 6.28(b), the lightIntensity value has been biased by noise. This influences the line width, making it uneven. Another possible variation is applied in Figure 6.29. Here, lines are made to wobble by adding low-frequent noise to the v parameter itself, shifting the peak of the triangle function from side to side.

6.7.7 Discussion

In comparison to the mipmapping approach described in Section 6.1 the procedural texture provides virtually unlimited detail (up to its precision limits). Even when zooming in on a surface, there will be constant-density hatching lines, in contrast to mipmapping which only works as long as the texture is minified. However, the expressiveness of hand-drawn textures (Section 6.3) can hardly be achieved by a procedural definition. Also, the fragment shader presented here is rather complex, the bitmap-textured variants have a much higher performance.

6.8 Stroke animation

While much care has been taken to make strokes appear steady and frame-coherent, an interactive application might benefit from animating strokes. Deliberately breaking the frame coherence can add liveliness or artistic expression to the animation.

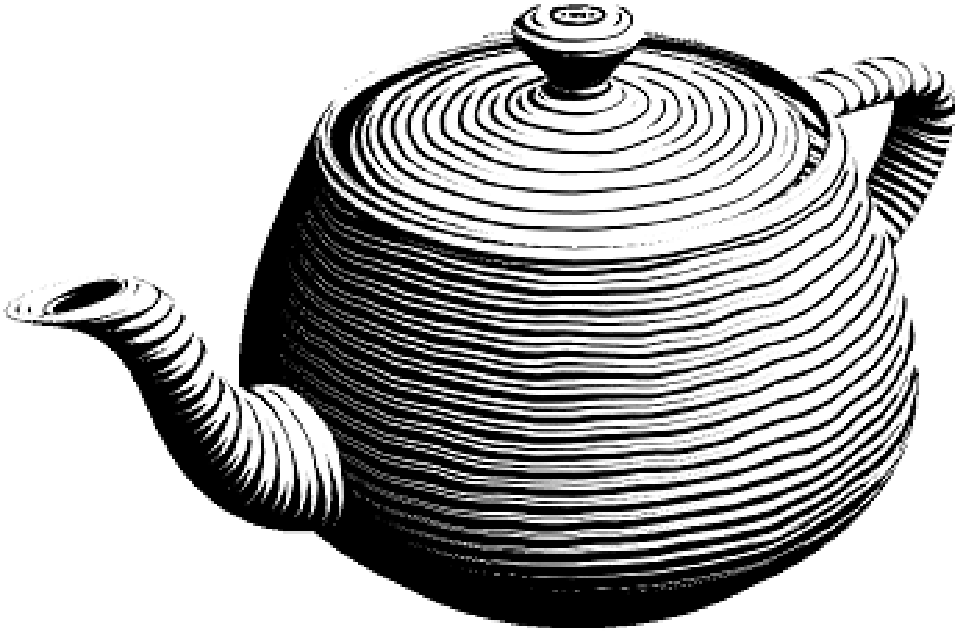


Figure 6.29: For this illustration, low-frequency Perlin noise was used to shift the hatching parameter which creates "wiggly" lines.

6.8.1 First Generation Hardware

The first generation of GPUs is rather limited in its pixel processing capabilities. One method is to use an animated halftone texture. A series of halftone screens is created and uploaded in every frame. If memory permits, all frames of the animated screen can be uploaded and bound in turn. If the hardware supports 3D textures, the single bitmaps can be uploaded at once and selected through the third texture coordinate, which allows smooth interpolation between frames. This method is quite flexible because the artist is free to choose how the animated strokes look like. If the halftone screens are coherent, so will be the rendering. However, a texture upload in every frame is rather expensive in terms of bandwidth, whereas storing all frames on the GPU quickly exhausts the graphics memory.

Another, not as flexible approach is to use a texture containing tileable noise and us it to modulate the lighting contribution. This will give the strokes an uneven thickness. When this noise texture is shifted relative to the objects surface, the strokes will change their thickness over time. However, there will be a noticeable shower-door effect because the movement of the modulating texture is uniform all over the surface.

6.8.2 Second Generation Hardware

Graphics processing units of the second generation provide allow for dependent texture reads. This permits to distort one texture through an other texture. If the one texture is animated, the dependent texture will be animated, too.

This can be used to make strokes wobble in time. A texture is used that contains low-frequency tileable noise offset vectors. When accessing the halftone screen through this dependent texture it gets locally distorted. Animating the noise texture (by moving it relative to the surface or using multi-frame sequences) results in animated, wobbling strokes.

6.8.3 Third Generation Hardware

Since the third generation hardware supports noise functions in the pixel shader natively, adding animation is rather easy. If for normal rendering n-dimensional noise is used, it can be replaced by $n+1$-dimensional noise with the last parameter indexed by time.

Also, the rate of change can be controlled easily. When low-frequent time-dependent noise is used, the strokes will vary smoothly. Raising the frequency will result in seemingly random changes and break frame coherency.

6.9 Discussion

The technique of using mipmaps for portraying surface features of constant screen-size was independently developed and first published by [Klein et al., 2000]. It differs from our approach in that strokes are drawn into each mipmap level individually so there is no coherence between mipmap levels. Also, the lighting is static, that is, the textures already contain the lighting contribution and do not adapt to varying lighting conditions.

More similar to our approach is the real-time hatching work by [Praun et al., 2001]. The main difference is that multiple textures are used for different intensities, and that the blending computations are performed at the vertex level. Thus, the shading is equivalent to interpolated Gouraud shading and cannot be combined with per-pixel modulations like our detail textures. These deficiencies were overcome in [Webb et al., 2002] which introduced a per-pixel multi-level threshold scheme. It achieves a very good stroke quality, but is much more demanding than our halftoning scheme.

The main differences of our real-time halftoning technique to classic halftoning is in how the halftone screen is constructed, and in the processing order of pixels. In classic halftoning, the halftone screen is fixed and used throughout the image. The image is processed in a linear order, for example, row by row. In contrast, in our approach the halftone screen is attached to the 3D objects, and projected to the screen in each frame. When the scene is animated, the halftone screen will be different for each frame. The processing order of pixels depends on the rasterization of the 3D objects.

When objects overlap on the screen, pixels will be overwritten. This is the reason why both black and white pixels are generated, to facilitate the normal hidden-surface removal process. If only black pixels would be generated, hidden objects would “shine through” the front-most object’s surface. One could of course render the whole halftone screen to the frame buffer, render the intensity to another buffer, and do the threshold operation as a post-processing step. This technique, also known as *deferred shading* [Olano et al., 2002, p. 12], trades higher memory and bandwidth consumption for fewer arithmetic operations. It would be advantageous if the shading operations are very complex and the scene has a high depth complexity. However, with the exception of the procedural shader presented in Section 6.7, the shaders introduced in our work are rather lightweight, so they would not benefit from deferred shading.

In comparison to the explicit stroke rendering scheme, the bitmap-based scheme is faster and relatively easy to integrate in conventional rendering pipelines, because it mainly involves setting up the texturing stages. Any game engine that supports texturing and configuring the texture combining stages could use this method.

However, bitmaps provides less control about the stroke appearance. Also, anti-aliasing is an issue. Multi-sample anti-aliasing , which is the anti-aliasing method supported by most graphics accelerators today, samples textures or calculates pixel shaders only once per pixel, in contrast to super-sampling which samples the texture for each sub-pixel. Thus, if the texture or the shader result is not anti-aliased, multi-sampling will not help at all. In contrast, the geometry-based strokes can be anti-aliased very well, either using polygon smoothing or full-scene anti-aliasing.

7 Explicit Outline Rendering

In many traditional techniques, outlines play an important role for the image. In sketches and stylized drawings they sometimes are the only image element. Cartoons often employ only flat colors and outlines. In other styles, like pen-and-ink drawings or engravings, outlines are used in combination with strokes which indicate shading or detail. in contrast, outlines are rarely used in realistic styles. Here, other techniques for accentuating parts of an image are more common [Hoppe et al., 1996].

In combination with halftone rendering techniques, outlines serve the purpose of separating objects from each other, as well as to indicate features on the surface of objects. Also, the transition from outlines to shading strokes is somewhat fuzzy. For example, the lines used to draw bricks in a wall could be used to indicate shading by detail, or they could be used to outline each brick.

In the context of this work, we only call geometrically defined lines "outlines." For example, when a brick wall was modeled as a single box, it would be drawn with an outline surrounding it, while each brick was drawn as texture. On the other hand, if each brick was modeled separately,

While outline rendering techniques are not the main topic of this thesis, many rendering styles that can be implemented by halftoning techniques benefit from added outlines. Some of the developed techniques for accelerated outline rendering are discussed next.

First, a simple technique of edge classification is shown (Section 7.1). Then a technique is discussed which specifically makes used of hardware-accelerated vertex programs (Section 7.2). The last section proposes an acceleration technique based on subdivision (Section 7.3).

All these techniques have in common that they operate on the geometric data of a model. The following chapter will discuss image-based outline rendering techniques.

7.1 Pre-Classification to Reduce Silhouette Tests

Silhouettes are view-dependent and therefore silhouette edges need to be determined in every frame. Supposed that for an intended illustration style we do not need to distinguish between discontinuities and silhouettes, we can safely apply an optimization: We only need to consider strictly convex, smooth edges. Non-smooth edges are drawn anyway, and smooth edges in concavities cannot be silhouettes if the volume is solid. This can significantly reduce the number of edge candidates.

For example, in the eight-sided prism used to approximate a cylinder shown in Figure 7.1, there are 42 edges, but only 8 smooth edges are silhouette candidates when applying the "smooth-and-convex" rule. If an object only contains sharp and non-convex edges, no silhouette detection needs to be performed at run-time at all.

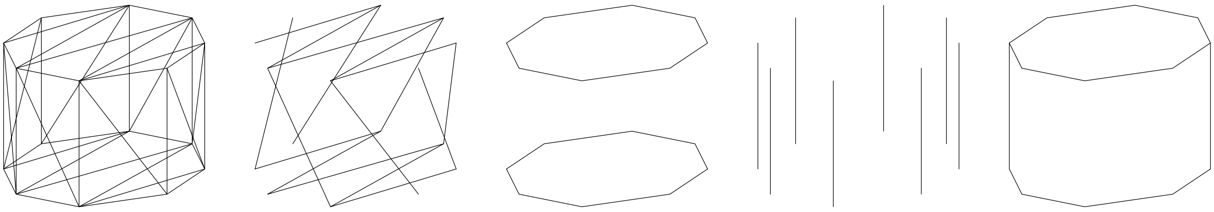


Figure 7.1: Silhouette determination: all edges (left), smooth non-convex edges are never drawn (second), sharp edges are always drawn (third), smooth convex edges are silhouette candidates (fourth), result with hidden lines removed (right).

7.1.1 Determining Convexity

Convexity of edges is determined in a preprocessing step. For each smooth edge, the unshared vertex of one adjacent triangle is tested whether it lies in front of the plane of the other triangle. In Figure 7.2 this is demonstrated with the shared edge $\overline{AC}$ of the triangles ABC and ACD. To determine if D is in front of ABC's plane we can take the dot product of ABC's normal N_{ABC} and the vector $\overrightarrow{AD}$:

$$\overline{AC} \text{ is strictly convex} \iff (D - A) \cdot N_{ABC} < 0$$

In practice, we compare to a small ϵ instead of 0 to ensure that co-planar triangles do not become silhouette candidates.

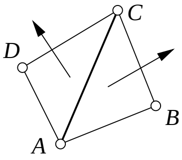


Figure 7.2: Determining convexity of an edge

7.1.2 Classification

The output of the classification algorithm is two lists of edges: One contains the edges that always have to be drawn, the other is the list of silhouette candidates. Initially, both lists are empty. Then, each edge of the model is tested in turn:

- If the edge is marked as sharp, add it to the always-draw list.

- Otherwise, if the edge is strictly convex, add it to the silhouette candidates list.
- Otherwise, discard the edge.

Returning to the example of Figure 7.1, we would get an always-draw list of 16 edges, and a list of silhouette candidates containing 8 edges. This means 18 edges are removed completely, which is 42% of the original 42 edges. If an object has more smooth edges, the ratio will be lower. On the other hand, if an object contains only flat faces, like a cube or other technical subjects, it is possible that only edges remain that need to be drawn always. That would mean no additional silhouette determination would have to be performed at run-time.

An additional optimization that can be performed in the preprocessing step is to search connected chains of vertices in the always-draw list. The advantage is that at run-time, these chains can be drawn more efficiently using polyline primitives. Drawing individual line segments uses two vertices per line. In a polyline, only the first segment is specified with two vertices, any subsequent edge in the chain is created by only specifying the end vertex. Since the resulting list is static, it can be also stored as a display list.

7.1.3 Silhouette Testing at Run-Time

At run-time, the list of always-draw edges can be displayed as-is. If it was stored as display list, the display list is called. If chains were collected, they are rendered using the polyline primitive, otherwise each edge in the list is drawn as individual line segment.

For the edges in the silhouette candidate list, it has to be determined whether for the current viewer position an edge is a silhouette. This is checked in the traditional manner of determining the facingness of both adjacent polygons. If a silhouette candidate edge is adjacent to both a front-facing and a back-facing polygon, it is a silhouette and thus gets drawn. Otherwise, it is discarded for this frame.

In comparison to the brute-force approach that just checks all edges in the model, this can be a significant reduction. Also, in our approach, checking if a polygon is facing the viewer is only necessary for the wings of silhouette candidates (which amounts to 16 out of 28 triangles in the example of Figure 7.1).

7.2 Silhouette Rendering With Vertex Programs

The one CPU-intensive run-time calculation remaining in the approach described in the last section is the determination of whether a silhouette candidate is indeed a silhouette edge for a given viewer position. Additionally, each of the silhouette edges has to be transferred to the graphics board in every frame. In this section, we describe an approach to offload the silhouette determination entirely to the GPU using vertex programs. Since all silhouette candidate edges can be stored in video memory, there is also a bandwidth saving. This method could as well be used without the pre-classification algorithm to determine only the silhouettes of a model in a brute-force way.

The basic idea is to test the facingness of an edge's two adjacent polygons, and then draw the edge only if the facingness differs. However, there are several problems here. Firstly, there is no data structure on the GPU where one could access the polygons adjacent to an edge. In fact, not even the edge itself is accessible as entity with current graphics hardware. Secondly, there is no way to "discard" a primitive in a vertex program. Possibly, future hardware will allow this with a programmable primitive processor.

But, as we have seen in Chapter 5, there is a way to enable or disable individual lines using a vertex program. We will use the exact same approach here. The idea is to put any information for reaching the decision into each vertex of the edge, and performing the same computation. An edge can be discarded by various means, as discussed in Section 5.4.5 (p. 69).

7.2.1 Data Structures

Using Cg notation, the basic structure for a vertex in the GPU-based silhouette determination method is as follows::

```
struct SilhouetteVertexOrtho {
  float4 Position : POSITION;
  float4 NormalA : ATTR1;
  float4 NormalB : ATTR2;
};
```

This allows accessing the vertex position, and the normals of both adjacent polygons for determination of facingness. Facingness is again computed by scalar multiplication of the normal with the view direction. However, this only works correctly under orthogonal projection. If a perspective projection is used, the view direction is dependent on the vertex position, so the result of the silhouette test might differ for both ends of an edge.

This might or might not be a problem depending on which method is used for culling edges. To avoid the discrepancy, the same position needs to be used for determining the view vector in both vertices. This can be accomplished by adding another attribute to the above structure. We can use the midpoint of the edge as reference position in both vertices:

```
struct SilhouetteVertex {
  float4 Position : POSITION;
  float4 NormalA : ATTR1;
  float4 NormalB : ATTR2;
  float4 Midpoint : ATTR3;
};
```

In addition, if a quad-based scheme is used for rendering lines, a Direction vector needs to be added. For details, see the discussion of line vs. quad-based rendering in Section 5.4.1 (p. 64).

Such a vertex structure is uploaded for both vertices in every edge that is destined for silhouette testing. The Position attribute is different in each vertex, while NormalA, NormalB, and Midpoint are the same for both ends and are initialized with the adjacent faces' normals and the average of both vertex positions, respectively.

7.2.2 Silhouette Testing on the GPU

At run-time, a vertex program tests the facingness of each face using its normal stored in NormalA or NormalB. If the facingness is the same, the vertex is adjusted to make the edge be invisible. For the sake of simplicity, we assign an alpha value here, other schemes can be used as well, as discussed in Section 5.4.5.

```
void silhouette_test(SilhouetteVertex vertex,
                     out float4 position : HPOS,
                     out float4 color : COL0)
{
  position = mul(glstate.matrix.mvp, vertex.Position);

  float4 normalA = mul(glstate.matrix.invtrans.modelview[0], vertex.NormalA);
  float4 normalB = mul(glstate.matrix.invtrans.modelview[0], vertex.NormalB);
  float4 view = mul(glstate.matrix.modelview[0], vertex.Midpoint);

  float facingnessA = dot(view, normalA);
  float facingnessB = dot(view, normalB);

  color = facingnessA * facingnessB < 0.0
     ? float4(0.0, 0.0, 0.0, 1.0) : float4(0.0, 0.0, 0.0, 0.0);
}
```

This Cg procedure translates into a 23 instruction vertex program, which is relatively moderate. This allows this approach to be employed even on detailed models, especially if combined with the pre-classification of silhouette candidates. Of special importance is that all operations are carried out by the GPU alone, the CPU is free to do other calculations in the mean time.

7.3 Silhouette Detection Through Subdivision

Subdivision surfaces have several features that make them especially apt for rendering line drawings. For example, this representation explicitly contains smoothness information, that is, whether an edge in the model is a crease or not. Special rules are used for subdividing these edges. For another example, subdivision surfaces support multi-resolution methods by design. Operations can be carried out at coarse levels and propagated to finer levels of detail.

Amazingly, these features do not seem to have got exploited in non-photorealistic rendering to date. Either subdivision surfaces are treated as polygonal models after performing a few subdivision steps [Markosian et al., 1997; Kaplan et al., 2000], or they are only used as a generic smooth surface representation [Hertzmann and Zorin, 2000].

7.3.1 Modified Butterfly subdivision

In this work the Modified Butterfly subdivision scheme is used [Zorin et al., 1996]. It was chosen mainly because it is an interpolating scheme. Although approximating schemes in general achieve a better surface quality, this is irrelevant in the context of non-photorealistic rendering. Much more important is that the coarser subdivision levels—indeed, even the control mesh—roughly resemble the limit surface, which is approached after few subdivision steps (see Figure 7.3).

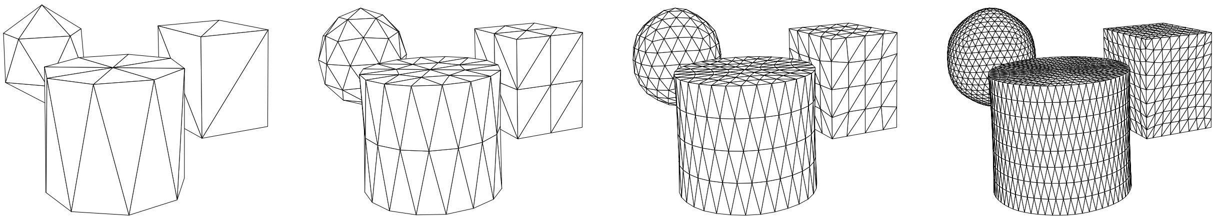


Figure 7.3: Subdivision of some simple objects. The initial mesh is shown on the left, then three steps of subdivision were performed.

The rule used for subdividing crease edges is different from the normal edge subdivision rule. The surface subdivision only uses vertices on one side of a crease, so sharp edges are preserved in each refinement step. That means that to make a line drawing, the sharp edges “only” need to be drawn along with the silhouettes (Figure 7.4).

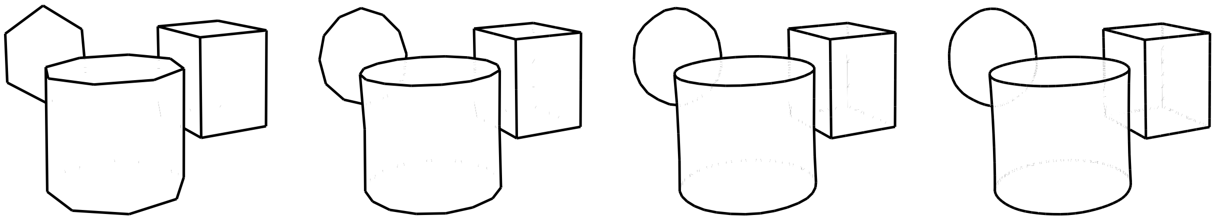


Figure 7.4: Silhouettes and creases after 0, 1, 2, and 3 subdivision steps.

7.3.2 Silhouette propagation

The approaches currently used in the field of non-photorealistic rendering operate on the fully refined mesh. But an interesting property of subdivision schemes is that a silhouette edge in subdivision level l "creates" a silhouette edge in level $l+1$. To determine silhouette edges in level $l + 1$ one can consider edges in those triangles that were adjacent to a silhouette in level l.

However, it is not guaranteed that this will find *all* silhouettes in level $l + 1$, as pointed out by [Markosian, 2002]. While silhouettes on the object's boundary are found, the subdivision process itself can introduce undulations that might lead to new, inner silhouette edges. For the sake of completeness, it would be desirable to identify situations where these new silhouettes occur, and deal with them explicitly. But since these "wrinkles" are usually not wanted, we will limit the discussion to the normal case.

The insight of silhouette coherence between subdivision levels leads to a new method for fast silhouette determination that is intertwined with the subdivision process. The idea is to find silhouette edges in one subdivision level, and *propagate* this information to the next level. That way, most edges in the finer subdivision levels do not have to be processed at all (see Figure 7.5). Additionally, as described before, only smooth silhouettes need to be tested, since sharp edges are drawn anyway.

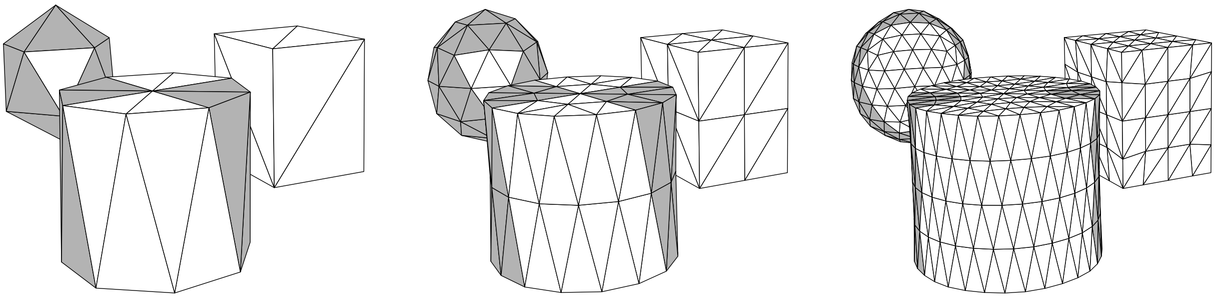


Figure 7.5: Silhouette propagation example: Triangles adjacent to the silhouette in one level (left) are subdivided and become the silhouette band for the next subdivision level (middle). The propagation process is repeated (right).

In the set-up phase, the silhouette edges in the controlling mesh are gathered. Here, any deterministic algorithm can be used. In the most straight-forward case, the orientation for all faces is determined and the edges sharing both a front-facing and a back-facing triangle are marked as silhouette edges, as illustrated in Figure 7.6 (left).

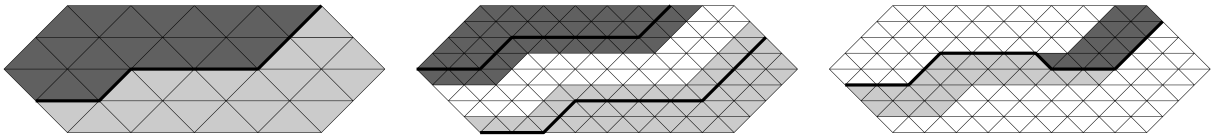


Figure 7.6: Silhouette propagation scheme. Before subdivision (left), silhouette band (center), refined silhouette (right). Different shades of grey depict front/back faces.

Then, in the subdivision step, each edge is subdivided, creating a new vertex for each edge. There is a narrow band of triangles along the previous-level silhouettes that contains the edges of the new silhouette. This band consists of the triangles that emerge from subdividing the triangles that shared at least one vertex with the silhouette (see Figure 7.6, middle). Triangles outside this band do not need to be considered, because these are known to have the same orientation like the triangles adjacent to them in the band.

To determine the silhouette for this subdivision level, the triangles in the band are categorized as front-facing or back-facing. Then all edges sharing a front facing and a back facing triangle are finally marked as silhouettes (Figure 7.6, right). The whole process is repeated until the desired level of subdivision is reached.

7.3.3 Fast Contour Drawing

While it is certainly possible to keep on subdividing until there are no visual artifacts anymore, this creates an excessive number of triangles. Usually, to overcome this problem, adaptive subdivision techniques are introduced. These adaptive schemes will generate more detail in visually important regions, for example, in areas of higher curvature.

Considering line drawings, "visually most important" are the creases and silhouettes. So, an adaptive scheme can be applied that subdivides areas containing creases or silhouettes with higher resolution than other regions of the mesh. However, why should we spend so much effort on subdividing *faces* when all we want to draw in the end are the *edges*? Instead, we can use *curve subdivision* to refine creases and silhouettes without the need to subdivide surfaces any more than necessary.

For creases, this approach is obviously justified. Crease edges are a special-case in the subdivision process anyway. They get subdivided using the 4-point curve-subdivision scheme [Dyn et al., 1987]. For silhouettes, however, we have to ensure the surface is subdivided with sufficient precision, so that the deviation of the actual silhouette from the curve subdivision silhouette is negligible.

This is illustrated in Figure 7.7. The silhouettes and creases are drawn with one level of subdivision. The actual mesh geometry is shown for comparison. On the left, no 3D subdivision was performed. Here, the rim of the cylinder deviates noticeably from the object's silhouette. After performing one (middle) or two (right) surface subdivision steps, however, the interpolated curve is sufficiently close to the actual silhouette.

The curve subdivision process is as follows: First, chains of crease edges are assembled. For this, an arbitrary crease edge is chosen, and edges sharing one vertex of the edge are connected, until a corner is found or the chain is closed. A vertex is a corner if at least three sharp edges meet [Hoppe et al., 1994]. Then the same chaining happens for smooth silhouette edges, stopping at corners, and, additionally, at crease vertices (vertices on a crease). Sharp silhouettes could be handled as well if a distinguished drawing style for silhouettes was desired.

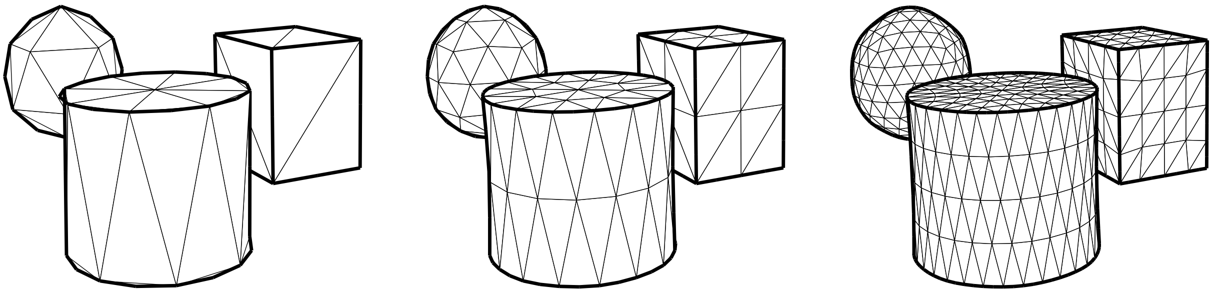


Figure 7.7: Drawing by subdivision: Curves were subdivided once.

Then the actual curve subdivision is performed on the chains. The 4-point subdivision scheme is used (see Figure 7.8). For closed chains, the regular stencil can be applied everywhere. For open chains, the first and last vertex are weighted with 1/2.

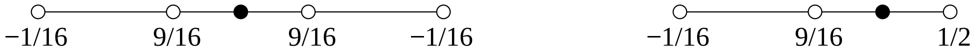


Figure 7.8: Stencil of 4-point curve subdivision scheme: Regular (left) and end of chain (right).

Note that we can drop the depth component of the curve points altogether and use 2D curve subdivision as a drawing primitive. The curve-subdivision only considers the vertex positions of the curve itself and thus is completely unrelated to the surface tessellation. So except for perspective foreshortening, the interpolation of projected 2D vertices produces the same drawings as the projection of the three-dimensionally subdivided curve.

One level of curve subdivision is of course not quite sufficient. But with three levels, the curves become smooth, as shown in Figure 7.9. In fact, they are even smoother now than in the last image of Figure 7.4, which actually has one more level of 3D surface subdivision. This is only noticeable in magnification, though.

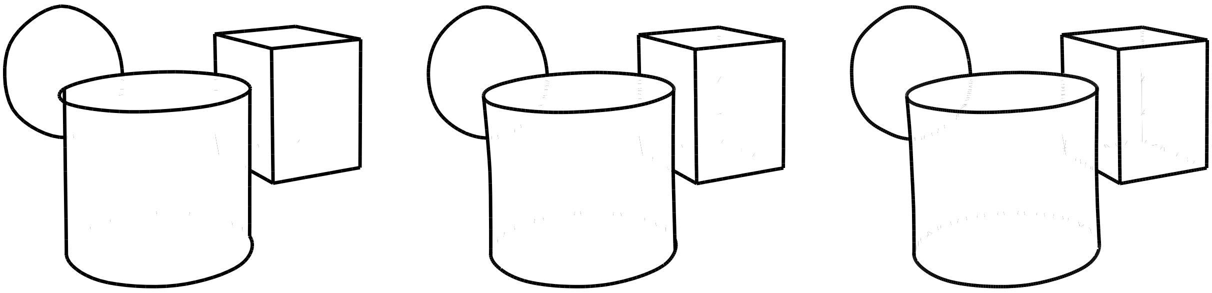


Figure 7.9: Drawing by subdivision: Curves were subdivided three times.

7.3.4 Results

The presented methods were experimentally implemented using the Squeak Smalltalk environment [Squeak, 2001]. Being an purely interpreted system, no absolute timings comparable with, for example, [Markosian et al., 1997] can be given. However, using curve subdivision refinement rather than a full surface subdivision step gave a relative speed-up by the factor 10.

A problem with using the Modified Butterfly subdivision scheme is that it is not widely supported yet in modeling programs. Our approach was to model the polygon mesh while imagining the resulting surface, and then to evaluate the result of subdivision in our test application. A model created this way is shown in Figures 7.10 and 7.11.

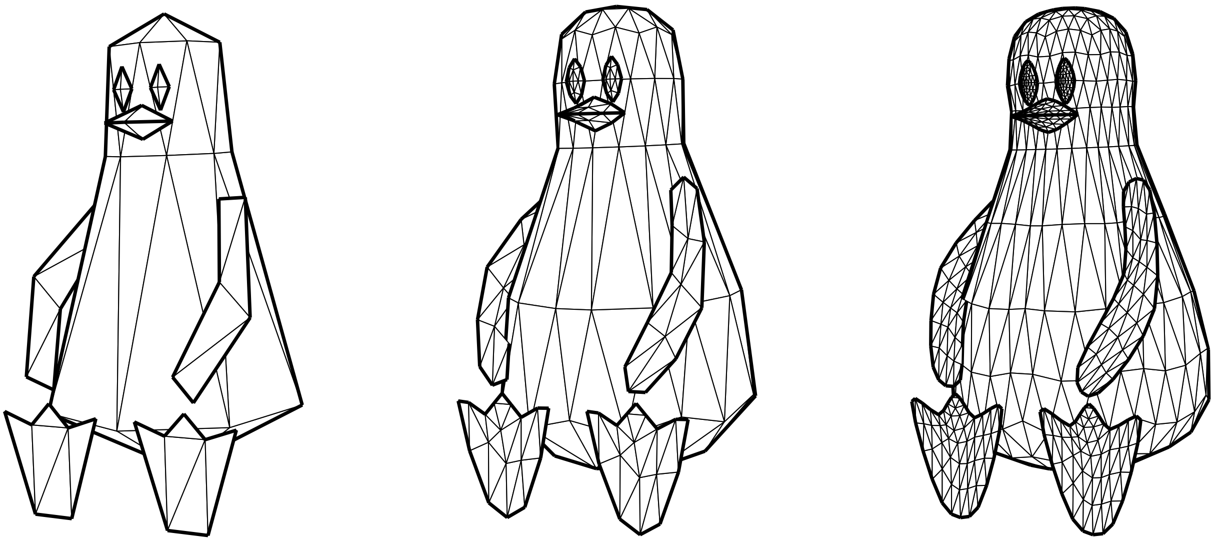


Figure 7.10: Sample object without curve subdivision in three refinement levels.

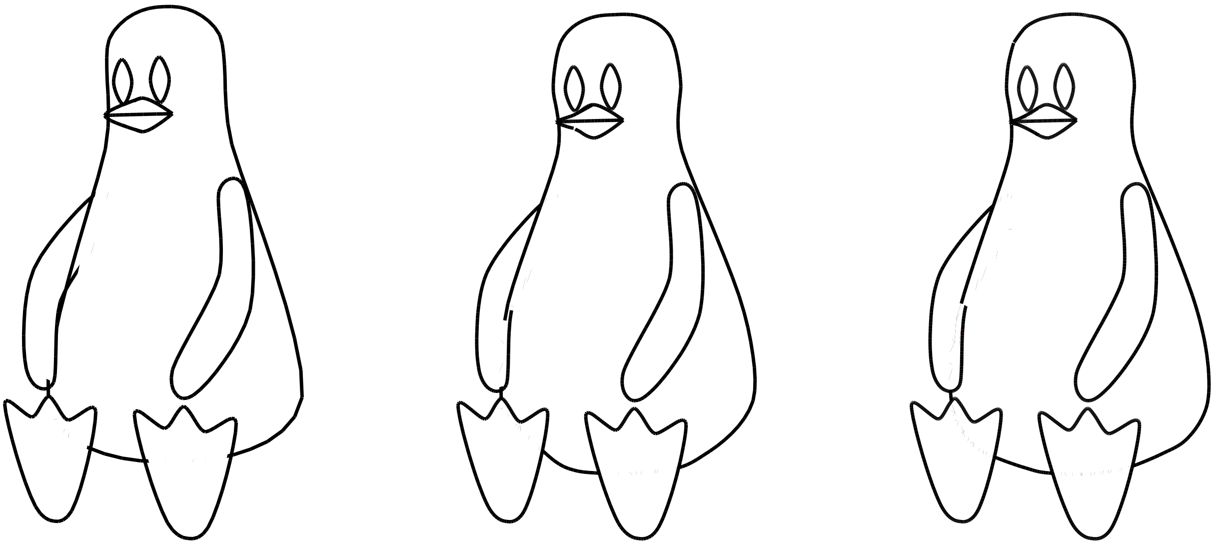


Figure 7.11: Sample object with curve subdivision, same three refinement levels.

8 Implicit Outline Rendering

In this chapter we will discuss texture-based methods for finding outlines. The basic principle for image-based contour drawing was introduced by Saito and Takahashi. It starts by rendering image buffers storing values derived from geometry, rather than colors (so-called G-buffers). Discontinuities in these buffers correspond to contours, which can be detected using first or second order differential filters [Saito and Takahashi, 1990].

A real-time implementation of these filters requires evaluating the filter kernel. This can be done with textures if there is powerful enough fragment programmability [James, 2001; Mitchell et al., 2002].

We will discuss the basic principle of filtering with textures (Section 8.1) and real-time rendering of G-buffers (Section 8.2) first, before introducing new methods for contour detection. An implementation of the Sobel operator is devised which works with only 4 bilinearly filtered texture samples, even though this operator originally requires 6 non-zero weighted samples (Section 8.3). Even more radically, a contour detection method using only a single texture unit is introduced, which exploits artifacts of bilinear texture filtering (Section 8.4). The chapter concludes with new methods for rendering contours contrasting with the background, rather than contours of a uniform color (Section 8.6).

8.1 Filtering With Textures

Filtering an image means to create a new image, where each pixel value $p'_{x,y}$ at Cartesian coordinates (x, y) is calculated as the weighted sum of the values in the $(2w + 1)$-neighborhood $(x \pm w, y \pm w)$ of the original pixel $p_{x,y}$. For example, for a 3×3 neighborhood, nine weights $(A \dots I)$ are used and nine pixels are sampled for each output pixel:

$$p'_{x,y} = \begin{pmatrix} A & p_{x-1,y-1} & + & B & p_{x,y-1} & + & C & p_{x+1,y-1} & + \\ D & p_{x-1,y} & + & E & p_{x,y} & + & F & p_{x+1,y} & + \\ G & p_{x-1,y+1} & + & H & p_{x,y+1} & + & I & p_{x+1,y+1} & \end{pmatrix}$$

The so-called "filter kernel" is the collection of weights and usually written as matrix:

A	B	C
D	E	F
G	H	I

In terms of a real-time implementation, the image is stored as a texture, and texture-combining operations are used to perform the weighting and adding. For this, the image is rendered as a textured quadrilateral exactly covering the screen so there is a one-to-one correspondence between texels and pixels. However, with most second-generation GPUs it is not possible to actually sample the neighbors of one texel. The way multi-texturing works is that for each pixel, the texture is sampled at the pixel's position using pre-assigned texture coordinates. So instead of sampling a texel's neighbors in one texture, the image is loaded into multiple texture units, with texture coordinates shifted by one pixel [James, 2001]. This makes the former neighbors coincide with the texel (see Figure 8.1).

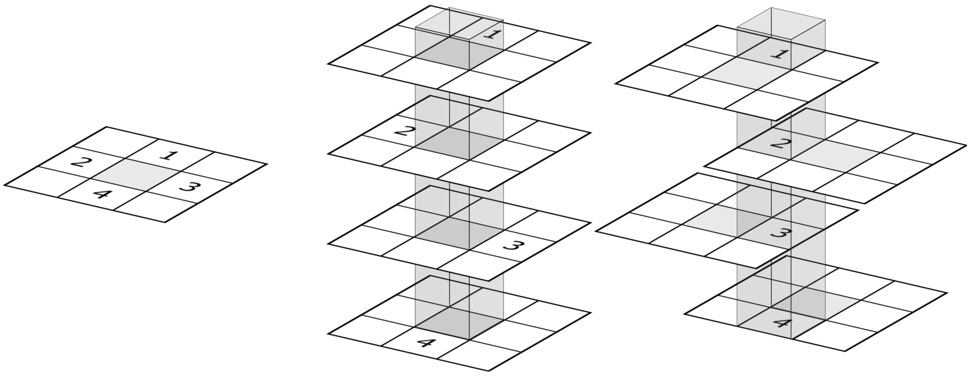


Figure 8.1: Sampling the neighborhood of a texel is implemented by shifting texture coordinates so former neighbors coincide.

Naturally, this technique can only sample as many independent texels as there are texture units. In first-generation GPUs, only 2 or 3 texture units are available, the second generation provides 4 to 6 units. This poses a problem for general filters, which, for the common case of a 3×3 filter, need 9 samples to cover the neighborhood.

In third generation GPUs, a single texture unit can be sampled more than once. For example, hardware that supports the OpenGL ARB_fragment_program extension is guaranteed to allow at least 24 texture operations per pass. However, this flexibility might not be as efficient as the use of the traditional texture access via interpolated texture coordinates, so the following discussion has its merits even for third generation hardware.

8.2 Rendering G-Buffers

Since the filtering as explained above requires the G-buffers to be available as textures, the scene first has to be rendered as a texture. Instead of performing lighting calculations, however, the geometric attributes are converted to colors (see Figure 8.2).

This rendering is most efficient if the hardware supports directly setting the render target to a texture (using the OpenGL render-to-texture extension). Another possibility is to render into a frame buffer and copy it to the texture. The frame-buffer can be the back-buffer or an off-screen buffer (using OpenGL p-buffer extension). If the hardware supports

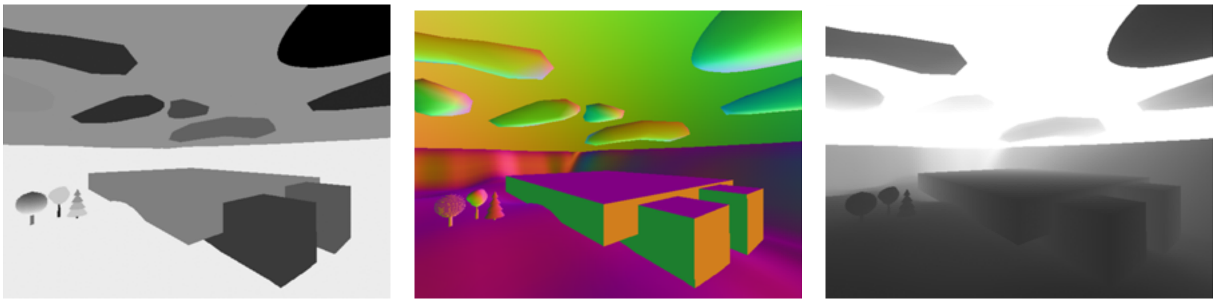


Figure 8.2: G-Buffers are rendered to textures. Object id (left), normal (center), depth (right).

non-power-of-two textures, the texture can be made exactly as large as the frame buffer, otherwise the texture size must be rounded up to the next power of two.

The mapping from geometric attributes to color values can be achieved with a vertex program. Doing this allows the same geometry data to be used for both regular and G-buffer rendering. A common convention for mapping from normals to colors is to compress the range of each component (x, y, and z) from the interval $[-1, 1]$ to the range representable as colors $[0, 1]$. This is demonstrated in this vertex program:

```
struct vert2frag {
  float4 position : POSITION;
  float4 color : COLOR0;
};

vert2frag gbuffer_vert(float4 pos : POSITION,
                       float4 normal : NORMAL,
                       uniform float4x4 ModelViewProj)
{
  vert2frag OUT;
  OUT.position = mul(ModelViewProj, pos);
  OUT.color = 0.5 * normal + 0.5;
  return OUT;
}
```

Rendering the normal buffer and depth buffer can be done in one pass by encoding the normal as RGB and the depth as Alpha, and writing to an RGBA texture format. If the hardware supports multiple render targets, even more G-buffers can be rendered in one pass. All texture targets are bound simultaneously, and a fragment program can dispatch color values into their respective targets.

8.3 Sobel Filter on Four-Texture Hardware

The Sobel filter is a first-order differential filter that is commonly used for edge detection. It was recommended for contour finding by [Saito and Takahashi, 1990]. The Sobel filter comes in two variants, a horizontal and a vertical one, which detects discontinuities in the horizontal and vertical gradient, respectively.

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

Figure 8.3: Filter weights for horizontal (left) and vertical (right) Sobel filter.

As can be seen from the weights in Figure 8.3, six of the nine weights are non-zero. That means that six texels must be sampled. The three zero-weighted samples can be omitted because they do not contribute to the filter result. Thus, this filter cannot be applied directly on a four-texture hardware, which is common for the second generation of GPUs.

To implement this filter on such hardware nonetheless, we will take advantage of another texture unit feature: the magnification filter. Rather than point-sampling textures, which takes exactly one sample per pixel by rounding the texture coordinate to the nearest integer, the hardware also can perform *bi-linear filtering* of texels. It works by taking the fractional part of the texture coordinates to weight the nearest texel with its immediate neighbors (see Figure 8.4). Bi-linear magnification filtering is enabled in OpenGL by setting the TEXTURE_MAG_FILTER texture parameter to LINEAR (the default is NEAREST which performs point-sampling).

Figure 8.4: Texture sampling modes: Point-sampling chooses the nearest texel, so the color of texel P is returned (left), while bi-linear texture filtering returns the weighted average of four texels, in this example roughly $\frac{4}{9}P + \frac{2}{9}Q + \frac{2}{9}R + \frac{1}{9}S$ (right). The exact texture coordinate is indicated as black dot, the sampling area is shown as grey square.

We now choose a special texture coordinate setup that allows us to sample more than one texel with each tap. This layout is shown in Figure 8.5. Four texture samples are taken. Each sample is placed exactly on the border between two texels so that those two texels contribute equally to the result.

The upper left texture sample t_0 returns the average of the upper-left and center-left texels. Accordingly, t_1 samples the center-left and lower-left texels, and analogously for t_2 and t_3:

$$\begin{aligned} t_0 &= \frac{1}{2}p_{x-1,y-1} + \frac{1}{2}p_{x-1,y} \\ t_1 &= \frac{1}{2}p_{x-1,y} + \frac{1}{2}p_{x-1,y+1} \end{aligned}$$

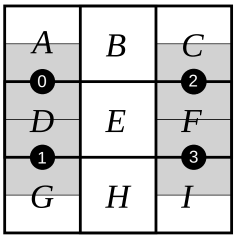


Figure 8.5: Horizontal Sobel filter implemented with four texture samples (indicated as black dots). Sample 0 samples texels A and D, sample 1 samples texels D and G, sample 3 samples texels C and F, sample 4 samples texels F and I. The sum of samples 0 and 1 weighed by -2 and samples 2 and 3 weighed by 2 constitutes the horizontal Sobel filter.

$$\begin{aligned} t_2 &= \frac{1}{2}p_{x+1,y-1} + \frac{1}{2}p_{x+1,y} \\ t_3 &= \frac{1}{2}p_{x+1,y} + \frac{1}{2}p_{x+1,y+1} \end{aligned}$$

By combining these texture samples as follows

$$p'_{x,y} = 2(-t_0 - t_1 + t_2 + t_3)$$

we get exactly the horizontal Sobel operator (recall Figure 8.3):

$$p'_{x,y} = \begin{pmatrix} -p_{x-1,y-1} & + & p_{x+1,y-1} & + \\ -2p_{x-1,y} & + & 2p_{x+1,y} & + \\ -p_{x-1,y+1} & + & p_{x+1,y+1} & \end{pmatrix}$$

The texture coordinate setup and texture combining can be implemented as a custom vertex program and fragment program. The vertex program has to shift texture coordinates to create the sample pattern from Figure 8.5, while the fragment program applies the weights to each sample to get the result.

The following Cg programs assume that a non-power-of-two (NPOT) texture, also know as “rectangle texture,” is used. These differ from regular power-of-two textures in that they can have arbitrary dimensions, and in NVIDIA’s implementation they are accessed with non-normalized texture coordinates. This means the texture coordinates are not from the interval $[0, 1]$, but $[0, s]$ where s is the size in texels. The offset from one texel to the next is $\frac{1}{s}$ for regular textures, but 1 for NPOT textures. We also assume that the Model-View-Projection matrix is set to provide a one-to-one mapping of texels to pixels when a quad is rendered to the screen.

```
struct vert2frag {
  float4 position : POSITION;
  float2 texcoord0 : TEXCOORD0;
  float2 texcoord1 : TEXCOORD1;
  float2 texcoord2 : TEXCOORD2;
  float2 texcoord3 : TEXCOORD3;
```

```
};

vert2frag sobel_vert(float4 pos : POSITION,
                     uniform float4x4 ModelViewProj)
{
  vert2frag OUT;
  OUT.position = mul(ModelViewProj, pos);
  OUT.texcoord0 = pos.xy − float2(−1.0, −0.5);
  OUT.texcoord1 = pos.xy − float2(−1.0, 0.5);
  OUT.texcoord2 = pos.xy − float2( 1.0, −0.5);
  OUT.texcoord3 = pos.xy − float2( 1.0, 0.5);
  return OUT;
}

float4 sobel_frag(vert2frag IN,
                  uniform samplerRECT texunit0 : TEXUNIT0,
                  uniform samplerRECT texunit1 : TEXUNIT1,
                  uniform samplerRECT texunit2 : TEXUNIT2,
                  uniform samplerRECT texunit3 : TEXUNIT3) : COLOR
{
  float3 t0 = texRECT(texunit0, IN.texcoord0).xyz;
  float3 t1 = texRECT(texunit1, IN.texcoord1).xyz;
  float3 t2 = texRECT(texunit2, IN.texcoord2).xyz;
  float3 t3 = texRECT(texunit3, IN.texcoord3).xyz;
  float3 sobel = 2.0 ∗ (−t0 − t1 + t2 + t3);
  return float4(sobel, 1.0);
}
```

The resulting edge value can be scaled to enhance the contrast, and then used as alpha value to blend the contours over the frame buffer. In a second pass, this process has to be repeated analogously with the vertical version of the Sobel filter.

8.4 Denormalization Filter

In the previous section, bilinear texture filtering was used to sample more texels than there are texture units. In this section, we will devise a method that uses bilinear filtering exclusively to detect contours on normal buffers with a single texture unit.

This new method builds on the fact that normals cannot be interpolated correctly with linear interpolation. The more disparate the normals are, the more de-normalized the result gets. This is a problem in regular rendering, because to obtain accurate results, the interpolated normals have to be re-normalized. This is an expensive operation because it involves a square root and a division.

In our approach, we detect the de-normalization of normals caused by linear interpolation. Specifically, the de-normalization is introduced by bilinear sampling of a normal buffer.

When two adjacent normals are equal, their interpolated result will be equal, too, and still have unit length. However, if the two normals differ considerably, the interpolated normal's length will be less than one. This denormalization is detected in a fragment program and used to draw contours (see Figure 8.6).

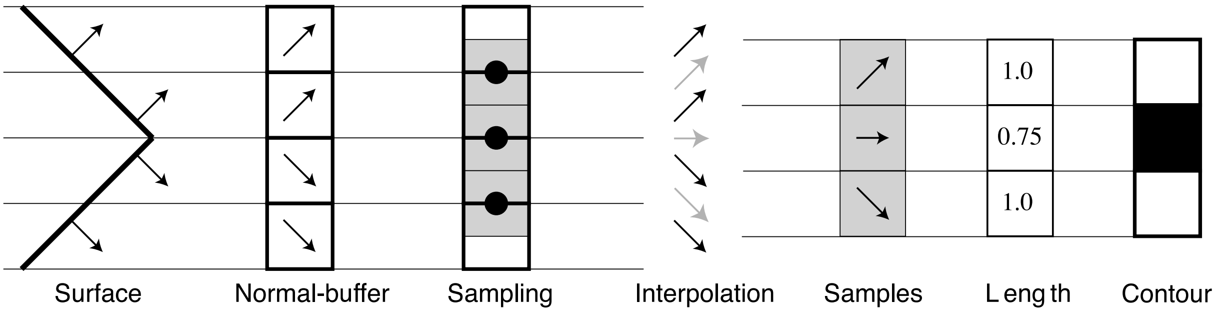


Figure 8.6: Contour detection from a single texture. A surface is rendered to a normal buffer, which is sampled as texture with bilinear filtering so that each pixel samples adjacent texels. The bilinear filtering linearly interpolates adjacent normals. If the original normals differed considerably, the resulting normal is shorter than 1.0. This is detected in a fragment program and used to draw a contour.

To take full advantage of bilinear filtering, the texture coordinates are shifted by half a texel with respect to pixels (see Figure 8.7). Thus every pixel samples four texels.

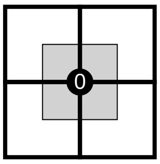


Figure 8.7: Texture coordinates are shifted by half a texel with respect to pixels.

The detection of a non-unit length of the sampled normals would normally require a square root. However, we are only interested in determining if the length of the interpolated normal N is less than 1:

$$\begin{aligned} |N| = \sqrt{N \cdot N} &< 1 \\ N \cdot N &< 1^2 = 1 \end{aligned}$$

So it is sufficient in the fragment program to use the dot product of the interpolated normal with itself, and compare this to 1. In practice, a small difference between two normals should be neglected because it indicates a smooth surface. Only larger differences should be detected as contours. So instead of comparing to 1, we use a smaller number $1 - \epsilon$. In addition, a smooth threshold instead of a hard one is used (recall Section 6.2.4):

```
struct vert2frag {
  float4 position : POSITION;
  float2 texcoord0 : TEXCOORD0;
};
```

```
vert2frag denorm_vert(float4 pos : POSITION,
                      uniform float4x4 ModelViewProj)
{
  vert2frag OUT;
  OUT.position = mul(ModelViewProj, pos);
  OUT.texcoord0 = pos.xy − float2(−0.5, −0.5);
  return OUT;
}

float4 denorm_frag(vert2frag IN,
                   uniform samplerRECT normalbuffer : TEXUNIT0,
                   uniform float epsilon) : COLOR
{
  float3 N = texRECT(normalbuffer, IN.texcoord0).xyz;
  float l = dot(N,N) + epsilon;
  l = 1.0 − 4.0 ∗ (1.0 − l);
  return float4(l, l, l, 1.0);
}
```

For this method to work, it has to be ensured that the normals stored in the G-buffer are in fact normalized. The usual interpolation of normals as colors as shown in Section 8.2 might not be sufficient for this if the geometry is too coarse. A normalization of the interpolated normal should be performed for each pixel when the G-buffer is rendered. This can be implemented by a normalization cube map or in a fragment program.

8.5 Comparison

The two approaches, Sobel filter and detecting denormalization (Denorm), both operate on G-buffers. However, Sobel requires two passes and 4 texture units, while Denorm is a one-pass single texture approach. Thus, Denorm runs on a wider range of hardware and is usually faster.

Sobel delivers contours of significantly higher quality (see Figure 8.8). The quality of the Denorm filter can be improved by per-pixel re-normalizing the normals in the G-buffer.

An additional shortcoming of the Denorm approach is that in its current form, it can only be applied to normal buffers, but neither to depth buffers nor to *id* buffers. This leads to some missed contours. Perhaps a mapping from *id*s or depth values to normalized vectors could be employed to enable the use of these attributes with the Denorm filter, a study of this is left for future work.

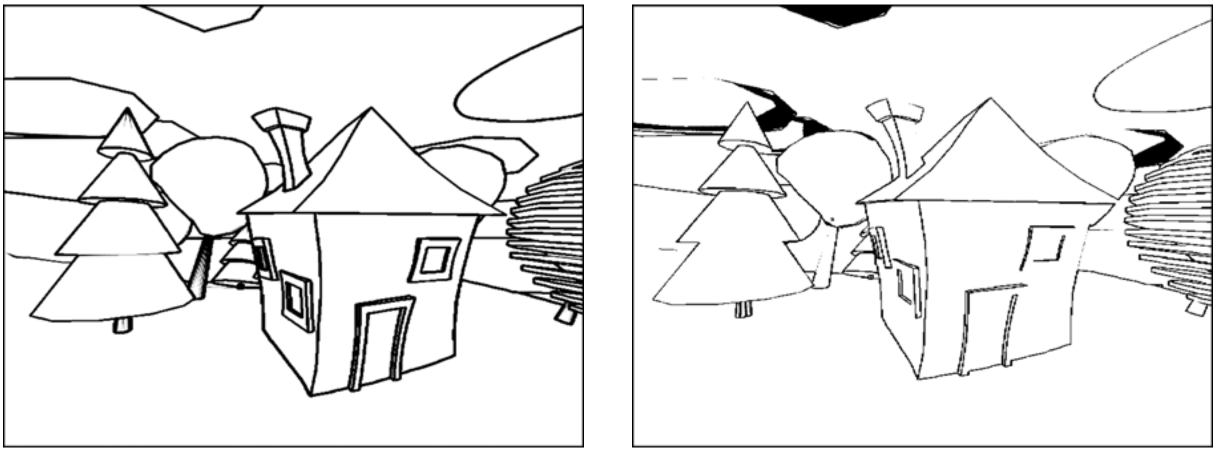


Figure 8.8: Comparison between Sobel contour detection (left) and Denorm contour detection (right). Sobel finds more lines and produces stronger strokes. The blackened areas in the Denorm image come from not explicitly normalizing the normals in the G-buffer.

8.6 Contrast-Based Outlines

So far, only uniformly-colored contours have been discussed. This is sufficient if the chosen color (black in most cases) does contrast with the surface colors used throughout the scene. In particular, it works well with discretely shaded objects (cel shading), because the colors are known in advance, rather then being computed at run time by evaluating a lighting model.

However, in arbitrary scenes it is not always the case that a contour is not visible. This happens when a surface adjacent to the contour is of the same color as the contour. Two cases are possible. If the surface on one side of the contour is contrasting with the surface on the opposite side, no contour is actually necessary (which might be exploited to reduce the number of contours drawn [Winkenbach and Salesin, 1994]). If both sides happen to be of similar color as the contour itself, the contour will barely be visible. It is desirable to enhance the contrast between the contour and the surfaces.

8.6.1 Contrasting Double Contours

A simple method for ensuring contours are visible independent on the background color is to draw the contour in two colors. Regardless of what color the “underlying” surface is, one or the other contour color will be in contrast to it. We present a texture-based approach here.

The basic idea is two draw the contours twice using a different color for each pass. The second pass must be offset from the first one, otherwise the contours rendered first would simply be overdrawn.

The algorithm proceeds as follows. First the scene is rendered. Next the first contour pass is performed. The contour is used as the alpha value, the color is set to the first contour color. Up to this point, this is exactly the contour algorithm as explained above. Now the contours are drawn again, but the second contour color is selected, and the G-buffer texture coordinates are offset by one pixel to the lower right (see Figure 8.9 for an example).

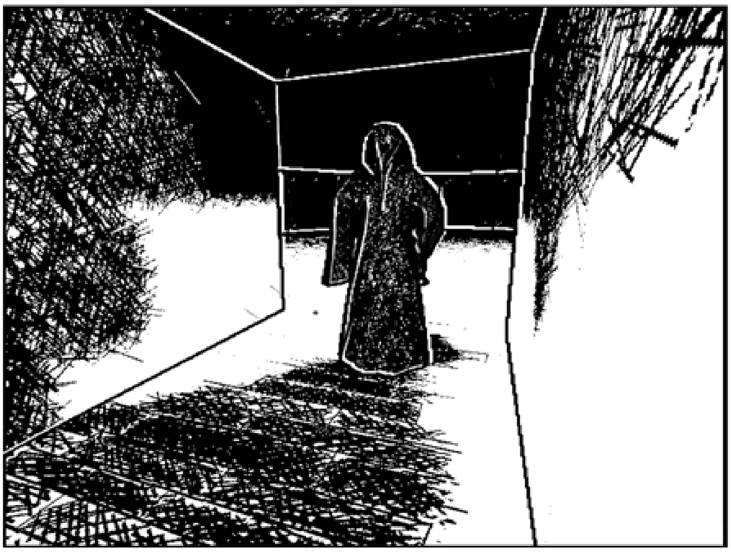


Figure 8.9: Example of a scene rendered with contrasting double contours. Note how edges are visible in both dark and light regions.

We chose to offset the second pass by a fixed amount in screen space. This is not optimal. A better solution would be to draw the second contour "outwards" of the object. However, the offset would not be uniform for all pixels, which could not be implemented by adjusting texture coordinates. Moreover, "outwards" is only defined for silhouette edges, not for inner contours. If a way was found to adjust the offset for each pixel, then a worthwhile offset direction might be derived from the lighting direction, so that the darker contour always points away from the light.

The major drawback of this technique is that contours are still not guaranteed to be visible. If the direction of a contour coincides with the offset direction, the second pass will overdraw the first one. If the surface beneath happens to be of the second contour color, no contour can be discerned.

8.6.2 Color-Adjusted Contours

A promising idea to be explored is to ensure visibility and contrast of contour lines by analyzing the contents of the frame buffer to decide how an outline should look like. Instead of simultaneously drawing contours in two colors, the color of the contour would be adapted to contrast with the surface.

One approach could be to simply use the inverse color of that found in the frame buffer when drawing a contour. This assumes that the inverse does contrast with the original color. However, for colors like mid-level gray this is not true. Also, the inverse color might not be esthetically pleasing if performed in RGB space (for example, red $[1, 0, 0]$ is the inverse of cyan $[0, 1, 1]$).

A better approach would switch between two contour colors. A dark contour color will is used in light areas, a light contour color in dark areas. To avoid repeatedly switching between the two colors in areas of high visual detail, a blurred version of the frame buffer should be used to estimate the average brightness at a pixel.

Another possible enhancement of this algorithm would be to detect a visual contour in the frame buffer. If there is a contour found both in the frame buffer and in G-buffer, the contour does not have to be drawn at all.

It remains to be seen in future work if this approach will work and deliver satisfying results.

9 Case Studies

Some of the real-time halftoning techniques developed in this thesis have been applied to larger projects. The following case studies will discuss the motivation of incorporating halftoning techniques in an application context, as well as reporting about practical experience drawn from an actual implementation of these methods.

Section 9.1 deals with the interactive presentation of a virtual reconstruction. An architectural model derived from archaeological findings is presented in the context of today's actual environment. The environment is rendered photorealistically, while the reconstruction is depicted in a halftoning style. This emphasizes the ability of the technique developed to be integrated with traditional rendering methods.

In Section 9.2, a pen-and-ink style computer game is presented. It was created by taking an existing game, which was originally rendered in a photorealistic style, and converting it into a comic-like appearance. We demonstrate the ease of integration of our implicit halftoning in common-place, texture-based applications.

9.1 Archaeological Walk-Through

In a virtual reconstruction of a historical site, facts, assumptions and fiction exist side by side to create a convincing illusion of seeing into the past. Especially in a museum, exhibitors have to consider carefully what images they convey to the public, as people tend to take a given picture for scientifically proven truth. Viewers, however, cannot distinguish between what has been scientifically proven, what is the result of careful reasoning, and what has simply been made up to fill in gaps in an image. Also, even experts are sometimes distracted by the amount of detail shown in a photorealistic image which may not represent their own level of knowledge with the object being portrayed.

The application of non-photorealistic rendering methods to the field of archaeology extends the repertoire of available presentation options (also called presentation variables [Noik, 1994]). The ability to select between different types of visualization enables a user to choose the form of presentation that best serves the given communicative purpose best.

9.1.1 The Project

From 1959 to1968, an excavation was carried out at the Cathedral Square in Magdeburg. Remnants of a large building were found. Because of the unusual floor plan, the findings were categorized as foundation walls of the “Aula regia”, the main palace of Otto the Great, the first German emperor who lived from 912 to 973 [Nickel, 1973]. In 1986, the historian Cord Meckseper attempted a reconstruction of this building using the preliminary excavation data [Meckseper, 1986]. Based on his drawings, students of our department built a virtual reconstruction in an animation course. It resulted in a photorealistic animation which was publicly shown in exhibits and on public television. The success of this work ignited the idea of presenting the virtual reconstruction in an exhibition about Otto the Great in the Magdeburg Museum of Cultural History, which was scheduled for 2001.

In 1998 the archaeologist Babette Ludowici started an in-depth analysis of the excavation protocols, drawings, and findings. Then, in 1999, she found something that almost would have stopped the project: The analysis of the stratigraphic sequence invalidated the quadratic plan, it proved that remnants of two buildings stemming from different periods were accidentally mistaken as one [Ludowici, 2000]. This invalidated the reconstruction as we had it previously shown.

Consequently, the concept had to be changed. Conveying a false impression of the original site should not easily happen again. So we started research into visually depicting uncertainty about virtual reconstructions [Strothotte et al., 1999]. In 2000, four students began working on visualization components, based on the Shark 3D game engine. They employed rendering techniques developed in this work. The new interpretation of the Ludovici’s findings was again provided as drawings by Cord Meckseper and his assistant, Maike Kozok. A virtual reconstruction, built by emergent media AG based on this prototype, was shown in the Magdeburg Museum of Cultural History from August to December 2001. The exhibition about Otto the Great attracted 300,000 visitors, of which about half actually viewed the virtual reconstruction [Freudenberg et al., 2001a; Masuch and Strothotte, 2001].

9.1.2 Real-Time Implementation

When we created the prototypical walk-through for the museum exhibition, we wanted to emphasize the vagueness of the depicted virtual reconstruction. Although a photorealistic model of the building already had been created (see Figure 9.1), visitors would certainly mistake the presentation as a fact, rather than speculation, which it in fact was.

Instead, the model was newly textured with halftone screens akin to a sketched brick building. Also, an indication map was hand-painted on the model that emphasized the edges of the model. This mostly eliminated the need of rendering outlines. Only for the round towers, which had no edges that could be indicated, a simple outline algorithm was applied (see Figure 9.2).

Figure 9.1: Virtual reconstruction of the building excavated at Domplatz Magdeburg (image courtesy of Niklas Röber)

The reconstruction was placed in an environment that was modeled to closely resemble today's surroundings of the excavation site. This serves to make the dimensions of the building more comprehensible to the viewers. The surrounding buildings and the plaza are rendered photorealistically, while halftoning is limited to the reconstruction. This is an advantage of using texture mapping and fragment processing for halftoning: If the halftoning had been performed as a full-screen postprocess (which is the standard for traditional halftoning), mixing different rendering styles would not have been so effortless.

What helped immensely in applying the halftone screen textures to the model was that they look rather similar to the rendered image. This way, the final appearance can already be judged in the modeling tool; this not possible if the screens had used the "normal" halftone screen convention. If this convention had been adopted, higher values in the halftone screen would mean a higher ink priority. If such a halftone screen itself is viewed as an image, it would appear white where the black ink will be placed.

9.1.3 Consequences for the Application

The use of real-time halftone rendering had several implications on this exhibition project. For one, real-time rendering allows interactivity. This enables the presenter to adjust the pace of a demonstration to the reactions of the audience. Not only that, but a completely different path through the virtual exhibition can be chosen depending on the visitors and the presenter. This also decouples the authors of the presentation from the ones presenting it - modifications in the schedule would normally require the authors to rewrite the animation. With an interactive walk-through, the users themselves can decide on what to explore next.

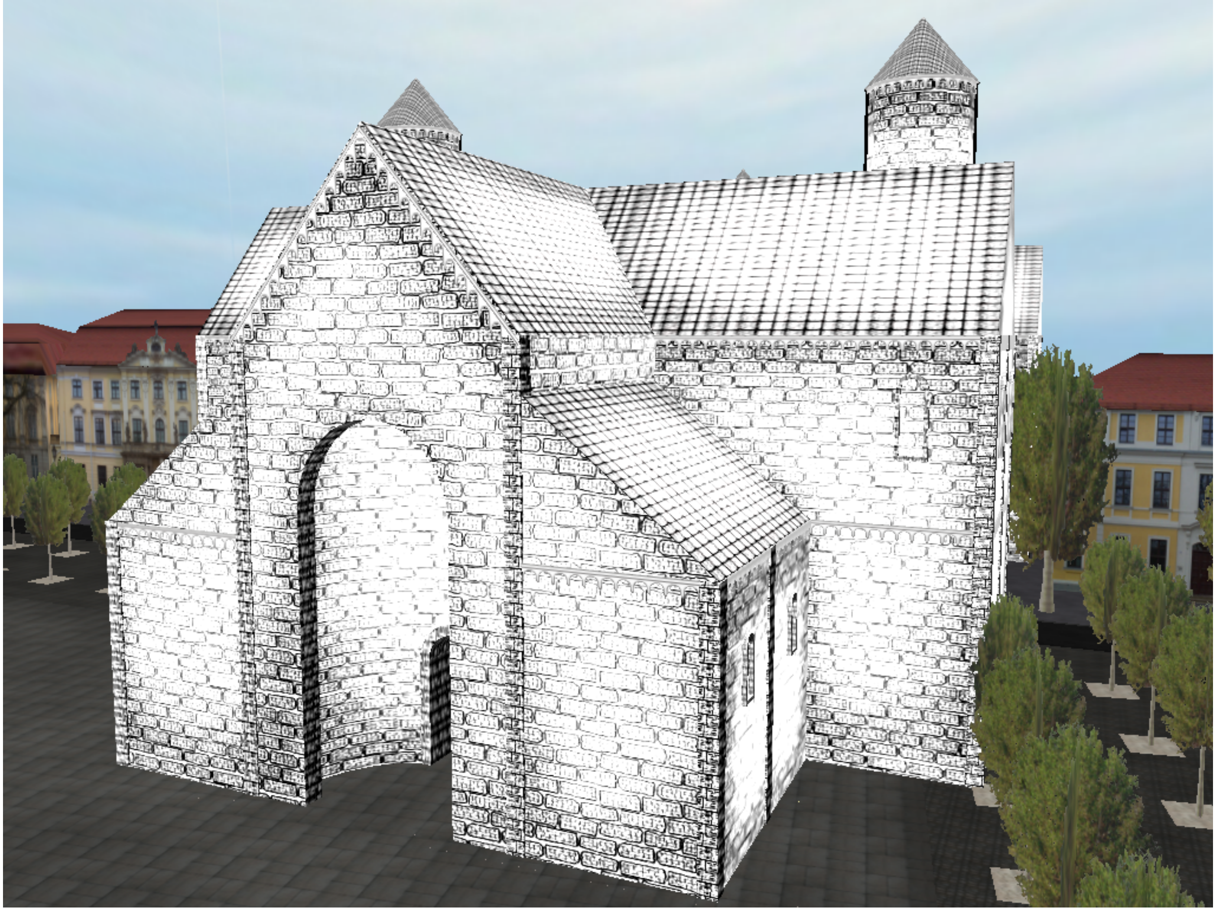


Figure 9.2: The reconstructed building is rendered with real-time halftoning, while the surrounding environment is shown in a photorealistic style.

On the other hand, an interactive application would have required more trained staff than just showing a pre-rendered animation. However, this was solved by implementing an autonomously running presentation that could be interrupted if desired. This combined both advantages of interactivity and low maintenance.

The choice of the implicit halftoning technique for the presentation of the virtual reconstruction assured both cost-effectiveness and quality. It was cost-effective because cheap PCs could be used to run the presentation without requiring expensive special purpose hardware. Also, the models were relatively easy to create and the modifications of the game engine have been minimal. The high quality of the animation largely stems from the intrinsic frame-coherence of the texture-based shading approach. Because of the frame-coherence, the animation is unobtrusive and does not draw as much attention to the presentation style as is observed in some hand-made animations. It still communicates the vagueness of the reconstruction very clearly, even without a comment a visitor can comprehend the uncertainty of the images shown.

9.2 Computer Games

Another desirable application of real-time non-photorealistic rendering is in creating new visual styles for computer games. Until now, three-dimensionally rendered computer games almost exclusively use photorealistic rendering techniques. That is not to say they look like a photograph, because very dramatic effects can be achieved “just” by using colored lighting, hand-painted textures, etc. But the rendering process itself is a photorealistic one, meaning that a uniform lighting model is applied to each surface in the scene which is simply projected to the screen. The only exception to this is cartoon shading, as was discussed in Section 4.4.3.

In contrast to this, 2D games more often have a unique visual style. This is facilitated by hand-painted backgrounds and sprites. If a similarly unique drawing style could be applied to a 3D rendering, this would give a game developer an additional tool to distinguish them from their competition. It can also become possible to capture the specific style of a comic book in a 3D adaptation.

9.2.1 Modifying a Game Engine

To verify the suitability of real-time halftoning in a game setting, we used an actual game engine to implement a demo level. We chose the Shark 3D engine, because it provides a modular shader source code that allows to adjust texturing parameters without source code modifications [Spinor GmbH, 2002]. In addition, students of our department had already created a demo level while doing an internship [Punkt im Raum, 2000], albeit in a photorealistic style (see Figure 9.3).

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.

9.2.2 Results

The converted game ran as smoothly as before. This came at no surprise since only the texturing was changed and the used shaders were not more complex than the photorealistic shaders before. Since the same texture parameterization as for the photorealistic version was used, even the animations (in this case, the bone-animated monk) play back perfectly.

Figure 9.3: Original game scene [Punkt im Raum, 2000].

A drawback of the high-contrast fine-detailed black and white structures became visible only on liquid crystal displays (LCDs). A moving high-contrast image such as produced by real-time halftoning causes a slight “unsteadiness”. It is noticeable that the scene looks slightly different when the camera is in motion than when the viewer stays still. The effect is hard to describe and is does not occur with CRT monitors. It seems to be caused by LCDs having a higher response time than CRT monitors, the pixels cannot switch from “off” to “on” as fast as required. With less-contrasting imagery (as common in photorealistic rendering), the effect is much less pronounced. It remains to be seen if the development of new low-response time LCDs will make remedy this.

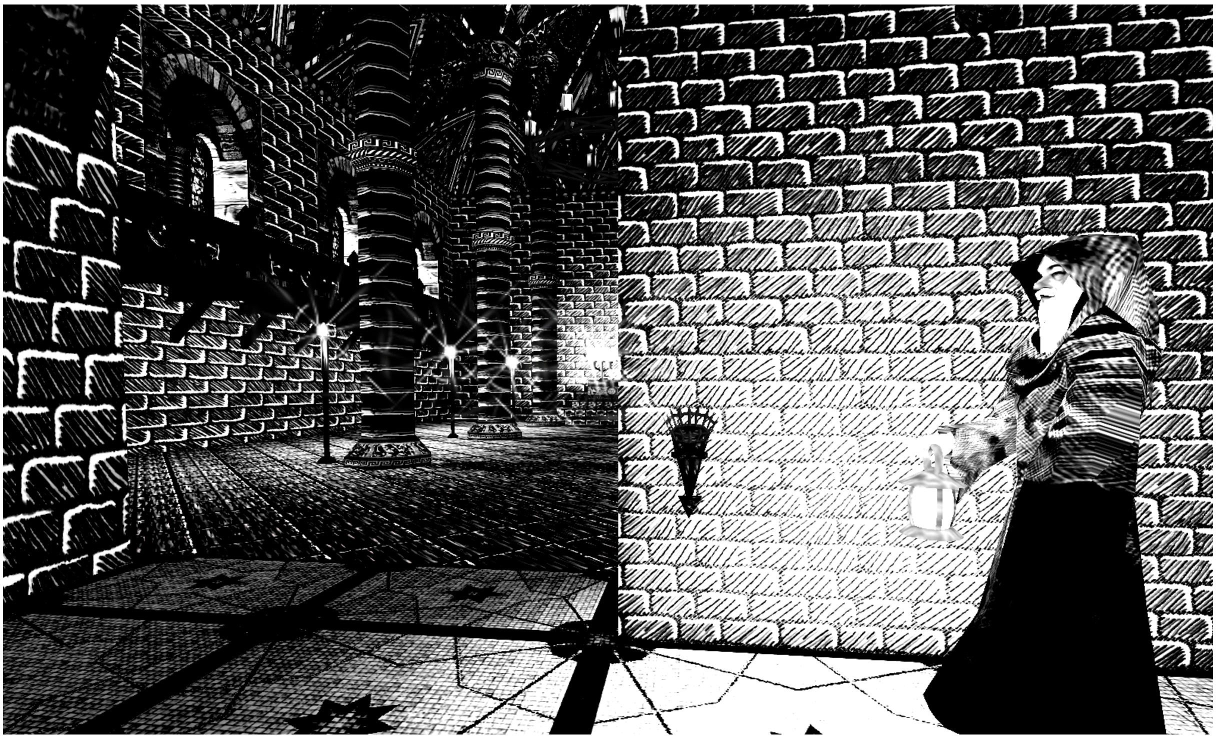


Figure 9.4: Converted game scene. New pen-and-ink style halftone screens were drawn for use on walls and floor, while most other textures were only converted to gray scale

10 Concluding Remarks

10.1 Summary

This work started out with the goal to develop techniques for non-photorealistic rendering that are usable in interactive applications. After reviewing research and applications dealing with real-time NPR, a particularly underdeveloped field was identified: non-photorealistic shading techniques. While fast outline rendering methods were already well researched and in common use, only very few successful attempts to shade surfaces in a non-photorealistic style that ran in real-time had been reported on in the literature.

At first, our research focused on texture-based NPR shading methods. The goal was to simulate shading with strokes of varying density and thickness. Several problems were identified, such as how to ensure frame coherence, avoiding the shower-door effect, maintain a certain stroke density even when objects move in the distance, and how to dynamically adjust stroke density and stroke width according to lighting.

Among the early results was a method to dynamically adjust stroke density by the use of specially designed mipmap textures. This allowed stroke textures to maintain a certain visual density over a wide range of distances. The problem of frame coherency and the shower-door effect were addressed by attaching strokes to surfaces. Indeed, this method was to turn out to be the seed for the development of more general methods which followed.

A method was developed to encode strokes as bitmap textures and render them in real-time, thinning out strokes depending on lighting. The fundamental technique was extended to allow control over individual strokes. A key building block for algorithms, the "warp-map" technique, was then introduced to group pixels in a stroke, even though they are processed independently. This concept, being able to apply an operation to a group of elements which are nonetheless executed independently represents a breakthrough in the quest for real-time NPR. Since in the bitmap-texture based shading scheme the actual image elements are not geometrically defined, but latent in the halftone texture, we call this "implicit halftoning". Only in the rendering process the elements manifest in the image.

The counterpart, "explicit halftoning" was not developed until later. In this scheme, the geometric extent of halftone marks is explicitly calculated and rasterized to the image. Since strokes are drawn individually, a higher level of control over individual strokes becomes possible. This applies the idea of threshold values to strokes for the first time.

Again, it allows data to be organized into independent units that can independently worked on in parallel, which enabled the successful implementation of this concept in hardware.

Since most rendering styles encountered in this thesis benefit from outline rendering, it became necessary to devise new, compatible methods here, too. These efforts were successful, so that both implicit and explicit outline rendering techniques could be developed.

10.1.1 Hardware Acceleration

The successful implementation of NPR rendering techniques require the use of graphics hardware in novel ways. Since the hardware still is designed to be used in a photorealistic context, it required programming on both vertex and pixel level to efficiently implement non-photorealistic methods. Moreover, a significant challenge is to lay out the data rapid such that can be accessed rapidly by the graphics hardware. This requirement is much less severe when implementing NPR algorithms on a general-purpose CPU.

We realized that modern graphics accelerators are optimized towards rendering textured polygons, hence we sought methods to exploit these for implicit halftoning techniques. We found that features which originate in photorealistic rendering could in fact be rededicated to new purposes in NPR. For example, bump mapping can be used to light strokes individually. Other features can also be used in innovative ways, like dependent texture reads to realize warp maps. Moreover, it pays even to look at unconventional only data to feed to the GPU, like the specially designed mipmap textures that are not constructed by filtering as is usually done.

In the case of explicit halftoning, the methods had to be even more tailored towards what the graphics hardware currently allows. Since algorithmically creating or removing primitives is not yet possible on the GPU, excessive numbers of strokes were generated and conditionally disabled or modified in vertex programs. Redundant data had to be stored in graphics memory because the current shader architecture is too inflexible to allow, for example, generating a polygon from a stroke description.

10.1.2 Spatial Shading vs. Stroke-Based Halftoning

We realized the shading techniques developed in this thesis were actually more general than simply stroke textures, which led to the new term “spatial shading” and later “stroke-based halftoning”.

By using the term “spatial shading”, we wanted to emphasize the spatially distributed nature of image intensities produced by shading an image with a series of image elements, such as strokes or dots. In contrast to conventional shading, pixels on the surface do not proportionally adapt their intensity to reflect a change in lighting. Rather, the ratio of light to dark regions in the image is changed, which manifests itself as a perceived change in image brightness to the viewer's eye. This spatiality aspect is not as dominant in the context of "halftoning", which traditionally focuses on minimizing the visibility of patterns in the halftoned image. Also, the driving force behind halftoning is the need to accurately present continuous-tone images with only black or a limited number of inks on white paper, a restriction we do not have in interactive graphics. For the most part, in non-photorealism we do not even care about an exact reproduction of tone, because we are more interested in the halftone-artifacts themselves.

However, we later found the term "spatial shading" too general and confusing. It evokes the idea of three-dimensionality while it actually was meant to refer to image space. We think the term "stroke-based halftoning" more accurately describes the category of techniques we sought to explore in this thesis. Besides, the traditional focuses of halftoning have already been extended by research into "artistic halftoning". Since our techniques can certainly benefit from further developments in halftoning, this term might even lead to further synergies between off-line and real-time halftoning methods in the future.

10.2 Limitations

Real-time non-photorealistic rendering is still very much in its infancy. Many NPR techniques are available, but not yet practical to be implemented in real-time.

In some respects, it is similar to the state of photorealistic rendering 20 years ago. For example, hidden surface removal is a basically considered a solved problem today, and the hardware implementation of the z-Buffer algorithm is supposedly fast and robust. However, since it operates on single pixels, rendering errors occur in the context of NPR. The basic image element to be hidden might be a brush stroke or wide silhouette line, which is larger than a pixel. Consequently, a stroke might be partially hidden, thus it appears clipped. In this thesis, this was counteracted by manually choosing offsets so these errors are kept to a minimum, however this is neither a general solution, nor is it elegant.

A hidden surface removal method better suited for NPR might be the painter's algorithm. However, sorting scene elements for each frame depending on distance is fundamentally incompatible with today's hardware acceleration architectures, which work best when a large number of primitives is displayed without touching the data over many frames. As CPUs get faster and the bandwidth to transfer data to the GPU increases, it might be possible to do the sort operation on the CPU and transmit all data to the GPU in each frame. However, it will be still far less efficient than z-buffered rendering.

The fastest methods for high-quality silhouette generation require a great amount of preprocessing, so they do not work on animating meshes yet. Implicit silhouette-finding algorithms using image-processing operations are general and of constant complexity, but are prone to miss edges and do not allow as much control of the appearance of outlines.

A limitation of current display hardware became evident with our halftoning applications. While traditional CRT monitors have no problem displaying our scenes, on LCD monitors the quality is degraded in motion. The images look fine without motion, but when animated, the display is not fast enough. The unusually high contrast of the rendered images generated by animated halftoning needs pixels on the screen to switch from black to white and back very rapidly. Even on TFT displays with a low response time the effect is noticeable. The image appears darker than when standing still because pixels are in continuous transition, and a flickering is observable.

10.3 Future Work

Non-photorealistic rendering styles will find their way into interactive applications. There are several factors that may influence the development and adoption of new rendering techniques.

More shading methods will need to be created. While this thesis explored a particular style of rendering drawing-like images, many more styles are possible. Techniques used in traditional media can serve as a source of inspiration for new automated techniques.

New graphics hardware could aid in efficiently implementing new NPR algorithms. For example, if geometry manipulation was possible on the graphics board, strokes as used in explicit halftoning could be generated more efficiently. Implicit techniques would benefit from a more direct support of image processing techniques, which currently have to be emulated by texture processing. Also, exact control of the texture filtering process is necessary to guarantee a certain appearance when manipulating mipmaps.

What is also needed for researchers in NPR is to find a general and common vocabulary. For example, “stroke-based halftoning” might provide a useful demarcation of shading methods similar to those developed in this thesis.

Non-photorealistic real-time rendering still has a long way to go. The problems are not only of technical nature. To be successfully employed in real-world applications, NPR methods need to be accepted by programmers, designers, and artists alike. Hopefully, some of the techniques presented in this thesis do fit well into the production pipeline of computer games. Many rendering styles are possible, and they will assist the creativity of game developers as well as developers of other interactive media.

Bibliography

Kurt Akeley. The Hidden Charms of Z-Buffer. *IRIS Universe*, (11):31–37, 1990.

Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering.* A.K. Peters Ltd., 2nd edition, 2002.

Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures.* Morgan Kaufmann, 1999.

Archeoptics Ltd. 3D Laser Scanning for Rapid Recording, Survey and Illustration. Flyer for AAI&S Conference 2001, Archeoptics Ltd., Glasgow, Scotland, UK, 2001. URL `http://www.archaeoptics.co.uk/downloads/presentations/`.

ATI, Evans & Sutherland. ATI's RADEON 8500 graphics technology to power Evans & Sutherland's high-end PC-IG visualization products. Press release, November 26, 2001.

Fabien Benichou and Gershon Elber. Output Sensitive Extraction of Silhouettes from Polygonal Geometry. In *Proc. 7th Pacific Graphics Conference*, pages 60–69, Seoul, Korea, 1999. IEEE Computer Society.

John W. Buchanan and Mario Costa Sousa. The Edge Buffer: A Data Structure for Easy Silhouette Rendering. In *Proceedings of NPAR 2000, Symposium on Non-Photorealistic Animation and Rendering (Annecy, France, June 2000)*, pages 39–42, New York, 2000. ACM.

Johan Claes, Fabian Di Fiore, Gert Vansichem, and Frank van Reeth. Fast 3D Cartoon Rendering with Improved Quality by Exploiting Graphics Hardware. In *Proceedings of Image and Vision Computing New Zealand (IVCNZ)*, pages 13–18, 2001.

Derek Cornish, Andrea Rowan, and David Luebke. View-Dependent Particles for Interactive Non-Photorealistic Rendering. In B. Watson and John W. Buchanan, editors, *Proceedings of Graphics Interface 2001*, pages 151–158, 2001.

W. Crome and G. Hartwich. *Urania Tierreich - Wirbellose Tiere 1*, page 54. Urania-Verlag Leipzig Jena Berlin, 1967.

Balázs Csébfalvi, Lukas Mroz, Hellwig Hauser, Andreas König, and Eduard Gröller. Fast Visualization of Object Contours by Non-Photorealistic Volume Rendering. In A. Chalmers and T.-M. Rhyne, editors, *Proceedings of EuroGraphics 2001 (Manchester, UK, September 2001)*, pages 452–460, Oxford, 2001. NCC Blackwell Ltd.

Oliver Deussen and Thomas Strothotte. Computer-Generated Pen-and-Ink Illustration of Trees. In Kurt Akeley, editor, *Proceedings of SIGGRAPH 2000 (New Orleans, July 2000)*, pages 13–18, New York, 2000. ACM SIGGRAPH.

Frédo Durand, Victor Ostromoukhov, Mathieu Miller, Francois Duranleau, and Julie Dorsey. Decoupling Strokes and High-Level Attributes for Interactive Traditional Drawing. In S. J. Gortler and K. Myszkowski, editors, *Rendering Techniques 2001. Proceedings of the 12th Eurographics Workshop on Rendering, (London, June 2001)*, pages 71–82, Berlin · Heidelberg · New York, 2001. Springer-Verlag.

N. Dyn, D. Levin, and J. Gregory. A 4-point interpolatory subdivision scheme for curve design. *Computer Aided Geometric Design*, 4(4):257–268, 1987.

Gershon Elber. Interactive Line Art Rendering of Freeform Surfaces. In Pere Brunet and Roberto Scopigno, editors, *Proceedings of EuroGraphics'99 (Milano, Italy, September 1999)*, pages 1–12, Oxford, 1999. NCC Blackwell Ltd.

Tom Forsyth. NPR Rendering. Posted to Game Development Algorithms Mailing List, `https://lists.sourceforge.net/lists/listinfo/gdalgorithms-list`, 11 November 2003.

Bert Freudenberg. Real-Time Stroke Textures. In *SIGGRAPH 2001 Conference Abstracts and Applications*, page 252, New York, 2001a. ACM SIGGRAPH, ACM Press. URL `http://isgwww.cs.uni-magdeburg.de/~bert/publications/Freudenberg-2001-RTS.pdf`.

Bert Freudenberg. Subdivision for Line Drawings. In *Proceedings Simulation and Visualization 2001*, pages 215–222, Magdeburg, 2001b. URL `http://isgwww.cs.uni-magdeburg.de/~bert/publications/Freudenberg-2001-SFL.pdf`.

Bert Freudenberg and Maic Masuch. Non-Photorealistic Shading in an Educational Game Engine. In Ralf Dörner, Christian Geiger, Paul Grimm, and Michael Haller, editors, *Workshop Proceedings Production Process of 3D Computer Graphics Applications – Structures, Roles, and Tools*, pages 45–48, Aachen, 2002. Shaker. ISBN 3-8322-0241-2.

Bert Freudenberg, Maic Masuch, Niklas Röber, and Thomas Strothotte. The Computer-Visualistik-Raum: Veritable and Inexpensive Presentation of a Virtual Reconstruction. In Stephen N. Spencer, editor, *VAST 2001: Virtual Reality, Archaelogy, and Cultural Heritage*, pages 97–102; 365–366, New York, 2001a. ACM.

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, 2001b. ISSN 1067-7055. URL `http://wwwisg.cs.uni-magdeburg.de/graphik/pub/files/Freudenberg_2001_WTI.pdf`.

Bert Freudenberg, Maic Masuch, and Thomas Strothotte. Real-Time Halftoning: A Primitive for Non-Photorealistic Shading. In Paul Debevec and Simon Gibson, editors, *Rendering Techniques 2002: Proceedings of the 13th Eurographics Workshop on Rendering (Pisa, Italy, June 26–28, 2002)*, pages 227–231, 331, Aire-la-Ville, Switzerland, 2002. Eurographics Association.

Bert Freudenberg, Maic Masuch, and Thomas Strothotte. Real-time halftoning: Fast and simple stylized shading. In Andrew Kirmse, editor, *Game Programming Gems 4*. Charles River Media, 2004. To appear.

Amy A. Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A Non-Photorealistic Lighting Model for Automatic Technical Illustration. In Michael Cohen, editor, *Proceedings of SIGGRAPH'98 (Orlando, July 1998)*, pages 447–452, New York, 1998. ACM SIGGRAPH.

Bruce Gooch and Amy Gooch. *Non-Photorealistic Rendering.* A K Peters, Ltd., Natick, 2001.

Bruce Gooch, Peter-Pike J. Sloan, Amy A. Gooch, Peter Shirley, and Richard Riesenfeld. Interactive Technical Illustration. In Stephen N. Spencer, editor, *Proceedings of the Conference on the 1999 Symposium on Interactive 3D Graphics*, pages 31–38, New York, April 1999. ACM Press.

Nick Halper, Tobias Isenberg, Felix Ritter, Bert Freudenberg, Oscar Meruvia, Stefan Schlechtweg, and Thomas Strothotte. OpenNPAR: A System for Developing, Programming, and Designing Non-Photorealistic Animation and Rendering. In Jon Rokne, Reinhard Klein, and Wenping Wang, editors, *Proceedings of Pacific Graphics 2003*, pages 424–428. IEEE, 2003. (Short paper).

Jörg Hamel. *A New Lighting Model for Computer Generated Line Drawings.* PhD thesis, 2000.

Russ Herrell, Joe Baldwin, and Chris Wilcox. High-Quality Polygon Edging. *IEEE Computer Graphics and Application*, 15(4):68–74, July 1995.

Aaron Hertzmann and Denis Zorin. Illustrating Smooth Surfaces. In Kurt Akeley, editor, *Proceedings of SIGGRAPH 2000 (New Orleans, July 2000)*, pages 517–526, New York, 2000. ACM SIGGRAPH.

Elaine R. S. Hodges. *The Guild Handbook of Scientific Illustration.* van Nostrand Reinhold, New York, 1989.

Axel Hoppe, Kathrin Lüdicke, and Ray Hausmann. Emphasis in rendered images using contrast information. In Ali Behrooz, editor, *Proceedings Knowledge Transfer '96 (London, July 1996)*, pages 513–522, London, 1996. Pace, Pacific & Middle East Center for Research. URL `http://wwwisg.cs.uni-magdeburg.de/graphik/pub/files/Hoppe_1996_ERI/`.

Hugues Hoppe, Tony DeRose, Tom Duchamp, Mark Halstead, Hubert Jin, John McDonald, Jean Schweitzer, and Werner Stuetzle. Piecewise smooth surface reconstruction. *Proceedings of SIGGRAPH 94*, pages 295–302, July 1994.

Tobias Isenberg, Bert Freudenberg, Nick Halper, Stefan Schlechtweg, and Thomas Strothotte. A Developer's Guide to Silhouette Algorithms for Polygonal Models. *IEEE Computer Graphics and Applications*, 23(4):29–37, 2003.

Tobias Isenberg, Nick Halper, and Thomas Strothotte. Stylizing Silhouettes at Interactive Rates: From Silhouette Edges to Silhouette Strokes. In George Drettakis and Hans-Peter Seidel, editors, *Proceedings of Eurographics 2002 (Saarbrücken, Germany, September 2002)*, volume 21, pages 249–258, Oxford, UK, 2002. The Eurographics Association, Blackwell Publishers.

Greg James. Operations for Hardware-Accelerated Procedural Texture Animation. In Mark DeLoura, editor, *Game Programming Gems 2*, pages 497–509. Charles River Media, 2001.

Scott F. Johnston. Mock Media. In *Advanced Renderman: Beyond The Companion*, volume 11 of *SIGGRAPH 1998 Course Notes*, pages 113–121. New York, 1998.

James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150. ACM Press, 1986. ISBN 0-89791-196-2.

Robert D. Kalnins, Lee Markosian, Barbara J. Meier andMichael A. Kowalski, Joseph C. Lee, Philip L. Davidson, Matthew Webb, John F. Hughes, and Adam Finkelstein. WYSIWYG NPR: Drawing Strokes Directly on 3D Models. In *Proceedings of SIGGRAPH 2002 (San Antonio, July 2002)*, pages 755–762, New York, 2002. ACM SIGGRAPH.

Matthew Kaplan, Bruce Gooch, and Elaine Cohen. Interactive Artistic Rendering. In *Proceedings of NPAR 2000, Symposium on Non-Photorealistic Animation and Rendering (Annecy, France, June 2000)*, pages 67–74, New York, 2000. ACM.

John Kessenich, Dave Baldwin, and Randi Rost. The OpenGL Shading Language, Version 1.05, 2003. URL `http://www.3dlabs.com/support/developer/ogl2/`.

Allison W. Klein, Wilmot W. Li, Michael M. Kazhdan, Wagner T. Correa, Adam Finkelstein, and Thomas A. Funkhouser. Non-Photorealistic Virtual Environments. In Kurt Akeley, editor, *Proceedings of SIGGRAPH 2000 (New Orleans, July 2000)*, pages 527–534, New York, 2000. ACM SIGGRAPH.

Michael A. Kowalski, Lee Markosian, J. D. Northrup, Lubomir Bourdev, Ronen Barzel, Loring S. Holden, and John F. Hughes. Art-Based Rendering of Fur, Grass, and Trees. In *Proceedings of SIGGRAPH'99 (Los Angeles, August 1999)*, pages 433–438, New York, 1999. ACM SIGGRAPH.

Adam Lake, Carl Marshall, Mark Harris, and Marc Blackstein. Stylized Rendering Techniques for Scalable Real-Time 3DAnimation. In *Proceedings of NPAR 2000, Symposium on Non-Photorealistic Animation and Rendering (Annecy, France, June 2000)*, pages 13–20, New York, 2000. ACM.

Wolfgang Leister. Computer Generated Copper Plates. 13(1):69–77, March 1994. ISSN 0167-7055.

Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proc. SIGGRAPH 01*, Computer Graphics Proceedings, Annual Conference Series, pages 149–158, 2001.

Babette Ludowici. Ottonische *aula regia* oder unbekannter Kirchenbau? Ein Arbeitsbericht zum Stand der Auswertung der Grabungen von 1959–1968 auf dem Magdeburger Domplatz. *Archäologisches Korrespondenzblatt*, 30(3), 2000.

Bill Mark, Steve Glanville, Kurt Akeley, and Mark Kilgard. Cg: A system for programming graphics hardware in a c-like language. In *Proc. SIGGRAPH 01*, Computer Graphics Proceedings, Annual Conference Series, pages 896–907, 2003.

Lee Markosian. Personal communication, October 2002.

Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-Time Nonphotorealistic Rendering. In Turner Whitted, editor, *Proceedings of SIGGRAPH'97 (Los Angeles, August 1997)*, pages 415–420. ACM SIGGRAPH, 1997.

Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Loring S. Holden, J. D. Northrup, and John F. Hughes. Art-based Rendering with Continouous Levels of Detail. In *Proceedings of NPAR 2000, Symposium on Non-Photorealistic Animation and Rendering (Annecy, France, June 2000)*, pages 59–64, New York, 2000. ACM.

Kodansha Mash-Room. *Akira*, volume 4. Carlsen Verlag GmbH, Hamburg, 5 edition, 1996.

Maic Masuch, Bert Freudenberg, Babette Ludowici, Sebastian Kreiker, and Thomas Strothotte. Virtual Reconstruction of Medieval Architecture. In M. A. Alberti, G. Gallo, and I. Jelinek, editors, *Proceedings of EUROGRAPHICS 1999, Short Papers*, pages 87–90, 1999.

Maic Masuch and Thomas Strothotte, editors. *Virtuelle Zeitreise: Der Computervisualistikraum in der Ausstellung "'Otto der Große, Magdeburg und Europa"'.* Otto-von-Guericke-Universität Magdeburg, Institut für Simulation und Graphik, 2001. ISBN 3980487415. ISBN: 3-9804874-1-5.

Cord Meckseper. Das Palatium Ottos des Großen in Magdeburg. *Burgen und Schlösser*, (27), 1986.

Jason L. Mitchell, Chris Brennan, and Drew Card. Real-Time Image-Space Outlining for Non-Photorealistic Rendering. In *SIGGRAPH 2002 Conference Abstracts and Applications*, page x, New York, 2002. ACM SIGGRAPH.

Hayao Miyazaki. Mononoke hime. Tokuma Shoten, Dentsu, Nippon Television, Studio Ghibli, 1997.

Ernst Nickel. Magdeburg in karolingisch-ottonischer Zeit. *Zeitschrift für Archäologie*, (7), 1973.

E. G. Noik. Encoding presentation emphasis algorithms for graph. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing*, volume 894, pages 428–435. Springer-Verlag, 1994.

J. D. Northrup and Lee Markosian. Artistic Silhouettes: A Hybrid Approach. In *Proceedings of NPAR 2000, Symposium on Non-Photorealistic Animation and Rendering (Annecy, France, June 2000)*, pages 31–37, New York, 2000. ACM.

Tom Nuydens. OpenGL Hardware Registry. `http://www.delphi3d.net/hardware/`, 2003.

Mark Olano, John C. Hart, Wolfgang Heidrich, and Michael McCool. *Real-Time Shading*. A K Peters, Ltd., Natick, Massachusetts, 2002.

Victor Ostromoukhov. Digital Facial Engraving. In *Proceedings of SIGGRAPH'99 (Los Angeles, August 1999)*, pages 417–424, New York, 1999. ACM SIGGRAPH.

Victor Ostromoukhov and Roger D. Hersch. Artistic Screening. In Robert Cook, editor, *Proceedings of SIGGRAPH'95 (Los Angeles, August 1995)*, pages 219–228, New York, 1995. ACM SIGGRAPH.

Victor Ostromoukhov and Roger D. Hersch. Multi-Color and Artistic Dithering. In Alyn Rockwood, editor, *Proceedings of SIGGRAPH'99 (Los Angeles, August 1999)*, pages 425–432, New York, 1999. ACM SIGGRAPH.

Oscar Meruvia Pastor, Bert Freudenberg, and Thomas Strothotte. Real-Time, Animated Stippling. *IEEE Computer Graphics and Applications*, 23(4):62–68, 2003.

Emil Praun, Hughes Hoppe, Matthew Webb, and Adam Finkelstein. Real-Time Hatching. In Eugene Fiume, editor, *Proceedings of SIGGRAPH'2001 (Los Angeles, August 2001)*, pages 581–586, New York, 2001. ACM SIGGRAPH.

Guillaume Provost. Personal communication. Pseudo Interactice Inc., Toronto, Ontario, Canada, September 2003.

Punkt im Raum. Requiem – A Technology Study for the Shark 3D Engine. `http://www.punkt-im-raum.com/eng/projekte_requiem.shtml`, 2000.

Ramesh Raskar and Michael Cohen. Image Precision Silhouette Edges. In Stephen N. Spencer, editor, *Proceedings of the Conference on the 1999 Symposium on interactive 3D Graphics*, pages 135–140, New York, April 1999. ACM Press.

Jarek R. Rossignac and Maarten van Emmerik. Hidden contours on a frame-buffer. In *Proceedings of the 7th Workshop on Computer Graphics Hardware*, 1992.

Takafumi Saito and Tokiichiro Takahashi. Comprehensible Rendering of 3-D Shapes. In Forest Baskett, editor, *Proceedings of SIGGRAPH'90 (Dallas, August 1990)*, pages 197–206, New York, August 1990. ACM SIGGRAPH.

Michael P. Salisbury, Corin Anderson, Dani Lischinski, and David H. Salesin. Scale-Dependent Reproduction of Pen-and-Ink Illustration. In Holly Rushmeier, editor, *Proceedings of SIGGRAPH'96 (New Orleans, August 1996)*, pages 461–468, New York, 1996. ACM SIGGRAPH.

Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive Pen-and-Ink Illustration. In Andrew Glassner, editor, *Proceedings of SIGGRAPH'94 (Orlando, July 1994)*, pages 101–108, New York, 1994. ACM SIGGRAPH.

Simon Schofield. *Non-photorealistic Rendering: A Critical Examination and Proposed System.* PhD thesis, School of Art and Design, Middlesex University, 1994.

Francois Schuiten and Benoit Peeters. *Der Turm.* Carlsen, Stuttgart, 3. edition, 1994.

Tiziano Sclavi and Giovanni Freghieri. *Ananga!* Number 133 in Dylan Dog. Sergio Bonelli Editore, 1997.

Adrian Secord, Wolfgang Heidrich, and Lisa Streit. Fast Primitive Distribution for Illustrations. In Paul Debevec and Simon Gibson, editors, *Proceedings of the Thirteenth Eurographics Workshop on Rendering.* Eurographics Association, 2002.

Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification (Version 1.4). Editor: Jon Leech, 2002. URL `http://www.opengl.org/`.

Peter-Pike Sloan, William Martin, Amy Gooch, and Bruce Gooch. The Lit Sphere: A Model for Capturing NPR Shading from Art. *Proceedings of Graphics Interface 2001 (Ottawa, Canada, June 2001)*, pages 143–150, June 2001.

Spinor GmbH. Renderer and Shader Architecture. *Shark3D engine technology white papers*, 2002. URL `http://www.shark3d.com/goto/technology_render.html`.

Squeak. An idea processor for children of all ages. `http://squeak.org`, 2001.

Christine Strothotte and Thomas Strothotte. *Seeing Between the Pixels. Pictures in Interactive Computer Systems.* Springer Verlag, Berlin · Heidelberg · New York, 1997.

Thomas Strothotte, Matthias Puhle, Maic Masuch, Bert Freudenberg, Sebastian Kreiker, and Babette Ludowici. Visualizing Uncertainty in Virtual Reconstructions. In *Proceedings of Electronic Imaging & the Visual Arts, EVA Europe '99*, page 16, Berlin, 1999. VASARI, GFaI.

Thomas Strothotte and Stefan Schlechtweg. *Non-Photorealistic Computer Graphics. Modeling, Animation, and Rendering.* Morgan Kaufmann Publishers, San Francisco, 2002.

Ubi Soft. XIII. Video game published by Ubi Soft, 2003. URL `http://www.xiii-thegame.com/`.

Robert Ulichney. *Digital Halftoning.* MIT Press, Cambridge, 1987.

Steve Upstill. *The RenderMan Companion: A Programmer's Guide To Realistic Computer Graphics.* Addison Wesley, 1990.

William Vance and Jean van Hamme. *XIII.* Dargaud, 1984.

Oleg Veryovka. Animation with Threshold Textures. In *Proceedings Graphics Interface*, pages 9–16. Calgary, Alberta, May 2002.

Oleg Veryovka and John W. Buchanan. Comprehensive Halftoning of 3D Scenes. In Pere Brunet and Roberto Scopigno, editors, *Proceedings of EuroGraphics'99 (Milano, Italy, September 1999)*, pages 13–22, Oxford, 1999. NCC Blackwell Ltd.

Oleg Veryovka and John W. Buchanan. Texture-based Dither Matrices. 19(1):51–64, 2000.

Wencheng Wang, Yanyun Chen, and Enhua Wu. A new method for polygon edging on shaded surfaces. *Journal of Graphics Tools*, 4(1):1–10, 1999. ISSN 1086-7651.

Matthew Webb, Emil Praun, Adam Finkelstein, and Hugues Hoppe. Fine Tone Control in Hardware Hatching. In *Proceedings of NPAR 2002, International Symposium on Non Photorealistic Animation and Rendering (Annecy, France, June 2002)*, pages 53–58, New York, 2002. ACM Press.

Lance Williams. Pyramidal parametrics. *Proceedings of SIGGRAPH 83*, pages 1–11, 1983.

Georges Winkenbach and David H. Salesin. Computer-Generated Pen-and-Ink Illustration. In Andrew Glassner, editor, *Proceedings of SIGGRAPH'94 (Orlando, July 1994)*, pages 91–100, New York, 1994. ACM SIGGRAPH.

Holger Winnemöller and Shaun Bangay. Rendering Optimisations for Stylised Sketching. In *Proceedings of the 2nd International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, pages 117–122. ACM Press, 2003. ISBN 1-58113-643-9.

Robert C. Zeleznik, Kenneth P. Herndon, and John F. Hughes. Sketch: An Interface for Sketching 3D Scenes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, pages 163–170. ACM Press, 1996. ISBN 0-89791-746-4.

Denis Zorin, Peter Schröder, and Wim Sweldens. Interpolating subdivision for meshes with arbitrary topology. *Proceedings of SIGGRAPH 96*, pages 189–192, August 1996.