2007-03-05

Stylized Hatching for 3D Animation

Trent Fielding Crow
Brigham Young University - Provo

STYLIZED HATCHING FOR 3D ANIMATION

by

Trent F. Crow

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

April 2007

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Trent F. Crow

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

____________________ ____
Date Parris K. Egbert, Chair

____________________ ____
Date R. Brent Adams

____________________ ____
Date Daniel Zappala

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Trent F. Crow in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Parris K. Egbert
Chair, Graduate Committee

Accepted for the
Department

Parris K. Egbert
Graduate Coordinator

Accepted for the
College

Thomas W. Sederberg
Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

STYLIZED HATCHING FOR 3D ANIMATION

Trent F. Crow

Department of Computer Science

Master of Science

Much research has been done in creating non-photorealistic renderings of objects that mimic the look of hand-made drawings by traditional artists. This thesis extends work in this area by presenting an NPR hatching method that can be applied to 3D animated films to help them feel more hand drawn. In contrast to most other NPR methods, this method preserves the 3D lighting and effects of the film that make it interesting to watch. This process includes a procedural algorithm to create a hatching pattern that can be easily integrated into any film's pipeline that uses Renderman. In addition, we create a set of controls to adjust the hatching that are easy to use and allow our style to be applied to many different objects in many shots of a film in an efficient manner. To show the success of our method, we will discuss the implementation and results of applying it to an actual 3D animated short film.

ACKNOWLEDGMENTS

I thank my committee for their time and effort in helping me in preparing this thesis document. They have selflessly given many hours towards helping me to be successful. I thank my fellow classmates and lab colleagues for many questions answered, much advice given, and support rendered in my behalf. I also give thanks to my wife, Eve, for being patient with me and having confidence in my abilities.

Contents

1 Introduction **1**

2 Paper to be Submitted **3**

2.1 Introduction 3

2.2 Related Work 6

2.3 System Design 12

2.3.1 Hatching in the Pipeline 16

2.4 Procedural Algorithm 18

2.4.1 Controlled-density Hatching 18

2.4.2 Stylized Hatching 24

2.5 Artistic Control 34

2.5.1 User Interface Parameters 34

2.5.2 Controlling the Hatching 40

2.6 Results 44

2.6.1 Defining the Hatching Style 45

2.6.2 Hatching with Different Lighting Schemes 48

2.6.3 Hatching Defining Shadows 48

2.6.4 Hatching Integration 48

2.6.5 Future Work 53

3 Conclusion **57**

4 Bibliography **59**

Chapter 1

Introduction

In this thesis, we present a non-photorealistic rendering method to stylize the look of a 3D animated film by adding hatching strokes to the sets and characters of the film. An important aspect of any film is the style in which its imagery is displayed to viewers, called its look and feel. Choosing a suitable style that interests viewers and helps a film accomplish its purpose can be very challenging. A common problem among many computer graphics films is the repetitive use of standard lighting models, shapes, and patterns that makes their styles unimaginative. Several limits exist that make adding a unique look and feel to a film challenging. These limits include the time alloted for making the film, its complexity in terms of sets and characters, and the duration of the film.

To solve this problem, we describe a method of adding a hatching style consistently among all of the shots of a film. The hatching reinforces the lighting that is already present and thus changes dynamically as the lighting changes. Our method has two major benefits over current existing methods. First, our style adds a hand-drawn look to a film while at the same time maintaining the 3D lighting already used in the film. This prevents flattening the look of the film and thus preserves the interesting qualities that make it look 3-dimensional. Second, our method is easily scalable for films that have a large number of shots containing many complex models and different lighting scenarios. The method is flexible enough to allow for the proper balance between time and quality when adding hatching to films of any length.

To accomplish this, our method uses a procedural hatching algorithm that is applied as a surface shader to 3D models. Since the model's surface parameters are used to create the hatching patterns, the algorithm may be applied to any model, regardless of its shape. Also, the procedural nature of the algorithm allows randomness to be added naturally to the patterns and allows them to be easily adjusted. In addition, we provide a process of organizing the surface shaders into groups of objects and shots so as to make the application of the hatching to multiple shots easily manageable. Finally, we provide parameters and tools to allow the user to quickly and efficiently arrive at desirable settings for the hatching and to help the user pre-visualize how changes to the hatching's parameters will affect the objects before they are rendered.

Chapter two of this thesis is in the form of a paper that will be submitted for publication. Following that, Chapter three provides conclusions.

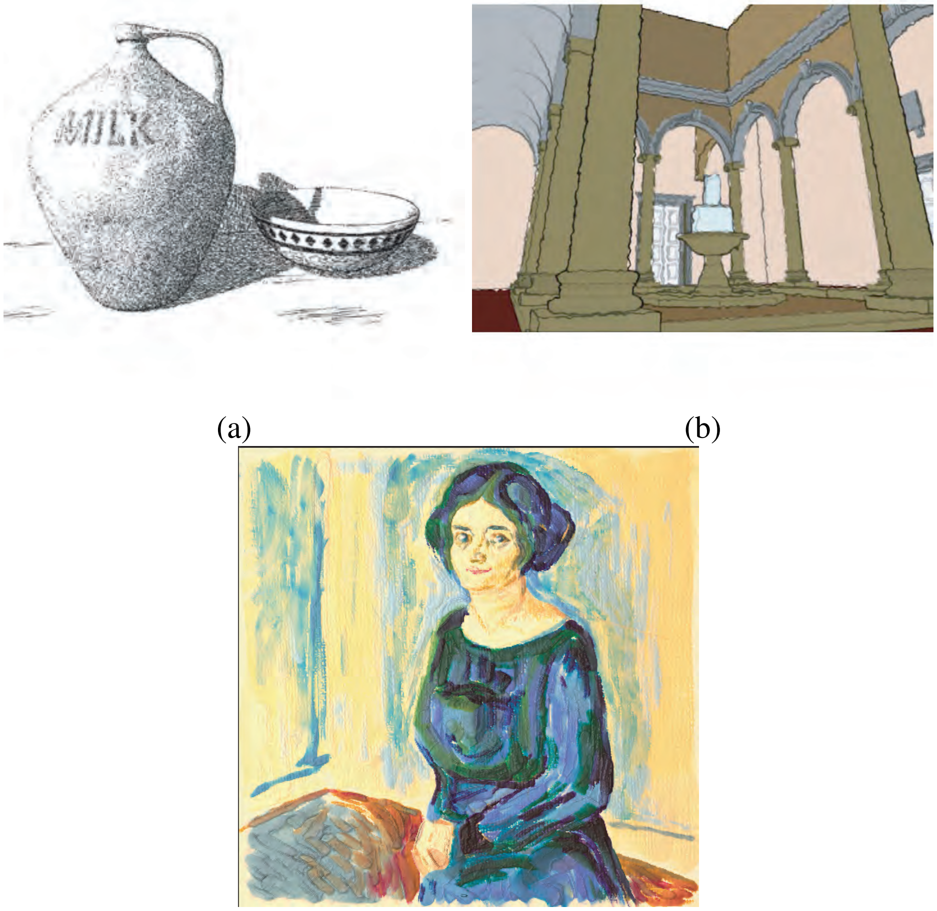
Chapter 2

Paper to be Submitted

2.1 Introduction

The art of making computer images look ever more pleasing, interesting, and exciting is a growing discipline. With increased work in photorealistic rendering, the gap separating real photographs from computer generated scenes is gradually diminishing. The related field of non-photorealistic rendering (NPR) deals with creating stylistic renderings that stress different types of information about a scene, often resembling well known artistic styles in disciplines of traditional, hand-made art. This information can include shape, form, composition, or feelings. Examples of specific art forms replicated include water color, impressionistic paintings, architectural drawings, and pen-and-ink hatching (Fig. 2.1).

NPR styles can be applied to still images, simple animations, and animated films. We use the term ”animated film” in this paper to denote an animation of 3D models that is more complicated than simple movements of an object or camera, but is an animation that contains a story, characters, and environments, like a live-action film. While most NPR papers use still images and simple animations as examples of their techniques, little research has been done in applying these techniques to animated films. This presents a greater challenge than applying hatching to still images or simple animations due to a greater number of elements that are combined into a film, such as the story, characters, and environments. These elements make the animation longer and more visually complex, thus requiring different controls and methods in creating hatching. Furthermore, an animated



(a) (b)

(c)

Figure 2.1: Different types of computer-made NPR renderings: (a) Pen-and-ink hatching from [WS96]. (b) A sketchy drawing from [ND04]. (c) An oil painting from [BWL].

film has a pipeline that consists of various stages of development that must be completed in a reasonable amount of time.

The goal of this work is to present a method of hatching that will stylize the look of an animated film. Hatching is a commonly researched NPR style that uses patterns and arrangements of line strokes to mimic the look of hand-drawn hatches and add interest to renderings by representing tones and textures with simple line strokes. Since traditional hatched drawings are created on 2D media, computer-generated hatched images naturally flatten the look of a 3D scene. This 2D look would not be desirable for a 3D animated film, however. Over the past several years, there has been an increased interest in 3D animations due to the increased realism that they can achieve as compared with 2D animations. Although photorealism is not the intent of most 3D animations, the realism added by 3D effects like lighting, shadows, and simulated phenomenon, entertains viewers because they better associate what they see in the film with real life.

We present a method of adding hatching to an animated film that combines the hatching with the lighting and shading in the film. This will allow an animation to maintain its 3D look while adding a unique style by emphasizing the contrast between the lighter and darker tones of its images. The hatching is part of the shaders of the objects and changes automatically according to how the lighting on the object changes. Various parameters may be used to adjust the style of the hatching and help maintain its consistency among the shots of the film.

Although the applications of our method may include any project that uses 3D modeled geometry, our technique is most useful for animations that contain many different models appearing in various different shots and where rendering in real time is not a concern. Our focus is to allow the user to efficiently apply the hatching and adjust it to look desirable in any type of lighting scenario on most types of objects.

The main contribution of this work is to create a stylized look for an animated film by combining 3D lighting with hatching. This style is unique because the resulting film looks 3D yet has a hand-drawn feel. The 3D aspects of the film are maintained instead of being flattened to a 2D look. Also, the hatching adds interest without becoming too overbearing or distracting.

Another contribution of this work is a Renderman pixel shader algorithm that creates a pen-and-ink style of hatching. Other pixel shader hatching algorithms are usually designed to be executed at real time on graphics hardware and are therefore less applicable to general rendering systems. Our algorithm focuses more on the quality of the resulting hatching rather than the speed at which it will render. Finally, while most hatching algorithms perform lookups into pre-defined stroke textures, our algorithm is completely procedural, allowing aspects of the hatching, such as width, length, density, and rotation, to be changed easily and predictably.

Our final contribution is the set of tools we provide for a user to apply the hatching to the shots of an animated film in a consistent and efficient manner. These tools include sliders to vary how the hatching parameters correspond with the shot lighting, options that help adjust the hatching to special cases, and controls to help speed up the process of finding optimal settings for the hatching. The sliders are used to remap the extremes of a parameter, such as the hatching width or frequency, to correspond to particular values of light. This allows the parameter to automatically change as the value of light changes on the object. The special case options are features that help adjust the hatching in situations where the hatching is difficult to use, such as on really small objects. Finally, special controls, such as ”hatching lights” and a color-coded view of the hatching, allow the user to arrive more quickly at the desired settings for the hatching in a particular shot. These tools are specifically designed to help the iterative process of adjusting the hatching to be quicker and more acurate.

The remainder of the paper proceeds as follows. Section 2 discusses previous research in hatching methods and the work most closely associated with the methods used in this paper. In section 3 we explain the overall design of our system and how we incorporate our stroke method into the workflow pipeline of a film. Section 4 describes the procedural hatching algorithm. Section 5 describes the tools we use to apply and control the hatching. Finally, in section 6 we present our final results and ideas for future work.

2.2 Related Work

Non-photorealistic rendering has attracted much attention in graphics research. Just as there are many different types of artistic styles, there have been a myriad of NPR techniques that have sought to mimic these artistic styles. This large spectrum of styles includes creating sketchy or blue-print like drawings [ND04], taking colors from paintings and applying them to photos [CSUN05], creating line drawings from 3D models [DFR04], and interactively painting scenes with styles similar to oil based paints and acrylics [BWL]. Bruce and Amy Gooch's book on non-photorealistic rendering is a reference for other work involving NPR [GG01]. Since our work is a method of hatching for animations, we will emphasize in this section work that has been done with hatching and other stroke-based methods, as well as methods of animating various NPR styles.

One of the earliest stroke based methods in NPR was done by Appel et. al.[ARS79], in which they discuss how to "halo" a line by making it appear to pass behind another line. At this time NPR was a very new field, and most work done was in building the fundamental algorithms that would be used in later works. Dooley, et al. present a method of creating lines from 3D model data [DC90]. They discuss several qualities of lines, including thickness and style, and how to interactively customize the attributes to create more effective illustrations.

Stroke textures [WS94] are images of strokes that are designated with priorities. The type and tone of hatching desired at a particular point on a model determines the texture to use, which is then projected back onto the model. Stroke textures are a powerful method and have been used in many stroke based methods, including [SABS94],[WS96],[PHWF01],[BW03b], and [BW03a].

One approach that is used frequently in NPR systems is an image based approach. In [SABS94], an image is annotated by the user with certain key strokes. Stroke textures are then chosen based on the input strokes and drawn automatically to create a new image that has the look of a pen and ink illustration. In [SALS96], Salisbury et al. assign a stroke texture to an input gray-scale image and create different scaled versions of the image while maintaining a consistent tonal value across all of them. Another image based method can be found in [Her98], where a photograph is used as input and then an image of paint strokes is built layer by layer, with each layer having progressively smaller strokes.

Instead of using an image based approach, our method's form of input is object-based, using a 3D model as input. In recent years with the increasing data processing power of computers, this form of input has been used more often than image based approaches. One of the most influential object-based papers was done by Winkenbach and Salesin [WS96]. This paper extended the stroke texture method they had previously introduced in [WS94] to work on parametric surfaces. When dealing with curved surfaces, they apply a method called "controlled-density hatching" to the stroke textures that orients the strokes along the surface and gradually makes them appear or disappear as the stroke extends across the surface. This helps maintain a consistent tone across the object. In [VB99], half-toned images are automatically created by applying halftoning textures to 3D models and using ordered dithering and error diffusion to help control the placement and look of the halftoning. Other object based methods are [HZ00], which uses a direction field for computing hatching on smooth surfaces that are not parameterized, and [WM04], which uses a hybrid between image and object based approaches to more clearly represent strokes in complex, detailed regions by approximating the surface properties of the local regions.

In recent years, many researchers have focused on animating NPR techniques. Animated NPR techniques may be off-line or real time. One of the first off-line animation techniques was done by Meier [Mei96] on animated brush stroke styles. This work addresses the problem of temporal coherence with brush strokes. Temporal coherence refers to how well an effect transitions from frame to frame in an animation. If a technique fails to properly address temporal coherence, then either a shower-door effect is created where the objects appear to swim underneath their textures, or the textures appear in random places from frame to frame. Both of these results are distracting for viewers to watch in an animation. Meier solves this problem by modeling surfaces as 3D particle sets and rendering them as 2D brush strokes in screen space. Another off-line animation technique was developed by Curtis [Cur98]. This technique creates a sketchy animation style by rendering a depth of the objects and then a vector field of particles that determine the directions of how to perturb the silhouette. In [DS00], depth information is used as well in creating a pen-and-ink illustration style for trees that addresses how to draw strokes in complex areas while preventing the resulting illustration from appearing too busy. Abstract primitives are chosen to represent the trees, such as planes or disks, depth continuities are determined to decide what part of the primitives to draw, and then a hybrid pixel based and analytical based algorithm is used to handle the complex geometric data.

More common than off-line animation techniques are real-time NPR techniques. One of the earliest of these techniques was presented in [MKT$^+$97] where a tool was proposed to improve the descriptions of lines and visible surface algorithms to create various NPR styles that animate at interactive rates. These styles were mostly stroke based silhouette styles that used straight and sketchy outlines, as well as hatching-type strokes that appeared near the borders of the outlines. In [LMHB00], the quality of real time NPR was improved with a technique that allows more than just silhouette NPR styles to animate at real-time, including fully hatched objects and toon-shaded style renderings. This technique allows objects to scale while maintaining stroke coherence in image space, instead of the more commonly used object space. During the following few years, the most effective real time stroke based techniques were based on the use of stroke textures and various improvements to the basic technique. In both [PHWF01] and [Fre01], layers of mip-mapped stroke textures, called tonal art maps, were used to create fast and convincing hatching styles, with the former being a more complex and true pen-and-ink style, while the latter was a simpler design yet not as CPU intensive. A variation of stroke textures was introduced by [Ver02], where threshold textures were used to add discrete colors. In this system, less bluring occurred at intermediate mip-map levels, although the hatching itself suffered from a simpler, more mechanical look than in [PHWF01].

Buchin et al. authored two variations of the stroke texture methods in [BW03a] and [BW03b]. In the former, they experiment with other variations, such as changing brush stroke patterns to create a pointillism effect and adding lighting variation to hand drawn illustrations. In the latter, their method allows the style of strokes to change interactively by encoding lookups into stroke textures.

In addition to the systems cited above, two other stroke texture based approaches were created during the same time period. [WB03] presents optimizations to stroke textures involving uncertainty functions and layered stroke textures. [FV03] expands the use of the stroke textures from [PHWF01] to encompass arbitrary textures, such as brick and rock textures. Finally, one of the most recent stroke texture techniques, presented in [FMS04], uses layered hand-drawn halftoned textures to create a nice stroke effect for a real time game engine.

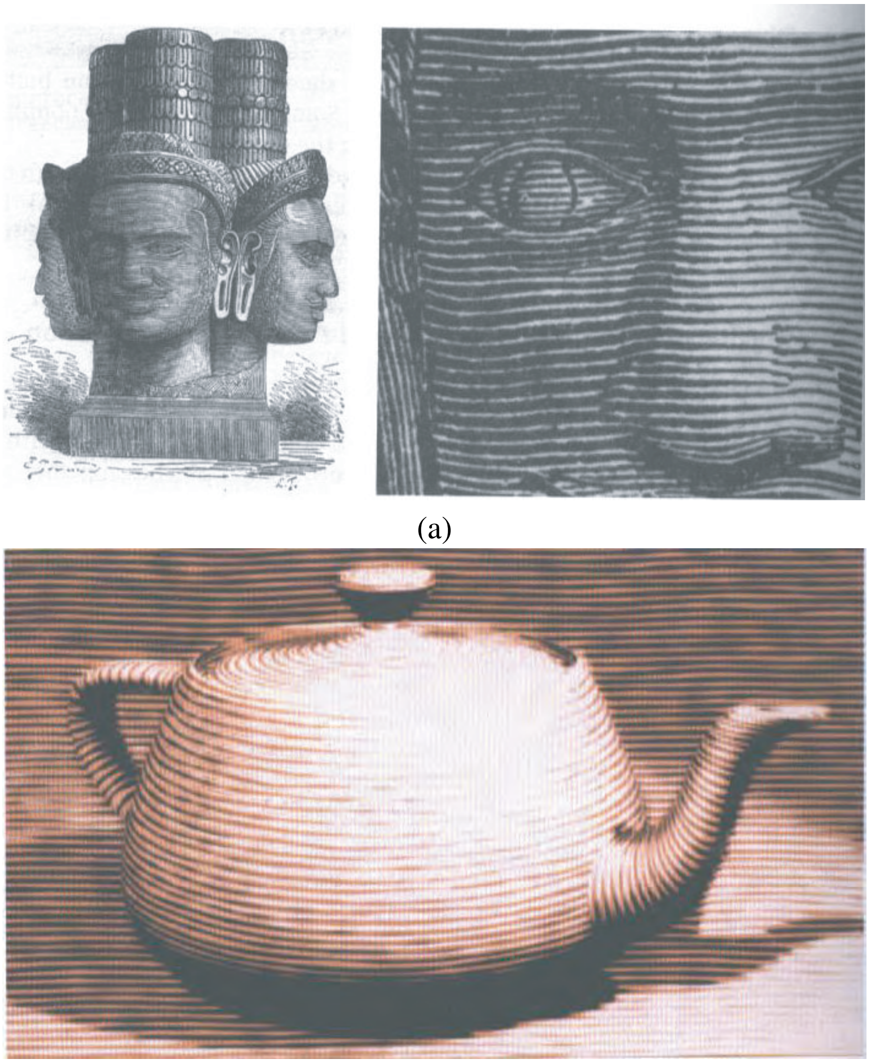
A pitfall of current real time stroke techniques is the quality of the final renderings. Real time techniques offer much in terms of speed and interactivity, but as a result the quality of the final animations is more comparable to video game quality graphics than movie quality renderings. We chose to implement an off-line animation technique so that the final quality of our hatching technique would be comparable to the quality of current professional 3D animated films and so the hatching would not be distracting to view for the duration of a film.

The most recent developments in animated NPR techniques have strayed away from stroke textures into other areas with different styles and approaches. A method for drawing suggestive contours is shown in [DFR04], where changes in viewpoint are analyzed and contours are rendered at real-time rates. A different type of contour technique that takes user-supplied surface and anchor strokes and draws essential lines for character faces is demonstrated in [LD04]. Other real time techniques include a method for depicting NPR motion in games [HHD04], a method for synthesizing painterly, sketchy, and cartoon styles from video [CRH05], and a techique that describes how to create animations of mosaics while preserving temporal coherence [SLK05]. Although these techniques are effective in their respective applications, they are not applicable to techniques that create stroke styles on 3D models.

The current work in stroke based NPR methods for animations is not suitable for an animated film for several reasons. First, these styles are not visually congruent with the purposes of a 3D animated film. As opposed to a 2D style animated film, a 3D animated film looks closer to real life and is more perceptually rewarding. Since NPR techniques are typically based on a traditional drawing or art form, the results are often flat looking. This is acceptable since NPR focuses on conveying different types of information besides what realism is able to offer. When applying an NPR style to an animated film, however, the 3D lighting and effects that define the 3D animation genre are lost, and the film appears 2D. Our objective, therefore, is to present a style of NPR that both conveys a simple, hand-made feel that is typical from stroke styles, and that also preserves the 3D lighting style that characterizes 3D films.

Another reason why current stroke techniques are not suitable for animated film is the lack of tools needed to effectively apply strokes to the shots of a film in an efficient manner. The demonstrations for other techniques are usually short and consist of a simple tranformation of the object or of the camera in the scene. Applying these techniques to a film that is at least several minutes in length would be difficult unless proper tools existed to help expedite the process of applying the method and adjusting the appropriate settings. In our method we offer tools that make finding desirable settings for the hatching of many different shots in a film more practical.

In their advanced tutorial on Renderman, Apodaca and Gritz present a stroke based surface shader that produces a look similar to wood block engravings [AG00] (See Fig. 2.2). Their techniques in drawing strokes with consistent widths across any object are based on ideas introduced by Winkenbach and Salesin in their paper on rendering hatching on parametric surfaces [WS96]. We base our procedural technique on the shader by Apodaca and Gritz and enhance it by creating strokes that look closer to a pen-and-ink hatching style and whose parameters make it very flexible for a variety of models and lighting conditions.



(a)

(b)

Figure 2.2: (a) An actual wood block engraving of Brahma head from Temple Phnom-Boc (shown in [AG00]). (b) A teapot model shaded with a wood block surface shader.

2.3 System Design

The contribution of this work is a unique NPR style that combines hatching with the lighting of a film. The inspiration behind this style started with our desire to create a style with two major properties, a style which felt hand-drawn yet looked 3-dimensional. We wanted a hand-drawn feeling that is similar to the style seen in field book or botanical illustrations. Fig. 2.3 shows examples of such illustrations. There are several characteristics from these images that help convey a hand-drawn feel:

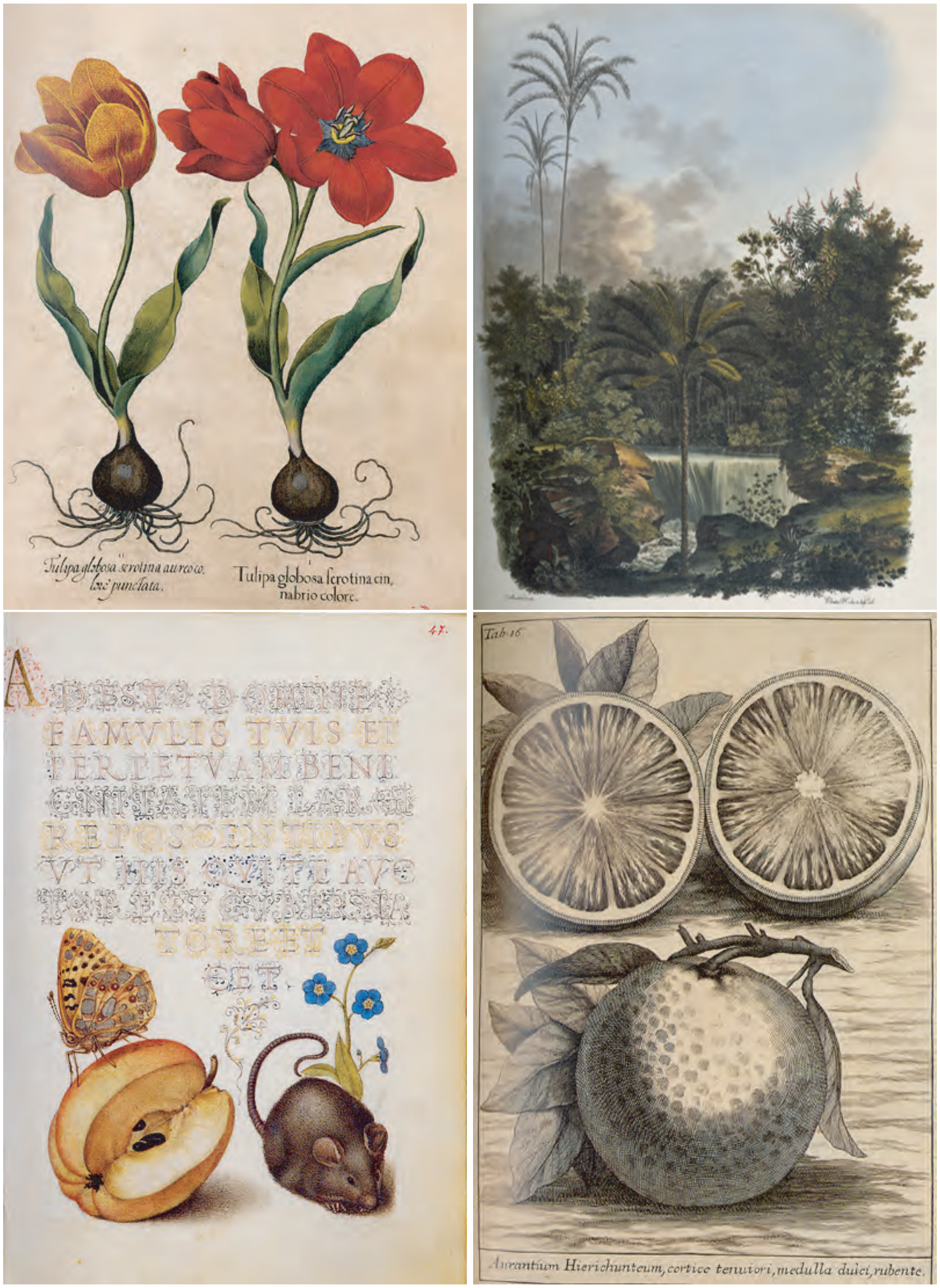

Tulipa globosa serotina aureo co, lore punctata.
Tulipa globosa ſerotina cin, nabrio colore.
47.
FAMVLIS TVIS ET
PERPETVAM BENI
Tab. 16
Aurantium Hierichunteum, cortice tenuiori, medulla dulci, rubente.

Figure 2.3: Examples of images that inspired the look and feel for the short film, Noggin. While we wanted a similar authentic, hand-drawn feel to our rendering style, we focused on exaggerating the look of the hatching so that it would have a greater effect on the tones in the final images.

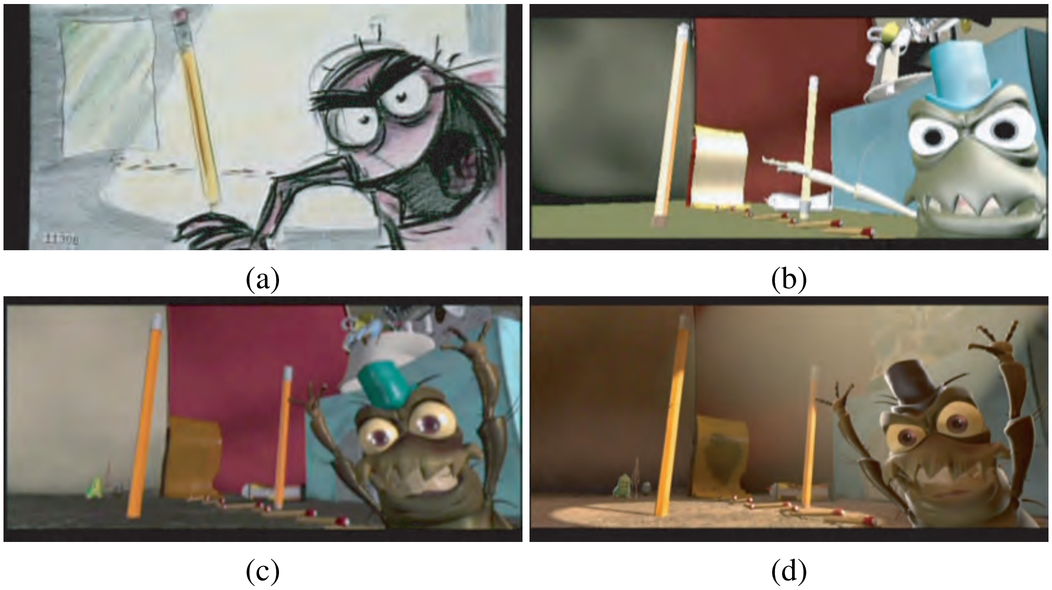

(a)
(b)
(c)
(d)

Figure 2.4: A shot from a 3D animated film at different stages of its pipeline (from *A Bug's Life*, Pixar 1998). (a) Conceptual Art - artists hand draw the key story points of the shot. (b) Editorial - 3D models are built with default colors and lighting and are roughly placed and moved in the shot. (c) Animation - final animation is created for the characters and cameras. (d) Shading and Lighting - final shaders are applied to objects and final lighting is set up for the shot.

1. A color scheme with simple textures and a small palette of saturated colors.
2. The colors are more faded on objects that appear far away.
3. A form of stroking that accents the feeling of being hand-made.

Although we desired our style to have these characteristics, the second major property of our style was that it look 3-dimensional. To fulfill this property, we have designed our method to be used within a typical production pipeline for a 3D animated film. The production pipeline of a film is the series of stages the film must pass through from the time the idea is conceived until the film is completely finished. These stages usually include tasks such as pre-visualization, modeling, animation, shading, lighting, and rendering. Fig. 2.4 shows an example of how a scene looks as it passes through each of the major stages of a film's pipeline. Since our method affects the look and feel of a film, it will be incorporated into the shading and lighting stages of its pipeline.

To fulfill both our hand-drawn and 3D properties, we restate the above characteristics as the design goals of our system:

1. An illustrated-looking hatching style - We sought to create a hatching style that is more prominent than what is noticed in the images of Fig. 2.3. By exaggerating the hatching effect, we emphasize the hand-drawn look that is characteristic of work done with hatching.
2. A color scheme with simple textures and flexible 3D lighting - Since our method is targeted for 3D animated films, our style needs to allow the use of a normal color palette typically used for the shading, lighting, and other effects in such films, as opposed to a monotonic style that is often used in hatching methods. Texture designs are kept simple so as not to conflict visually with the hatching. Because of the nature of some objects or effects, hatching may be altered or even excluded from certain objects.
3. Fading of colors on distant objects - The farther objects are from the camera, the more faded their colors should be. Since the direction of the viewer's eye is a crucial aspect of film-making, the fading controls must be flexible to allow the adjustment of individual objects.

Of these goals, the illustrated hatching style is the most complex in terms of design and execution and will therefore receive the most focus during the rest of the paper. The color scheme and fading goals are mostly addressed in this section and section 5 where we explain the design of our system and the tools for controlling the look of our hatching style. In the rest of this section, we describe how our system is organized to accomplish the above design goals by looking at how our method fits into a film's pipeline.

2.3.1 Hatching in the Pipeline

To work effectively in a film's pipeline, the workflow of our system must be scalable for films with any number of objects and shots. The major focus of our workflow is to ensure the hatching is consistent among objects and shots and that it can be properly adjusted on each object in an efficient manner. Here we describe the setup for how our method is accessed by each object in a film.

In a typical pipeline, the surface colors of objects are defined by surface shading programs, or shaders, that are connected to the objects of a scene and are executed when the scene is rendered. The most widely used system for rendering computer graphics for film quality productions is Pixar's Renderman. We therefore implemented our method as a shading system for Renderman. This allows our system to be flexible for many different rendering systems. It also allows us to take advantage of the useful features of the Renderman language. The most important feature is its use of the Reyes architecture as its core rendering algorithm, which uses a geometric pipeline that dices primitives and organizes pixels into buckets to help use memory as efficiently as possible. This system also allows us to filter the hatching to help prevent aliasing artifacts such as dot crawl and shimmering, as well as making the hatching look better when motion blur is applied.

In Renderman, shaders that define the color of the surface of an object are called surface shaders. Rather than being a specific surface shader, our method needs to apply our style to all of the surface shaders created for all of the objects of the film. Thus, we created a global surface shader that contains various features needed for our system to work effectively. This global shader is essentially a node that receives an object's surface texture as an input. The input texture is the basic color of the object before any lighting calculations are made. An option is set on the global shader to determine which of the film's lighting models are applied to the object (i.e. plastic, metal, marble, etc.). The global shader also contains the other parameters and controls used to adjust the hatching. These parameters will be discussed in section 5. In addition to hatching controls, there are other parameters as well, such as a ramp for specifying the distance at which an object is faded and by what amount.

While the global shader allows us to have individual control for each object, it can be very tedious and even impractical to have to set its parameters for every object in the film. For this reason, we put objects with like properties into shading groups and create default template global shaders for each group. The shots of a film are grouped into different scenes. Each scene has its own group of surface shaders called a palette. A template shader is then created for each group of objects in the scene's palette. The settings in the template only have to be set once for all of the objects in a shading group. If a particular object of a group requires unique settings, then the template is copied and only the settings that are different need to be specified. Likewise, each shot has its own palette of template shaders that are set only if an object requires special settings for that shot. In this manner, settings may be shared among objects of a group and among shots of a scene to save on setup time, while settings may still be specified for individual objects in individual shots to allow for the necessary level of control.

Although the parameters of a typical surface shader are finalized during the shading stage, most of the changes made to our global shader are done during the lighting stage. The hatching algorithm is a pattern that is part of an object's surface texture, but it also responds to the lighting model of the object. We explain in detail in the following two sections how the hatching algorithm works and how it it controlled by its parameters, including how it takes into account the amount of light hitting the object. Due to this dependency on light, the hatching parameters need to be fine-tuned on a shot per shot basis as the user sets up the lights for each shot. Other aspects of our style may also need to be set during this stage, including specifying objects that should not use hatching or changing hatching to stippling for smaller objects.

2.4 Procedural Algorithm

The most significant part of our method is the procedural algorithm that actually draws the stroking pattern. When an object is ready to be rendered, Renderman dices it into micropolygons and executes the instructions of the shader for each pixel of the object that appears in the final image. The algorithm creates a stroked pattern based entirely on the surface parameterization of the object and the input parameters of the shader and does not depend on the input of separate stroke images. The main advantage of this procedural approach is the ease of adapting to changes in the style of the pattern. A film is in a constant state of change until it is finished. As the pattern of the desired hatching changes, no separate images need to be recreated or changed, only the input parameters of the shader need to be adjusted.

2.4.1 Controlled-density Hatching

One of the keys to successfully creating an NPR render that looks like a particular style of art is to carefully preserve the main aspects of that style. One of the most fundamental aspects of a pen-and-ink hatched drawing is the consistency to the width of the pen strokes. Since the pen of the artist never changes in width as it is used throughout a drawing, varying tonal values are achieved by changing the number of hatches in a given area instead of changing the width of the hatches. While trivial in a standard drawing, the task is more complicated when using the surface parameters of 3D objects. For example, imagine the pattern on the surface of an object consists of several straight lines. As the object moves towards the camera, the stripes will grow in apparent size. In a hatched drawing, however, the lines of all objects should maintain a constant width. Instead of growing in size as the object moves closer to the camera, more stripes should be drawn on the object in order to maintain a consistent tonal value.

To solve this problem, Winkenbach and Salesin [WS96] introduced a method called controlled-density hatching. Instead of using the surface parameters of an object in its own object space to modulate the hatching, these parameters are instead transformed into screen space. Because the dimensions of a rendered image do not change, the hatches are based on consistent intervals that are constant as an object moves in 3D space.

Apodaca and Gritz applied the concept of controlled-density hatching from Winkenbach's paper to a Renderman shader [AG00]. This shader creates a form of hatched lines that resemble the engravings of a woodblock print. Fig. 2.2(b) shows an example of this shader applied to a teapot. Although the look of the shader is different than our target look, it provides a framework for doing controlled-density hatching in our own algorithm.

We start by explaining how a typical striping pattern would be approached in a shader and then show what is needed to control the density of the lines. We summarize the explanation given in [AG00], and refer the reader there for a more detailed coverage of the material. Our explanation provides a basic overview that helps in understanding the alterations in their work needed to create our method.

To create a striped pattern, our end goal is to create a continuous pattern that alternates between black and white. We do this by first recognizing that any point on the surface of an object can be uniquely described by a set of two surface parameters, *u* and *v*, that run horizontally and vertically, respectively. We choose to use *u* for our examples, though the same steps would be followed for *v*. Since this parameter's range is from 0 to 1, we can represent a smooth gradient across a surface by using 0 for black and 1 for white. To modify this into a repeatable pattern, the mod function can be used:

```
sawtooth = mod(u * frequency, 1)
```

The frequency corresponds to the number of regions into which the surface is divided. By evaluating the modulus 1 of *u* multiplied by the frequency, the result is *sawtooth*, which linearly increases from 0 to 1 over each region (see Fig. 2.5(a)). We create a stiped pattern by applying two more functions, as shown in the following lines:

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

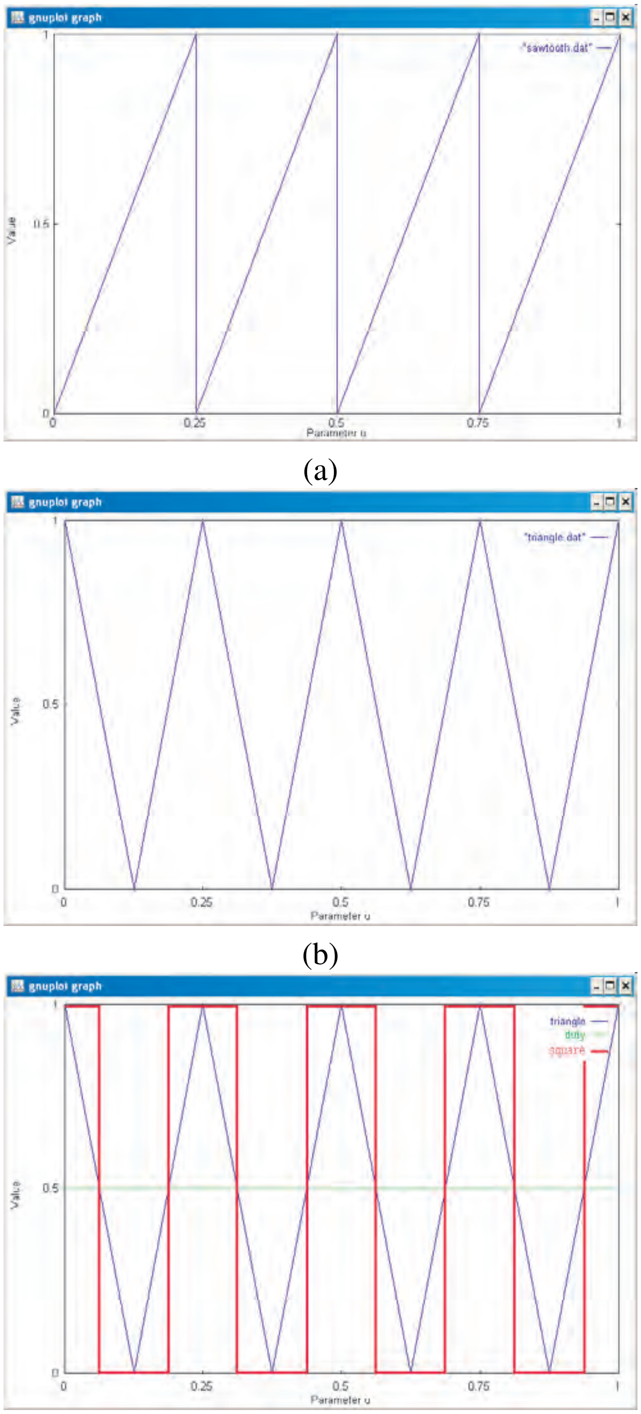
```
square = step(duty, triangle)
```

Feeding *sawtooth* into an absolute value function creates the *triangle* value, which is a series of ”uphills” and ”downhills” from 1 to 0 and back to 1 in each ”frequency” region (see Fig. 2.5(b)). Then we use *duty* variable as a constant valued threshold that intersects *triangle*. The step function returns 0 if *triangle* is less than duty, and 1 if it is greater than or equal to it (see Fig. 2.5(c)). By using our mappings of 0 to black and 1 to white as we stated before, the step function results in a striped pattern, as seen in Fig. 2.6. A low value of *duty* creates a dense patten of lines, while a higher value thins it out.

As seen in Fig. 2.6, the convergence of lines in the pinch-point areas of the sphere create a sense of darker tone in the middle of the sphere where the lines are wider. This is undesirable since we desire a consistent tone that is independent of the specific curvature of an object. This is solved by using a measure of how quickly u changes at any point across the surface to modulate the frequency of the lines. The Renderman Shader Language, or RSL, provides several built-in derivatives, including one called *dPdu* that is the derivative of a 3D point in space, P, with respect to u. Because this gives a measure of how stretched or pinched a surface is, *dPdu* is used to double the frequency as length(*dPdu*) doubles in value:

```
logdp = log(length(dPdu), 2)
ilogdp = floor(logdp)
stripes = pow(2, ilogdp)
sawtooth = mod(u * stripes * frequency, 1)
```

By using a log function, *ilogdp* will increase by one every time length(*dPdu*) doubles, therefore doubling the value of *stripes*. It is then used as an additional multiplier in the previous mod function to influence the frequency of the striped pattern (Fig. 2.7(a)). The discrete jumps in frequency are an obvious distraction, so the difference between *logdp*



(a)

(b)

(c)

Figure 2.5: (a) A graph of the *sawtooth* value. This results in a series of gradients that range from 0 to 1. (b) A graph of the *triangle* value, which uses absolute value to create symmetric ”spikes” in each gradient region. (c) A graph showing how the *duty* value is used with *triangle* to create a step function. The blue line is the graph of *triangle* from (b). The green line is *duty*, which is a threshold used to determine how large the striped regions will be. When *triangle* dips below *duty*, the result is 0; when *triangle* rises above *duty*, the result is 1. This step function, or *square*, is shown by the red line. The white stripes in Fig. 2.6 appear whenever *square* equals 1.

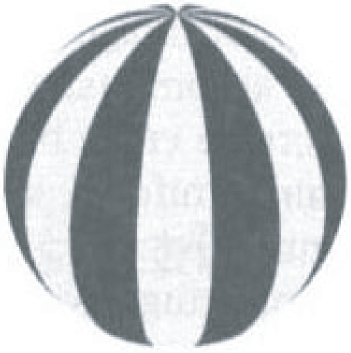
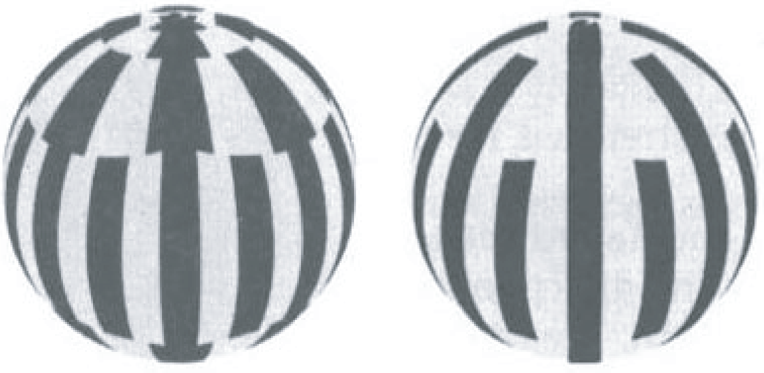


Figure 2.6: Simple striped pattern created with the mod and step functions.

and *ilogdp* is used to modulate *duty* in the step function to create lines of consistent width (Fig. 2.7(b)):



(a) (b)

Figure 2.7: (a) A step towards uniform line density. The frequency of the lines double as dPdu doubles, or as the lines pass through regions of greater surface area. (b) The difference between logdp and ilogdp, called *transition*, is used in the step function to modulate the frequency inbetween discrete frequency jumps. The resulting value *square* gives us lines of consistent width:

```
transition=logdp-ilogdp

square=step(duty*(1-transition/2), triangle)
```

The final problem that remains to control the density of the lines is that as the object recedes from the camera, the apparent width of the lines will change. To prevent this, the derivative needs to be modified to be relative to screen space. In RSL, a new derivative can be created by transforming *dPdu* into screen space and then scaling it to a desired level:

```
dp=dpscale*vtransform ("screen", dPdu)
```

The *dp* variable then replaces *dPdu* in the above log function. To taper the ends of the hatches, the use of *transition* by the step function is altered so that new hatches are more gradually introduced in transition areas:

```
transtriangle=abs((1+transition)*triangle-transition)
square=step(duty, transtriangle)
```

As shown in Fig. 2.8, this creates a pattern whose lines maintain a consistent tone across the whole object. The code listing in Fig. 2.9 shows the combined code used to generate this pattern. As the lines on the object approach sections of smaller area, existing lines gradually thin and then disappear. Likewise, as the lines approach sections of larger area, new lines gradually appear. The same happens as the apparent size of the object on the screen changes. New lines appear as the object comes closer to the view, and existing lines disappear as it moves farther away.

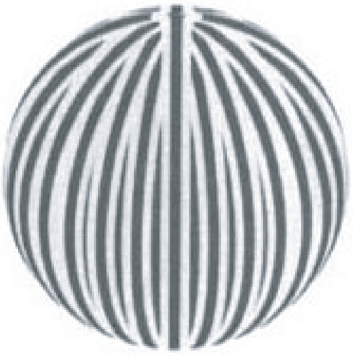


Figure 2.8: The amount of screen space that an object takes up is used to further modulate the frequency of the hatching. A more consistent tone is maintained than in Fig. 2.7.

Another important concept that Apodaca and Gritz present is modulating the width of the lines based on the value of light that hits the surface of the object. To accomplish

```
dp = dpscale * vtransform ("screen", dPdu)

logdp = log(length(dPdu), 2)
ilogdp = floor(logdp)
stripes = pow(2, ilogdp)

sawtooth = mod(u * stripes * frequency, 1)
triangle = abs(2.0 * sawtooth - 1.0)
transition = logdp - ilogdp
transtriangle = abs((1 + transition) * triangle - transition)

square = step(duty, transtriangle)
final_color = square
```

Figure 2.9: The combined code that produces controlled-density hatching, such as in Fig. 2.8.

this, the light contribution at the current pixel is converted to grayscale. By multiplying this number by *duty*, the lines gradually disappear as they approach areas of increased light. As shown in the next section, we use the light contribution to modulate many different aspects of the hatching in order to simulate changing tonal values.

2.4.2 Stylized Hatching

One of the goal properties of our style that we mentioned in the last section was a hand-drawn look. Although the woodblock shader provides an important basis for drawing strokes on surfaces, its strokes are too uniform to be interpreted as hand-made. Fig. 2.10 shows an example of a hatched image from [PHWF01] that better conveys a hand-made look. By analyzing results of recent stroking methods that share similar hand-sketched looks, we observed three characteristics common to these drawings:

1. smaller, individual strokes instead of long, continuous ones

2. strokes that overlap each other
3. strokes that contain randomness in placement, shape, and alignment

The challenge in incorporating these characteristics into our algorithm was to find solutions that would allow us to maintain controlled-density and light-modulated hatching.

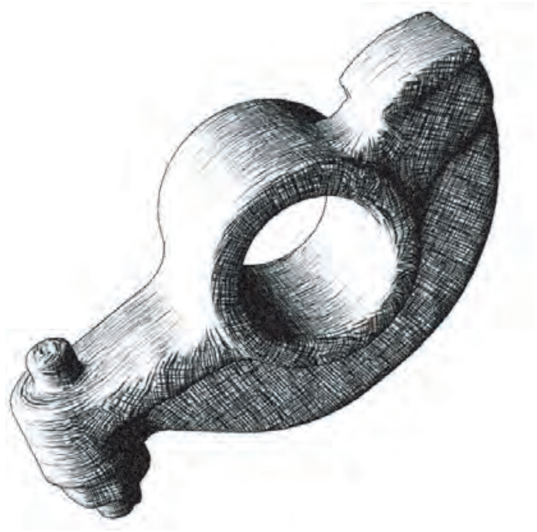
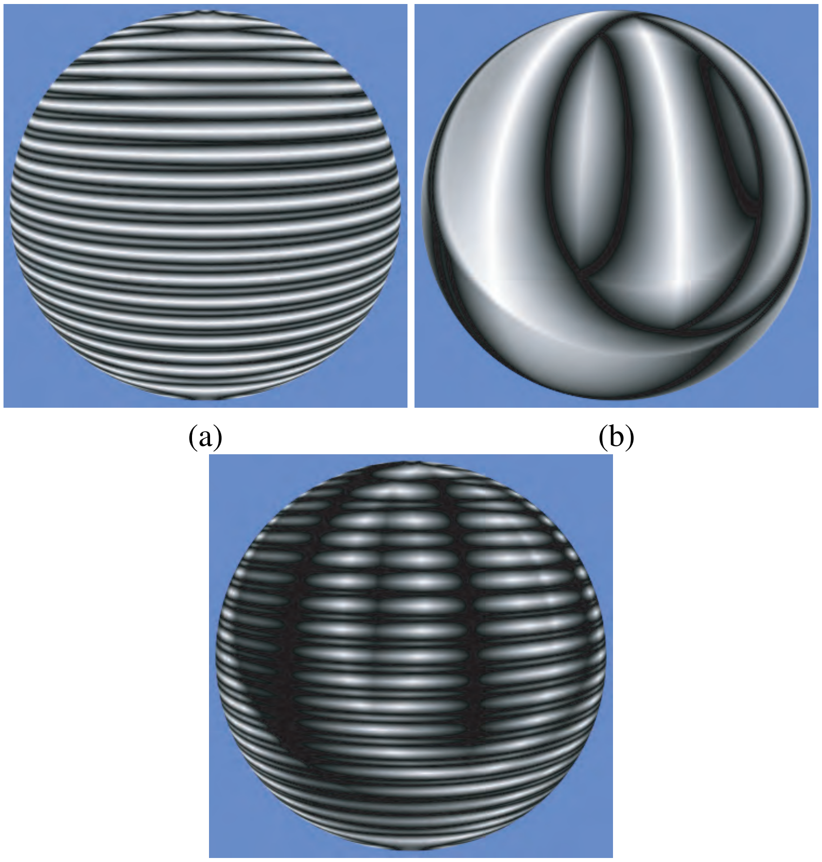


Figure 2.10: An example of hatching that is more characteristic of the type of hatching we wanted to achieve than the wood block shader's hatching. Several characteristics of this hatching make it more interesting and more effective in creating tonal shifts, such as smaller strokes of hatching and overlapping strokes.

Individual strokes Using more shorter strokes instead of fewer continuous strokes adds a more sketchy look to the hatching. It also allows more flexibility in randomizing various visual attributes of the hatches. To do this, we must divide the hatches into columns, similar to how we created rows of hatching in the woodblock shader. Similar to controlling the widths of the hatching, we need to preserve the length of the hatches as the object moves about in 3D space. Therefore, we must calculate another *transtriangle* parameter, this time in the opposite direction (the v direction). We call this value *transtriangle*2 and show the results of mapping this value to a sphere in Fig. 2.11. By taking the product of *transtriangle* and *transtriangle*2, we create a grid-like pattern of hatches.

From close inspection of Fig. 2.11(c), it is obvious that the hatches are diamond shaped. To round out the hatches, we need a function that will take a linear increase and



(a) (b)

(c)

Figure 2.11: (a) The *transtriangle* value mapped to a sphere. (b) *transtriangle*2 is similar to *transtriagle*, but goes in the opposite direction with a different frequency. (c) The product of *transtriangle* and the inverse of *transtriangle*2 create a pattern that is the basis for creating multiple hatches per row.

output a curve that is flat in the middle and that gradually falls off at the ends. In [EMP+03], a type of logarithmic function, called a bias function, is described as follows:

$$bias(b,t) = t^{ln(b)/ln(0.5)} \tag{2.1}$$

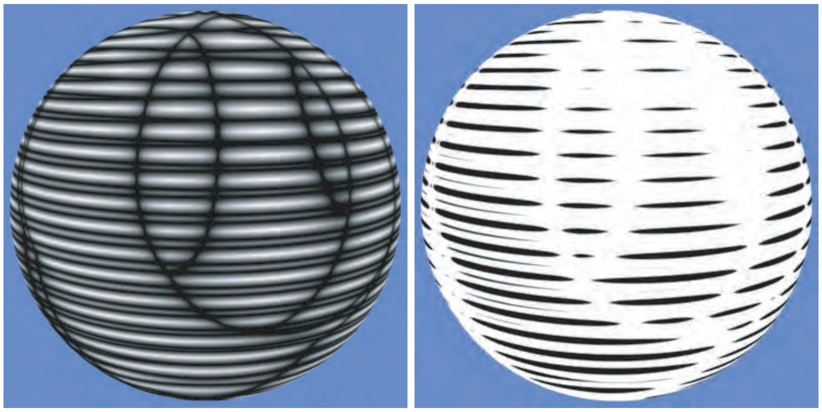
As t varies over the interval [0,1], the bias function follows these rules:

1. bias(b,0) = 0
2. bias(b,.5) = b
3. bias(b,1) = 1

By using the inverse of *transtriangle*2 as input into a bias function and then using the result as input into another bias function, both ends of the ”diamond” are smoothed. The new gradient is produced by inverting the second bias result and multipying it by *transtriangle* (Fig. 2.12(a)):

$$bias(.1, bias(.5, 1 - transtriangle2))) \tag{2.2}$$

We apply a smoothstep to the gradient to create our final hatch shape (Fig. 2.12(b)). In this example the size of the hatches has been increased higher than normal for explanatory purposes. To test the preservation of the widths and lengths of the hatching, we compare Fig. 2.12(b) with additional images of a sphere that are at two locations further from the camera. Notice that the widths of the hatches stay consistent (Fig. 2.13).



(a)

(b)

Figure 2.12: (a) By using bias, we have changed the hatching gradient from a diamond shape to a more rounded one. (b) A smoothstep applied to the gradient gives us defined hatches.

Overlapping strokes An artist creates varying tones in a hatched illustration by drawing overlapping layers. The human hand naturally crosses lines together instead of keeping them separated. The easiest way to mimic this look in a surface shader is by creating more than one copy of the hatching, vary each copy differently, and then layer them together. In RSL, the mix function is used to put one color on top of another color according to a given pattern:

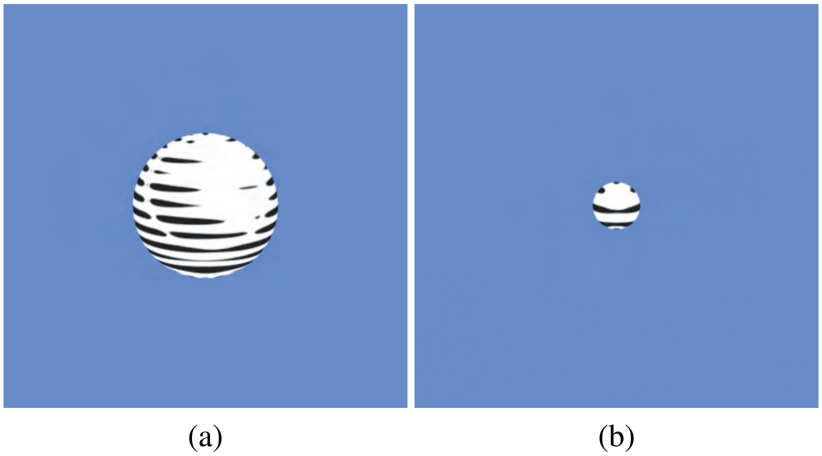


Figure 2.13: As the sphere from Fig. 2.12(b) above recedes from the camera in (a) and then even further in (b), the hatching maintains a consistent width.

$$mixedColor = mix(colora, colorb, blend) \tag{2.3}$$

where *a* is the color of layer 1, *b* is the color of layer 2, and *blend* is a float value in the range from 0 to 1 that determines the output according to the linear equation, *a**(1-*blend*) + *b***blend*. When representing the hatching as a float value between 0 and 1, 0 represents a pixel with no hatching, and 1 represents a pixel with only hatching.

Randomness in strokes To add randomness to strokes, we use the noise function of RSL. The noise function is a stochastic function that appears random but is really pseudo-random, meaning its output is repeatable when given the same input. We provide a certain input for the noise function depending on how much we want to vary a particular attribute of the hatching (i.e. we use the layer number as input if we want to vary a characteristic differently depending on its layer, or we use the hatch id as input if we want to vary each hatch individually). The output from the noise function is then the varying factor of the characteristic (e.g. the rotational offset, if we vary the rotation of the hatching).

To accomplish this, we represent each hatch with a unique parameter, or id. If the id is constant across all pixels that make up a specific hatch, it can be used in noise functions to vary the hatch without breaking it up in undesirable ways. For example, suppose we have a noise function that will produce a psuedo-random number based on the function's input. For any particular input value, the exact same output will always be produced. We use the output from such a function to perturb the position of a hatch in u and v. Since our id is consistent across a particular hatch, each pixel of the hatch will be moved in u and v by the same amount.

For a general pattern that is repeated in parametric space on the surface of an object, we can use row and column indices to calculate a unique id for each instance of the pattern. If the number of iterations of the pattern in the u and v directions are known, the floor function can be used to find the row and column of the ”cell” at which the current pixel is located:

```
rowIndex = floor(u * freq)
colIndex = floor(v * freq)
```

The variable $freq$ is how many times the pattern is repeated, which may be different in the u and v directions. To apply this technique to the hatching, we apply the floor function to the product of u and dp, our base frequency. The result is shown in Fig. 2.14. Each band of color is a constant value, and represents the area covered by one hatch. This constant value can therefore be used as an id for the hatch. However, the figure illustrates problem areas in the regions where the frequency of the hatching doubles in value, shown by abrupt changes in value while scanning horizontally across the model. The frequencies in these areas need to be modulated so that each grey band has a constant width across the whole object. This problem is similar to the one discussed in the previous subsection for Fig. 2.7, which was solved by calculating the *transtriangle* value across the area spanned by the *transition* gradient.

We can again use the *transition* value to smooth out these areas of jumps in frequencies, albeit in a different manner. By looking at Fig. 2.14 we see that the color of the centers of every other smaller band corresponds exactly with the color of the central 50% of the adjacent larger band. By offsetting the color values only in the top and bottom regions

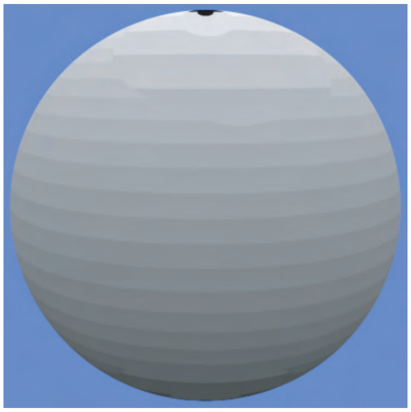


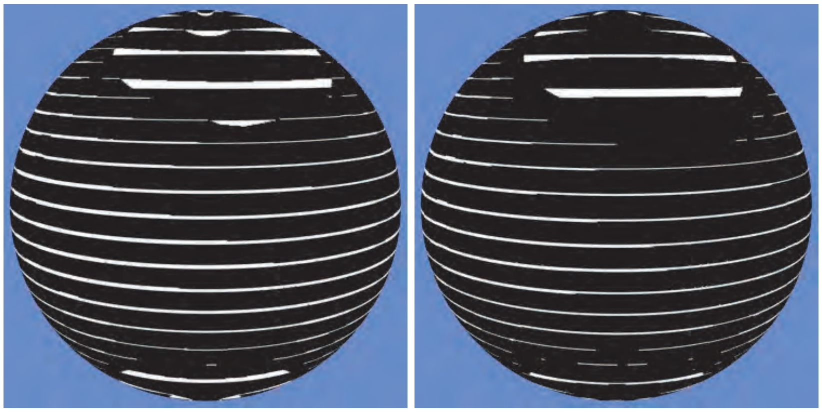
Figure 2.14: A visual representation of each hatch having a unique row index. Notice the discrete frequency jumps that are a result of not adjusting for transition areas.

of the larger bands, we can make them correspond in value to the smaller bands. Therefore, the first step is to automatically create masks for the top and bottom regions of the bands. By using the *sawtooth* variable, which we refer to here as *swt*, and the *transition* variable, we can pass them through a step function and create each of these masks. The formulas to do so are shown below, and images representing their values are shown in Fig. 2.15:

$$sawtrans = step(.5, swt) * (1 - step(transition/4 + .5, swt)) \tag{2.4}$$

$$sawtrans2 = (1 - step(.5, swt)) * step(.5 - (transition/4), swt) \tag{2.5}$$

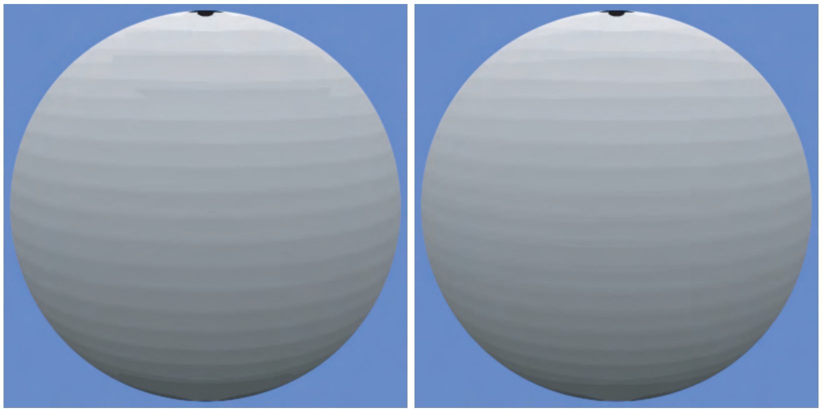
The next step is to calculate the amount by which to offset the tops and bottoms of the bands. Through experimentation we found a reasonable offset by multiplying the reciprocal of dp by .5. We then multiply the offset by our "top" mask so that the offset only affects the top regions of the bands. Subracting this offset from the row index lowers the values at the tops of the larger bands, making them consistent with the neigboring smaller bands (Fig. 2.16(a)). Likewise, we can increase the values at the bottoms of the larger bands by adding the product of the offset and the second mask to the row index (Fig. 2.16(b)). We now have regions with unique row indices that correspond with individual hatches and therefore maintain the property of controlled-density hatching. Since our hatching



(a) (b)

Figure 2.15: Masks used to isolate the (a) top and (b) bottom areas of the bands so that offsets may be applied to the masked areas in order to make smooth transitions of bands across frequency jumps.

technique uses multiple hatches per row, we apply our hatch id technique to the *v* direction to create a unique column index. We then use our row and column indices together as inputs to a 2D noise function to produce a unique value for each individual hatch, as shown in Fig. 2.17.



(a) (b)

Figure 2.16: Fixing the frequency jump problem. (a) The result after subtracting our offset from Fig. 2.14 in the areas masked by Fig. 2.15(a). (b) The result after adding our offset to Fig. 2.14 in the areas masked by Fig. 2.15(b).

Another method to vary the hatches is by using their layer index. For example, to vary the orientation of the hatching, we perturb the rotation of the hatch around its center. To change this characteristic differently for each layer of the hatching, the layer id is input

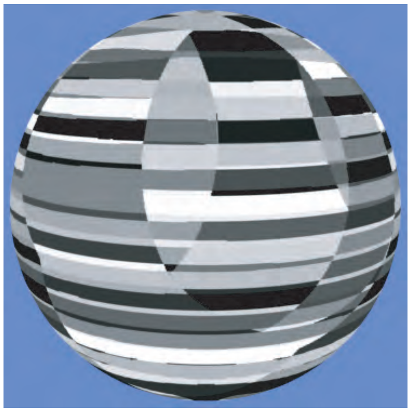


Figure 2.17: By using the row and column of a hatch as input into a noise function, a unique id can be created for each hatch.

into the noise function. The output from the noise function is then input into a function that rotates a point in 2D surface space. To find the new positions of a point at *u* and *v* on the surface of a hatch, we do the following:

```
for (i = 1; i <= layers; i += 1) {
    ...
    float noii = noise(i * 10 + 0.5);
    rotate2d(u, v, radians(noii), centerX,
        centerY, newU, newV);
    ...
}
```

Another technique for varying the hatching is known as bombing. In [May96], "bombing" is described as random placement patterns where a shape is dropped at random positions in space (see Fig. 2.18). Using row and column indices of a hatch, we randomize its position by doing the following:

```
float noirc = noise(c * 10 + 0.5, r * 10 + 0.5);
scat = udn(noirc * 967, min, max);
if (scat >= threshold && scat < 1-threshold) {
    rndx = udn(noirc * 834, xmin, xmax);

    rndy = udn(noirc * 726, ymin, ymax);

    ...

    hatching code

    ...

}
```

We use the noise output, *noirc*, as input to a uniformly distributed noise function, udn. The udn function returns a pseudo-random number that is evenly distributed between the given parameters *min* and *max*. By so doing we generate three different random outputs, *randx*, *randy*, and *scat*. The variables *randx* and *randy* are our random positions for the bombing effect. If *scat* is within a certain threshold, then the rest of the code to create the current hatch is executed. This allows only a portion of the hatches to be drawn, further helping the bombing effect. In a similar fashion, other attributes of the hatching may be varied. We describe the meaning and use of these attributes in the next section.

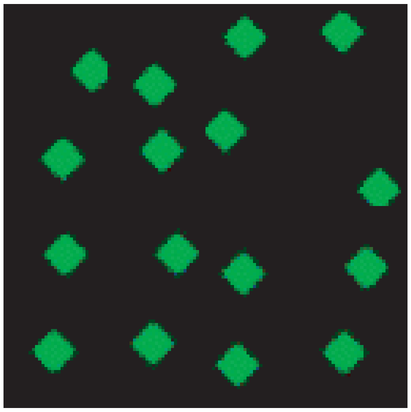


Figure 2.18: An example of the bombing technique described by [May96] using an image of a simple diamond pattern also shown in [May96]. Each diamond in a regular, evenly spaced grid of 4 x 4 diamonds has been perturbed and put in a new location within its own cell.

2.5 Artistic Control

While our procedural hatching algorithm is part of an object's shading, the most important part of our method is accomplished during the lighting phase. Although default values are set for the hatching's parameters on any object to which it is applied, the lighters must be able to effectively control how the hatching appears on objects. A particular effect may be applied to many different shots in a film, so it is especially important that the tools for applying the effect are straightforward and efficient to use. Most other recent NPR papers focus more on the speed or look of their methods but do not detail much about the interface and controls used to apply their methods. While applying hatching to many shots of a film, several factors my make the process slow or cumbersome. Many models may be too thin or small or have too irregular shapes to use normal strokes for hatching. Or as the lighting scenarios of the film change, the hatching may be adjusted more than once to prevent it from being inconsistent or overbearing. In this section we discuss the parameters used to adjust the hatching and actual examples of how we use these parameters and additional helper controls to light a shot.

2.5.1 User Interface Parameters

Once all of the object surface shaders have been completed for a shot, it is assigned to a lighter for the placement of its lights. As we discussed above, each shot contains a set palette and a shot palette of surface shaders. The input parameters of the shaders have already been set, except for any shot specific cases where they may need to be tweaked. Since the hatching is so dependent on lighting, its parameters are adjusted the most while the shot is being lit. In Tables 1 through 3 below are charts of the parameters used to adjust the appearance of the hatching and descriptions of what they do. We have divided the parameters into a few different categories to help explain their uses. Some of these parameters are rarely changed from shot to shot or object to object, while others are changed frequently. We briefly explain in the charts what each of the parameters do, spending more time with those that are more crucial to the look of the hatching.

Parameter Name	**Parameter Description**
single layers	number of layers of single hatching
cross layers	number of layers of cross hatching
cross degrees	angle offset of cross hatching
rotation offset	global angle offset for all hatching
u offset	global offset in u for all hatching
v offset	global offset in v for all hatching
cross start	lighting threshold where cross hatching starts
wiggle amplitude	height of waves in wiggles of hatching
wiggle frequency	frequency of waves in wiggles of hatching

Table 2.1: General attributes and settings.

The first group of parameters are general attributes and settings for the hatching. Once satisfying default values have been found, these parameters do not need to be adjusted much among different objects and shots. The rotational offset is for all of the hatching on the object and is offset from either the u or v direction. This is useful when the u and v directions are not visually suitable for the hatching. Likewise, the u and v offsets are global shifts of the hatching, and they exist primarily to prevent the hatching from looking identical on any two objects. The wiggles of the hatching are sine waves that give the hatches a sense of waviness if straight hatches are not desired.

Parameter Name	**Parameter Description**
rotate max	max rotational variation
rotate min	min rotational variation
vrep min	min hatch repititions in v
vrep max	max hatch repititions in v
urep min	min hatch repititions in u
urep max	max hatch repititions in u
hi width	upper bound of width of hatches

Table 2.2: Bounds on variation of the hatches.

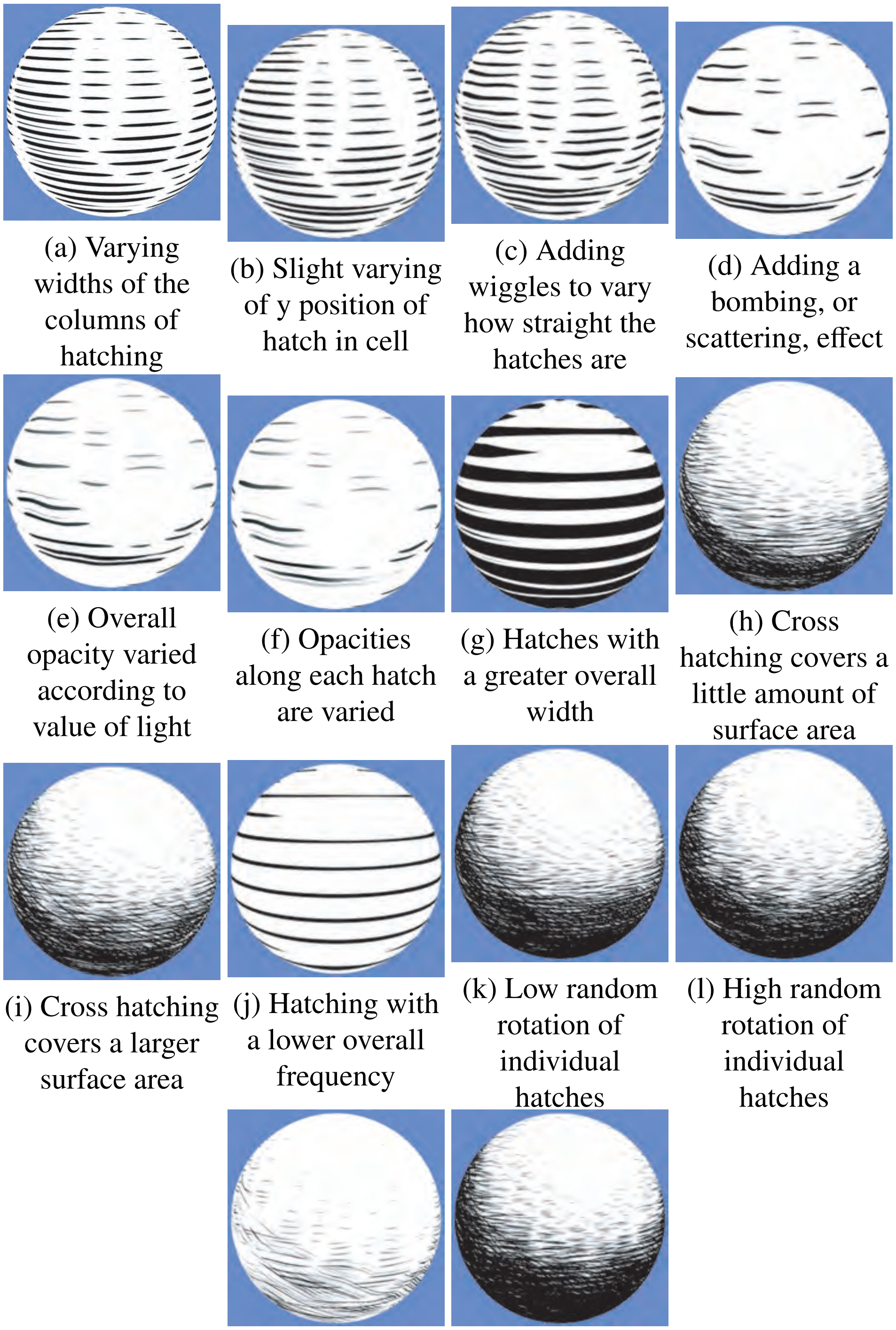
The parameters in Table 2 deal more with the bounds of the variation of the hatches. Examples include the minimum and maximum number of rows and columns of hatches or the extents of how far a hatch can be rotated from its normal direction. They are quick to tune and are not changed very often so that the hatching maintains a consistent look.

To better understand these first two groups of parameters, we have included a series of images (Fig. 2.19) that exhibit variations in the parameters being applied to a sphere. The first several images show certain parameters being applied in an iterative fashion. After these, there are more images that show additional possible parameter settings for the hatching.

Parameter Name	**Parameter Description**
lite1 oldmin	old min for overall intensity mapping 1
lite1 oldmax	old max for overall intensity mapping 1
lite1 newmin	new min for overall intensity mapping 1
lite1 newmax	new max for overall intensity mapping 1
lite2 oldmin	old min for overall intensity mapping 2
lite2 oldmax	old max for overall intensity mapping 2
lite2 newmin	new min for overall intensity mapping 2
lite2 newmax	new max for overall intensity mapping 2
vrep start	intensity mapping for max reps in v
vrep end	intensity mapping for min reps in v
urep start	intensity mapping for min reps in u
urep end	intensity mapping for max reps in u
scatter start	intensity mapping for max density of scatter
scatter end	intensity mapping for min density of scatter
cross scat start	mapping for max density of cross scatter
cross scat end	mapping for min density of cross scatter
width start	intesity mapping for max width
width end	intensity mapping for min width
cross width start	intensity mapping for max cross width
cross width end	intensity mapping for min cross width

Table 2.3: Remapping hatching due to lighting.

The final group of parameters, shown in Table 2.3, are parameters that control how the hatching responds to the intensity of light on an object. Of all the parameters, these are adjusted the most. For the above pairs of parameters that are named with *start* and



(a) Varying widths of the columns of hatching

(b) Slight varying of y position of hatch in cell

(c) Adding wiggles to vary how straight the hatches are

(d) Adding a bombing, or scattering, effect

(e) Overall opacity varied according to value of light

(f) Opacities along each hatch are varied

(g) Hatches with a greater overall width

(h) Cross hatching covers a little amount of surface area

(i) Cross hatching covers a larger surface area

(j) Hatching with a lower overall frequency

(k) Low random rotation of individual hatches

(l) High random rotation of individual hatches

(m) A few layers of hatching

(n) Multiple layers of hatching

Figure 2.19: Shown here is a series of images that show the effects of many of the hatching settings. Images (a) through (f) form a progression showing various ways of adding variety to the hatching, where each image contains the indicated type of variation added to the previous image. Images (g) through (n) show other settings that add variation to the hatching.

end, a remapping function is used to map the lighting value extremes (defaults are 0 and 1) to the values that are input for *start* and *end*. For example, suppose that the values 1 and 30 had been entered for the parameters *urepmin* and *urepmax*, respectively. Wherever there is an intensity of 0, the number of repititions of hatches in *u* at that point will be 1. Wherever there is an intensity of 1, the number of repititions of u will be 30. All intensity values between 0 and 1 will be mapped linearly between 1 and 30. The above min and max parameter pairs are used to remap the intensity value on an object to a different range to adjust the hatching tones. The following function is used to remap a variable from one range to another range:

$$\begin{aligned} newValue = (newMax - newMin)/(oldMax - oldMin) \\ *(value - oldMin) + newMin \end{aligned} \tag{2.6}$$

where *value* is the intensity of light in the domain from 0 to 1 that is originally calculated for the current pixel of the object that is being shaded. For example, suppose that a certain point on an object has an intensity of .75. The lighter decides that this value creates a hatching tonal value that looks too dark. To fix it, the values 0, 1, 0, 2, are input for the parameters *oldmin*, *oldmax*, *newmin*, and *newmax*. By plugging in these values to the above remapping equation, we get:

```
(2 - 0) / (1 - 0) * (.75 - 0) + 0 = 1.5
```

which changes the original value of .75 to 1.5, creating a lighter tone of hatching. If .5 is entered for both *oldmin* and *newmin*, only values .5 and above are be remapped to higher values while the rest remain unchanged.

Although the hatching parameters allow the hatching to be flexible for many types of models, there are some types of models where hatching is not desired. In many artistic drawings, such as pen-and-ink drawings, the brush or stroke is applied everywhere in the drawing. In an animated film, however, it is not always possible or even necessary to apply the stroke to every object. For example, the eyes of characters are very important for expressing feelings through the animation of a character and a stroking pattern could hinder the communication of these expressions to the viewer. Water, rain, and hair are examples of simulated effects that usually do not have a normal surface mesh on which hatching could be placed.

Additional parameters allow stippling to be used instead of hatching on very small or thin objects, such as grass and vines. With this option, objects that cannot recognizably display hatching will still maintain a similar look and feel to surrounding objects that are hatched. We use a quick and easy method to compute the stippling based on a method in [AG00] to create voronoi noise. At normal scales, this effect has the look of random cells. For the stippling, however, we scale the cells down to a small size to give the effect of lots of small dots. Opacity levels are varied to give the appearance of changing tonal levels. We refer the reader to [AG00] for an explanation of how the voronoi noise works, but we include below an example of how we used it in our method:

```
voronoiNoise(point*pointScale + offset,
    jitter, dist, featurePos)
stipple = smoothstep(smoothStart, smoothEnd, dist)
```

The voronoi noise function divides a surface into cells and computes a center point for each cell. The center points are perturbed within their respective cells according to the *jitter* input. The function takes a point as an input, and then computes the position of the nearest feature and the distance to this feature from the center point. These values are then set to the *dist* and $featurePos$ variables, respectively. Our implementation allows the user to scale and offset the input point, which serves to scale and randomize the look of the resulting cells. Finally, we use a smoothstep to harden the edges of the resulting cells to make them appear as dots instead of cells with gradients. In Fig. 2.20, the first image

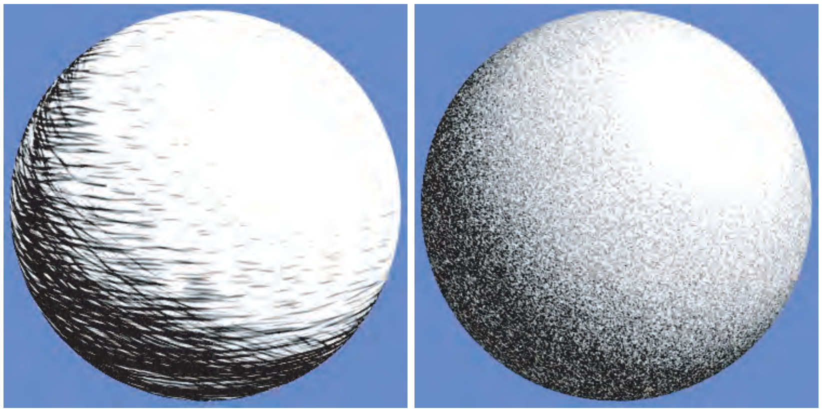

(a) (b)

Figure 2.20: Stippling applied to a sphere. (a) Normal look of hatching. (b) Stippling turned on.

shows an example of the normal hatching applied to a sphere. The second image is the same sphere with stippling applied, using the same settings and lighting conditions.

2.5.2 Controlling the Hatching

While the available parameters offer flexibility in designing the look of the hatching, it may be challenging to find desirable settings. Sometimes many iterations of trying settings, rendering, and changing settings are required before the desired settings are found. Following this process for many shots of a film can be very time consuming. Consequently, a few lighting controls have been created to help the lighter arrive at the desired settings more quickly. The first two controls are rendering ”modes” that help the user visualize how the lighting changes across an object’s surface. An additional control specifies whether a light should contribute just to the hatching, just to the color of an object, or to both.

The first of the two render modes is the grey light mode. When this mode is turned on, the object’s shader only outputs the value of the light contribution at every point on the object. Since these values are used as inputs into the hatching shader to vary its parameters, rendering these values gives insight into why the hatching changes as it does across the object. Fig. 2.21 shows the default hatching image from above and how it looks with the grey light mode turned on.

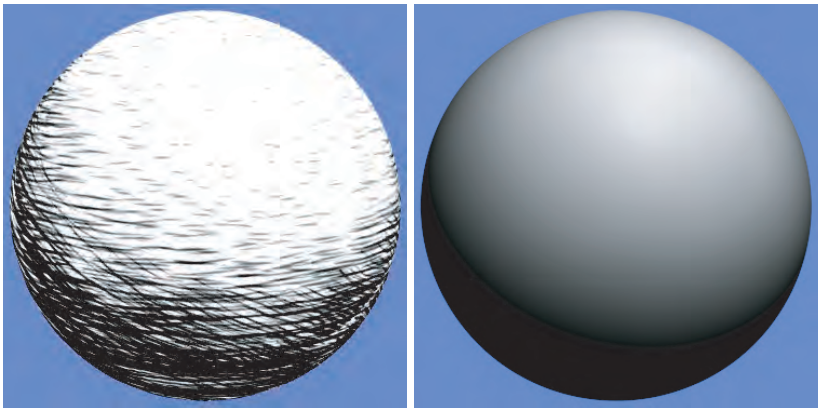
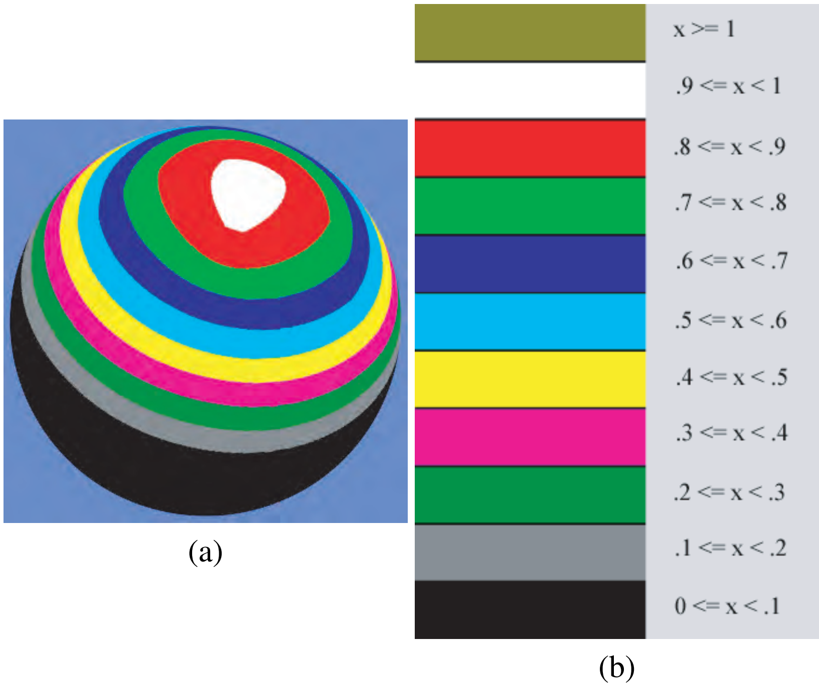

(a) (b)

Figure 2.21: When the grey light mode of the hatching is turned on, only the value of the lighting is output. (a) A normal image showing hatching on a shpere. (b) The image with the grey light mode turned on.

Even while using the grey light mode, discerning the exact grey scale values can still be challenging. For example, suppose the user wants to apply a cross hatch on portions of an object that are below a particular lighting value threshold, *lv*. While rendering with the grey light mode, several iterations of changing values and rendering are required to interpret the correct threshold at which to set the cross hatch. The color ramp mode is an additional rendering mode to help interpret lighting values across an object. When this mode is turned on, the lighting contribution values are divided into 11 different regions. Values ranging from 0 to .1 are in the first region, values from .1 to .2 are in the second region, values from .2 to .3 are in the third region, and so on until values 1 and above are in the final region. Each region is assigned a color, and this color is then rendered to the screen. This mode results in bands of color on the models that give a better idea of the breakdown of the lighting across the object. The lighting contribution can be divided into even more regions of color for more precise feedback. Fig. 2.22 shows an example of the color ramp and the ranges of values that each color represents.

We will give an example to show how the color ramp aids in setting and later adjusting the remapping sliders for the hatching. Suppose we start with a sphere that has hatching as shown in Fig. 2.23 along with its respective color code. According to the light-



(a)

(b)

Figure 2.22: (a) An example of how the color ramp mode looks when turned on for the sphere in Fig. 2.21. (b) The values that correspond to each color in the color ramp mode.

ing direction given for the shot, the hatching in the darker regions of the sphere must be darkened further while the hatching tone in the lighter regions must remain the same. By looking at the color code of the sphere, we see that the hatching tone ranges from 1 in the lightest area to about .2 in the darkest region. Therefore, we need to remap the lower bound of the lighting range to a lower value. We use the first set of remapping values to accomplish this. Since we want to remap the entire range of light to a greater range, we input 0 for $oldmin1$ and 1 for $oldmax1$. To darken the hatching and increase the total range, we need to decrease the lower bound while the upper bound remains the same. By using the values -.73 and 1 for our $newmin1$ and $newmax1$, respectively, we get the results shown in Fig. 2.24. As desired, the resulting sphere has a greater range of tones from light to dark across its surface.

To continue with the example, suppose a change is needed so that the hatching tone in the lower left corner of the sphere is lightened while the middle and lighter sections remain the same. Since we only want to affect one region of the sphere and one remapping has already been used, we must use the second set of remapping sliders to make this change. We refer to the color code in Fig. 2.24(b) and determine the range of values corresponding to the area we want to change. In this case, we want to use 0 for *oldmin*2 and .1 for *oldmax*2 (the black region). Since we want everything from .1 and above to remain the same, we need to use .1 for *newmax*2. To lighten the hatching, we then use .3 for *newmin*2. Fig. 2.25 shows the effect of this new change. Additional sets of remapping sliders may be added as situations arise that require more specific control over adjusting the tone of the hatching.

It is important to note that when we say the interval changes from [0, .1] to [.3, .1], we are changing the perceived density of the hatching only. The value 0 represents no light or full hatching density, and 1 represents the maximum amount of light or the lowest amount of hatching density. The amount of actual light that hits the object remains the same. Where the hatching's density used to change from 0 to .1, it's density now changes from .3 to .1 across the same spatial area.

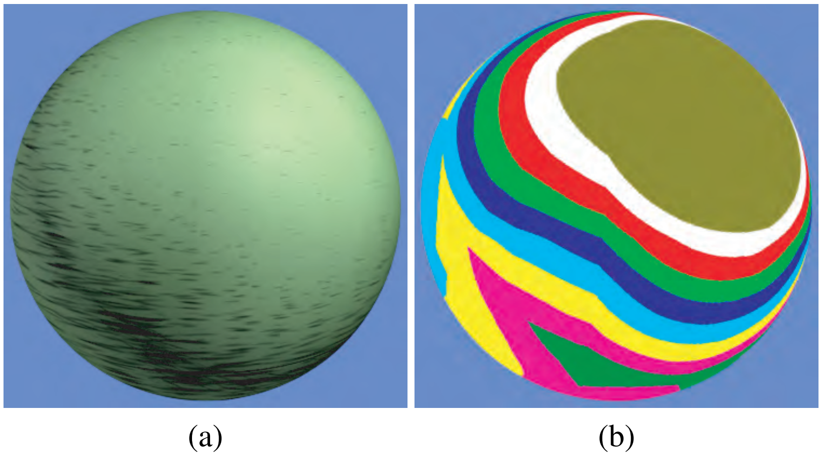


Figure 2.23: (a) A sphere with our starting settings for the hatching. (b) The same sphere with color ramp mode turned on.

Other useful tools for adjusting the hatching are lights that only affect the hatching or that only affect the color of an object. This is useful in cases in which the user determines that adjusting the position or attribute of a light to change the hatching tone is needed instead of adjusting the parameters of the hatching, but the color and intensity of the light on the object is correct. Or the reverse may be true where the hatching on a particular

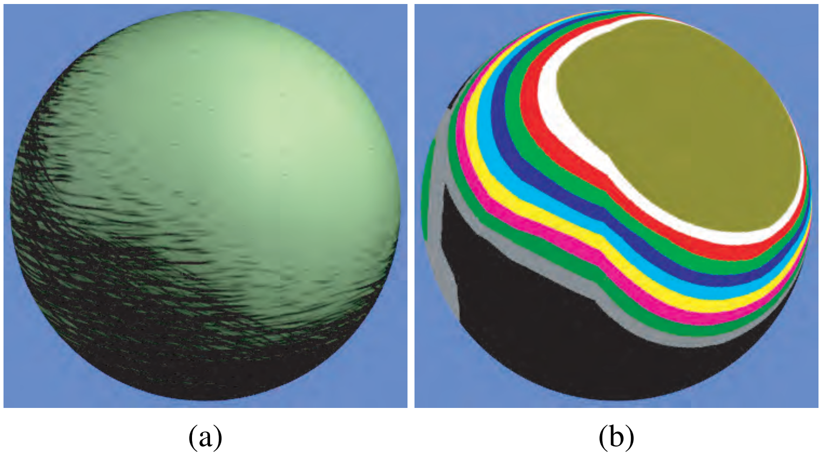
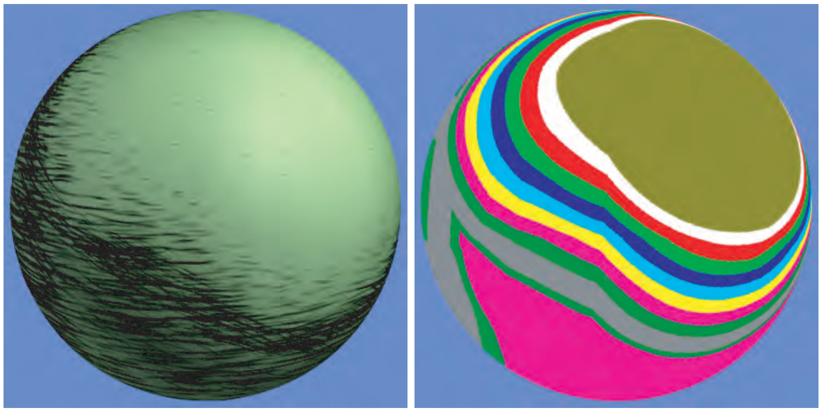

(a)
(b)

Figure 2.24: (a) By remapping the interval [0 1] on the sphere in Fig. 2.23 to [-.73 1], we get a greater range of tones from light to dark. (b) By comparing the change between spheres in color ramp mode, we can see more precisely how we increased the range of values.

object is set, but the texture color of the object that is showing through the hatching needs to be adjusted. To accomplish this, a light category option on the light shader specifies whether the light contributes only to the hatching of an object, only to its color, or both. The illumination models used in the surface shaders read in the category and omit the contribution of the lights to the hatching or color of the object accordingly. Therefore, if a color-only light is added to brighten the color of the object, it will not affect the hatching in any way. In some situations these categories provide a quick fix to an otherwise difficult lighting situation.

2.6 Results

As an NPR style for an animated film, we tested our method by applying it to a student short film called Noggin. The film consisted of 50 shots, totaling about 4 minutes. The film shows how our method adds tone and contrast to a film, giving it a unique hand-drawn feeling. Below we show different groups of images and their descriptions that show different ways hatching was used on the film.



(a) (b)

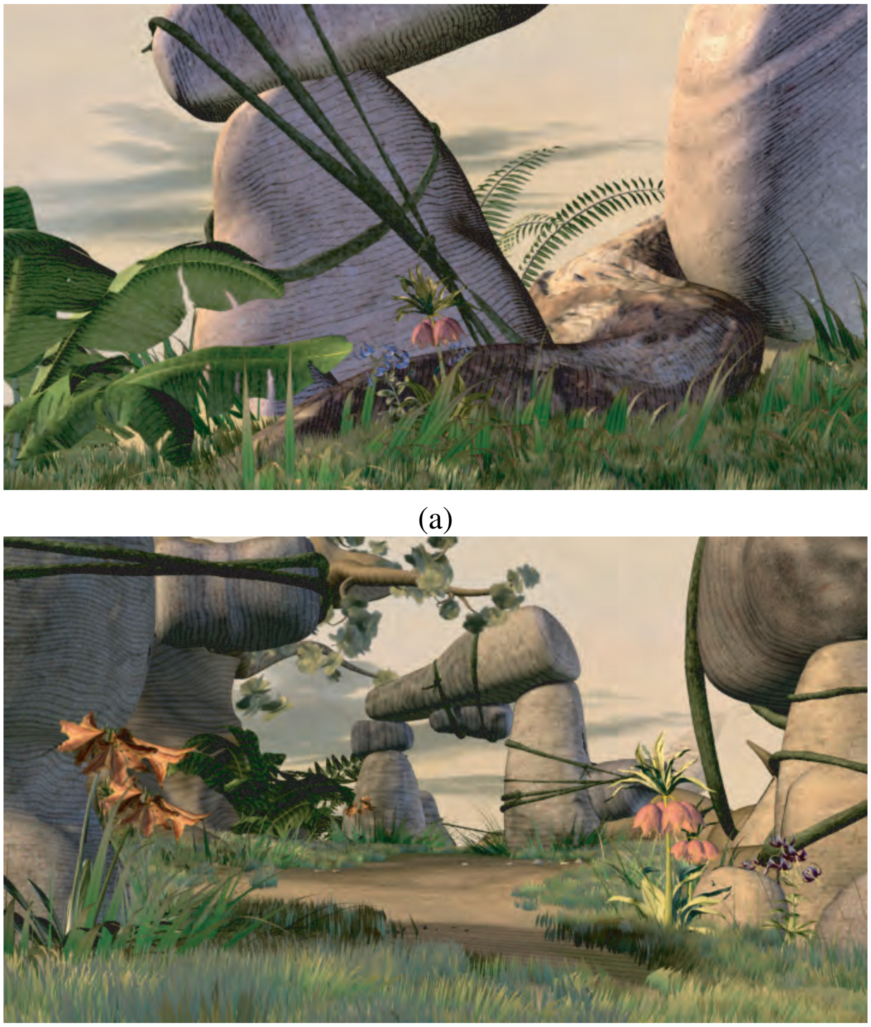
Figure 2.25: Remapping the second set of sliders from the interval [0, .1] to [.3, .1]. (a) The normal hatching that shows the result of this change. Notice how only the hatching at the bottom of the sphere changed, becoming lighter. (b) The sphere in (a) with the color ramp mode turned on.

2.6.1 Defining the Hatching Style

We experimented with various styles of hatching to determine which styles of hatching would be practical and usable with our system.

Our goal is a style that suggests a hand drawn look by using strokes to add tonal differences across objects and simulate 3D shading and lighting. In addition, we wanted to create hatching that had no distracting artifacts, especially when animated. Below are some potential problems we faced with our stroking style and our solutions to preventing or fixing the problems.

We attempted a style similar to the wood-block shader from the work of Apodaca that used long, continuous hatches. However, instead of increasing the width of hatches in areas of darker tones, we increased the frequency of the hatching. Fig. 2.26 shows an example of this. Although the tone of the hatching varied according to the values of light, the changes were not sufficient enough to create significant changes in the overall image. The hatches look more like stripes that are part of the underlying textures of the objects than a stroking technique that compliments the tonal changes in the images.



(a)

(b)

Figure 2.26: Two examples of images showing an early form of hatching similar to the wood block shader from [AG00]. The hatching does not add enough tonal contrast to the images; as a result, it looks more like stripes on the objects, rather than a part of their lighting.

One approach to fixing this problem was to increase the overall frequencies of the strokes and add an additional layer of cross hatching in the darkest tonal regions. Fig. 2.27 shows the result of this change. Moire patterns and other aliasing artifacts, such as dot crawling and shimmering, are problematic when creating patterns with many small features. These artifacts are especially distracting in animations. To help prevent aliasing, we used smaller overall frequencies, as well as anti-aliasing noise functions. These noise

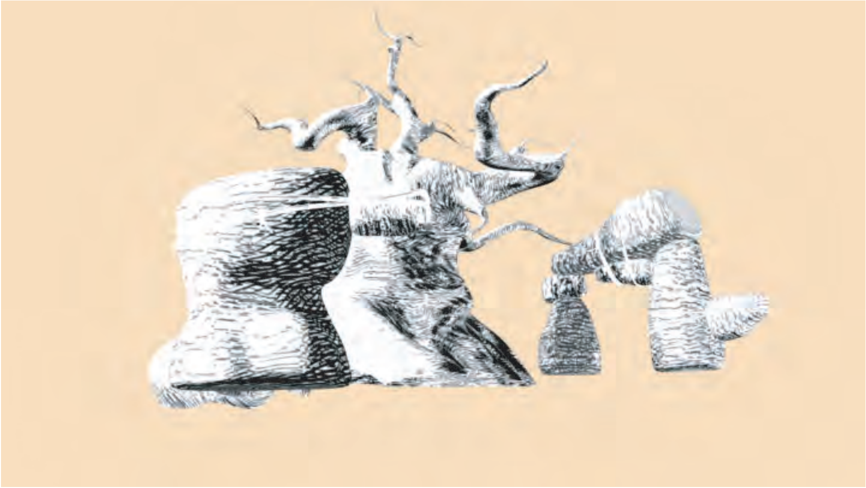


Figure 2.27: Too many small hatching strokes in one area can cause distracting moire patterns.

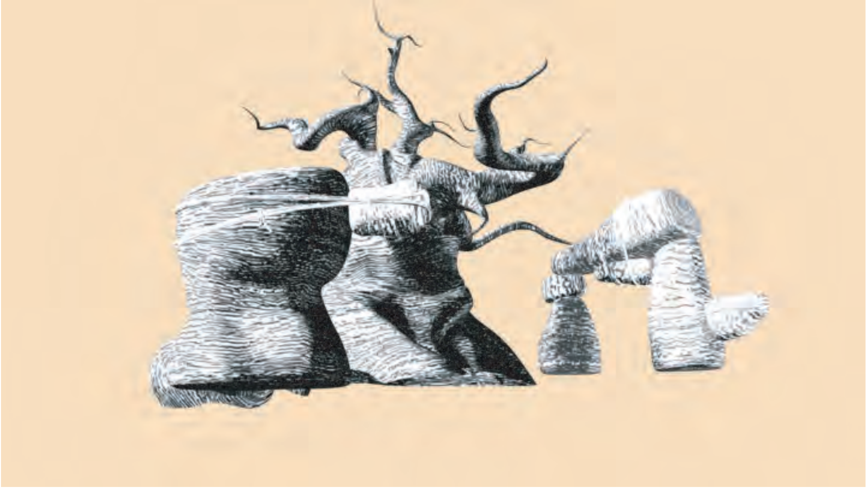


Figure 2.28: Although there is a nice change in tonal qualities in going from the light to the dark areas of this image, the hatching would be too overbearing on objects with added texture and lighting.

functions, described in [AG00], remap the input frequencies in order to prevent too low of sampling rates.

Another style we tried used greater contrast in the tones of the strokes, approaching near solid black in the darkest tonal regions, as shown in (Fig. 2.28). This style would only be desired for applications that are intended to be monochromatic in nature. For our method, therefore, the hatching is too overbearing and hides too much of the fine gradations in color that exist in the underlying textures and lighting, causing the 3D qualities of the image to be lost.

2.6.2 Hatching with Different Lighting Schemes

A film's style must be consistent throughout all of its shots. Our method must therefore be flexible enough for any lighting scheme throughout the film. In Fig. 2.29, we show images from five different shots of Noggin that each have a unique lighting scheme. Whether the shots are sunny, dark, foggy, or overcast, the hatching tones properly fit the surrounding lighting.

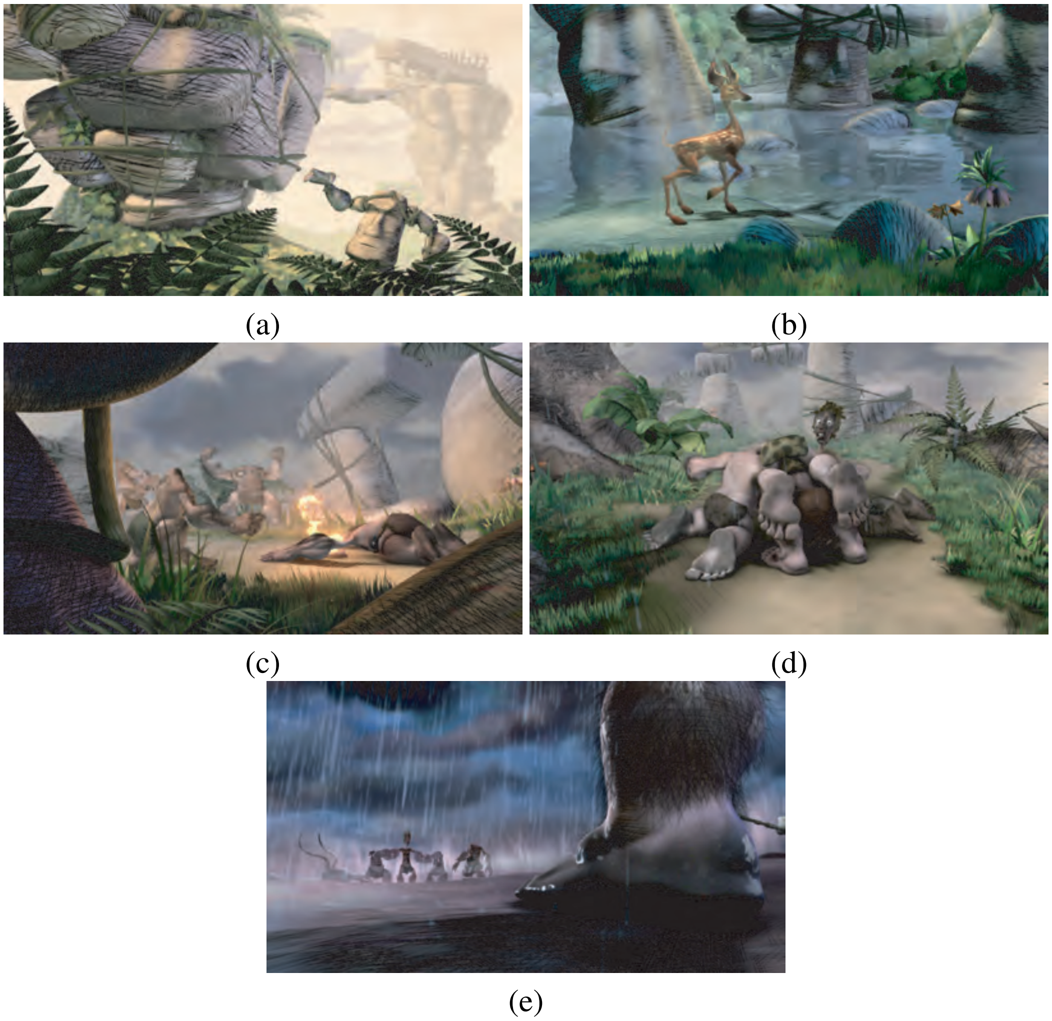
2.6.3 Hatching Defining Shadows

The images in Fig. 2.30 show examples of hatching in shadow regions. One of the most important criteria for any NPR method to be successfully used in animation is temporal coherence. Styles must be consistent and coherent across consecutive frames of animation without displaying distracting artifacts. In these examples, the hatching strokes gradually fade in and out in a smooth manner as the shadows pass over objects.

2.6.4 Hatching Integration

In the section on Artistic Control we discussed how we did not force our hatching technique on every type of object. The images in Figures 2.31 through 2.33 show cases in which the hatching was eliminated or changed. Having the control to make such changes is vital for making our method useful for other films, in that it shows that the effect will support and not hinder the final product.

The hatching adds certain characteristics to the images of a film. In Figure 2.34(a) we show an image before hatching has been applied. In Figure 2.34(b) through (d), we show three examples of the same image with different hatching settings applied to each one. The hatching adds contrast to the original image that reinforces the lighting by making its



(a)

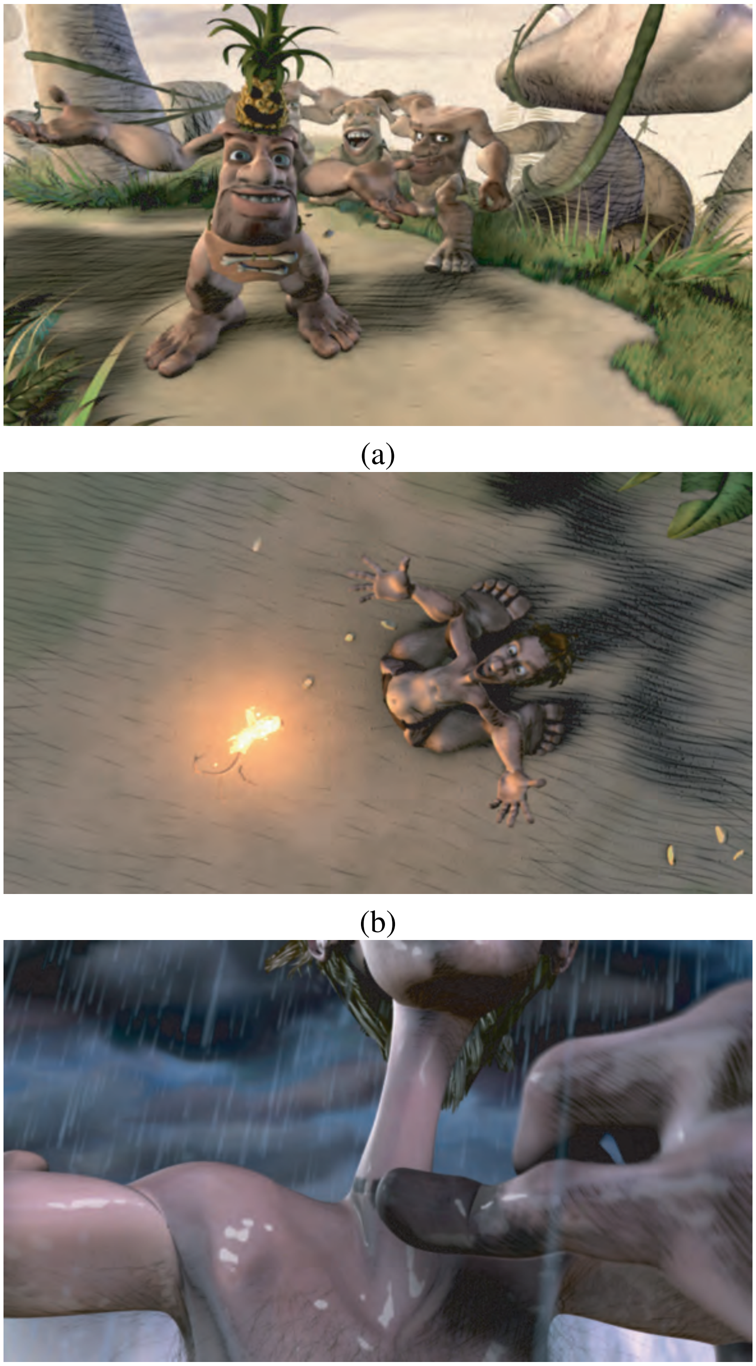
(b)

(c)

(d)

(e)

Figure 2.29: Hatching with Different Lighting Schemes

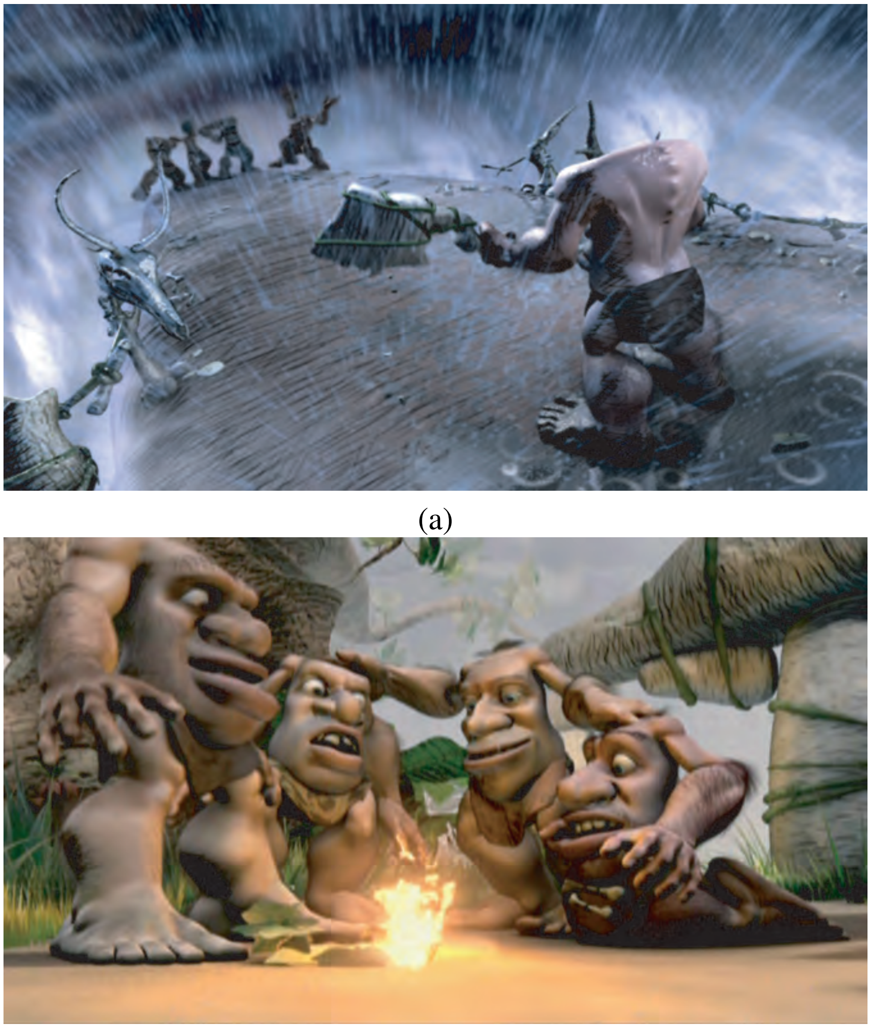


(a)

(b)

(c)

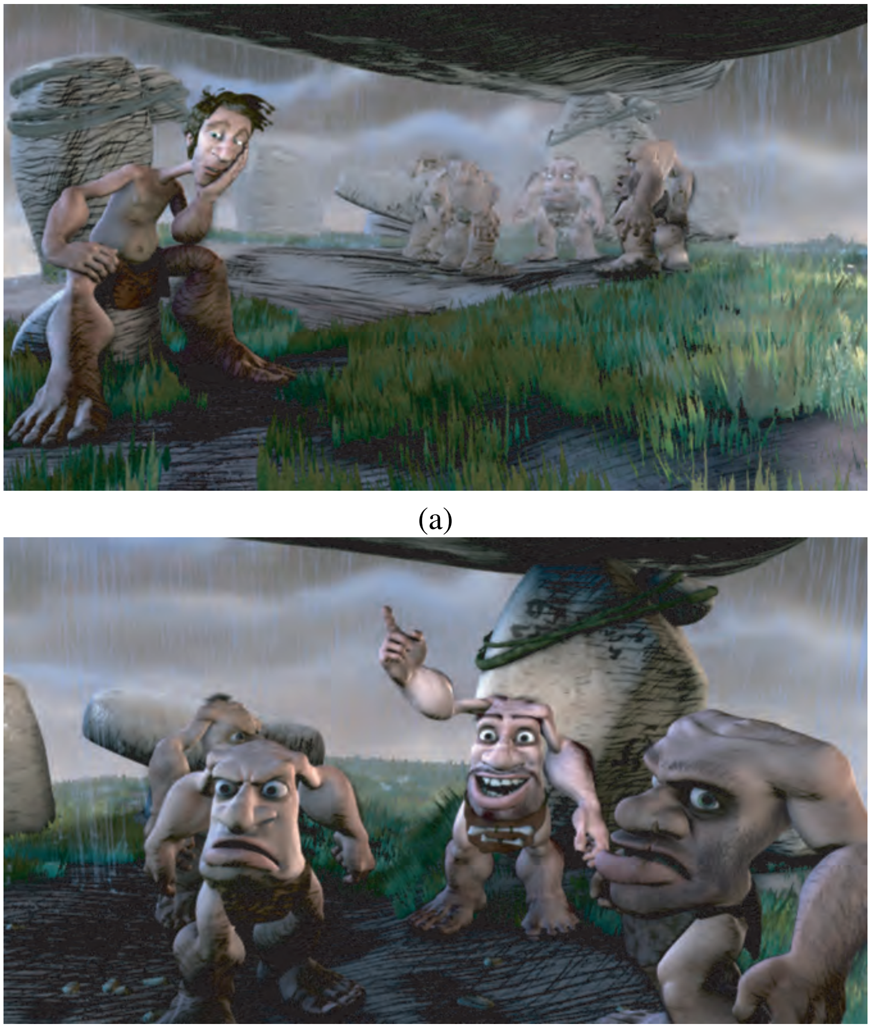
Figure 2.30: Hatching Defining Shadows



(a)

(b)

Figure 2.31: Hatching is used all throughout the film, even in shots with various effects, like rain, fire, and mist. Rather than try to enforce the hatching look in these effects, we provided the functionality to turn off the hatching on certain objects or parts of objects. Although doing so compromises the idea of consistency in using hatching on every object, this allows the hatching to be flexible to fit the needs of whatever film it is used on.



(a)

(b)

Figure 2.32: In (a), the hatching on objects in the background is faded along with the normal color to help convey depth in the image. In normal pen-and-ink hatching images, this would never be found. Due to our unique style of blending color lighting with hatching, this becomes necessary to help the look of the overall image. When the camera moves forward in the shot, as in (b), we see how the hatching unfades to stay consistent with the normal color of the objects.

apparent tone clearer. Also, the image without hatching contains smooth objects and little variation in color, which give it a definite computer generated look. Although the hatched images contain the same objects, the hatching in these images help them contain more variation and appear more hand-made.

2.6.5 Future Work

There are several ways that our method could be extended for future work. The stippling technique was used as an alternative to the hatching that had a similar effect but was better suited to very thin and small objects. The stippling would be improved by creating a more flexible technique. Currently the stippling is a combination of voronoi noise and an opacity mapping based on lighting. A better technique, however, would be to create circle patterns and then control the apparent density of the circles consistently among all objects, similar to how the hatching stroke pattern is created and its density is controlled. Different types of variation could be used on the circular patterns to denote different tonal levels. Like the hatching, this would make the stippling a more useful technique.

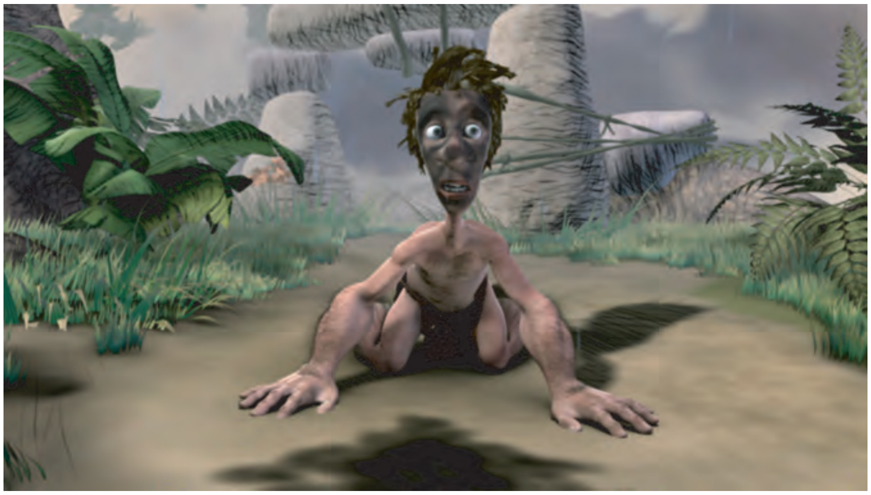
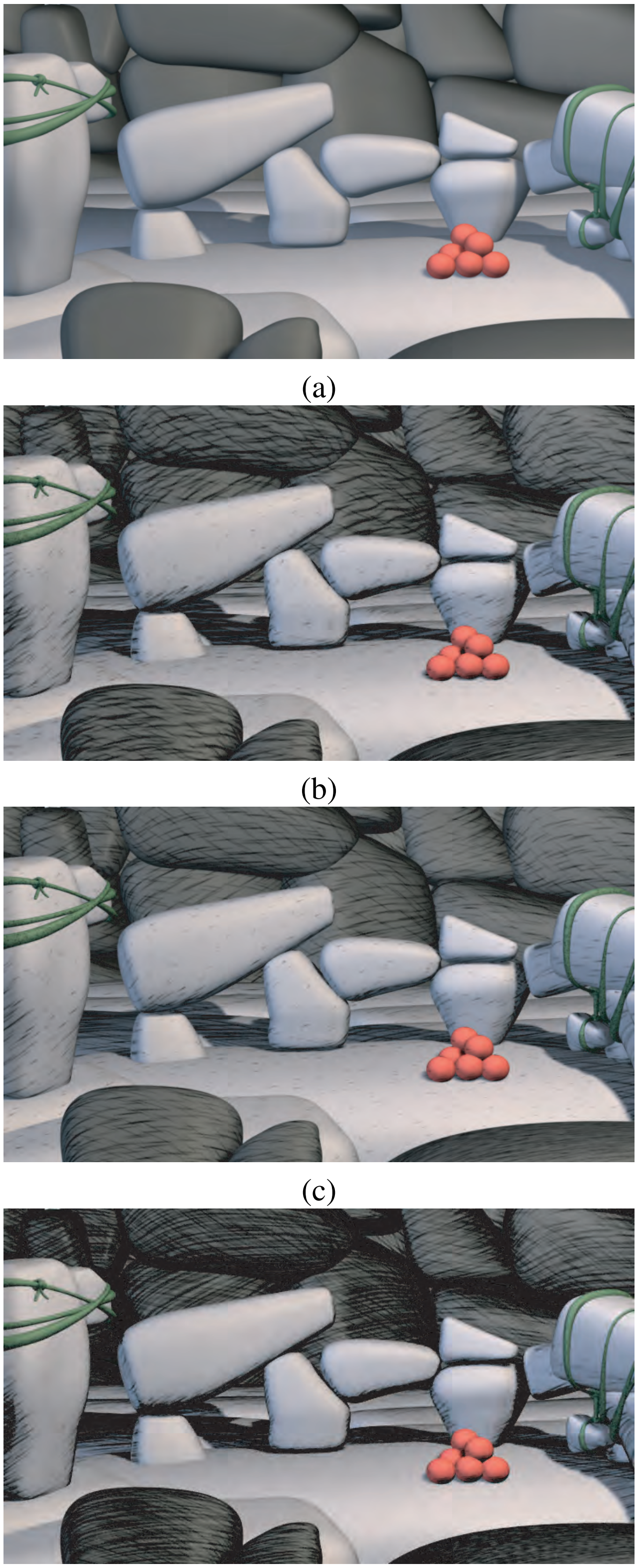


Figure 2.33: A light stipple is used on the grass instead of hatching. Trying to coordinate the hatches among the various blades of grass and make it work with moving cameras would have been extremely difficult and not very useful in the end. Allowing stippling for such small objects helped keep the look of the objects consistent in feel with the hatching on other objects without requiring a new and separate hatching technique.



(a)

(b)

(c)

(d)

Figure 2.34: (a) A scene with no hatching. (b) A scene with hatching applied. (c) As compared with (b), the hatching in this image is straighter and thinner. (d) The hatching in this image has a high frequency and a high amount of contrast.

Another area of future work could be to investigate other types of NPR techiques that can be used in 3D animated films. Many different styles have been used to create 2D animations. Determining how to apply these styles to 3D animations without making them look too flat and 2D would contribute much in the way of creative stylizations of film. NPR methods that are more pattern-based in nature could extend techniques used in our hatching method, such as was previously discussed on extending the controlled-density technique to stippling patterns. Controls such as the grey light and color ramp modes could also be utilized in other contexts to aid in visualizing changes in lighting. Our hope is that our contribution will serve as more than a technique to create hatching, but will help in opening up other avenues in the art of creative film making.

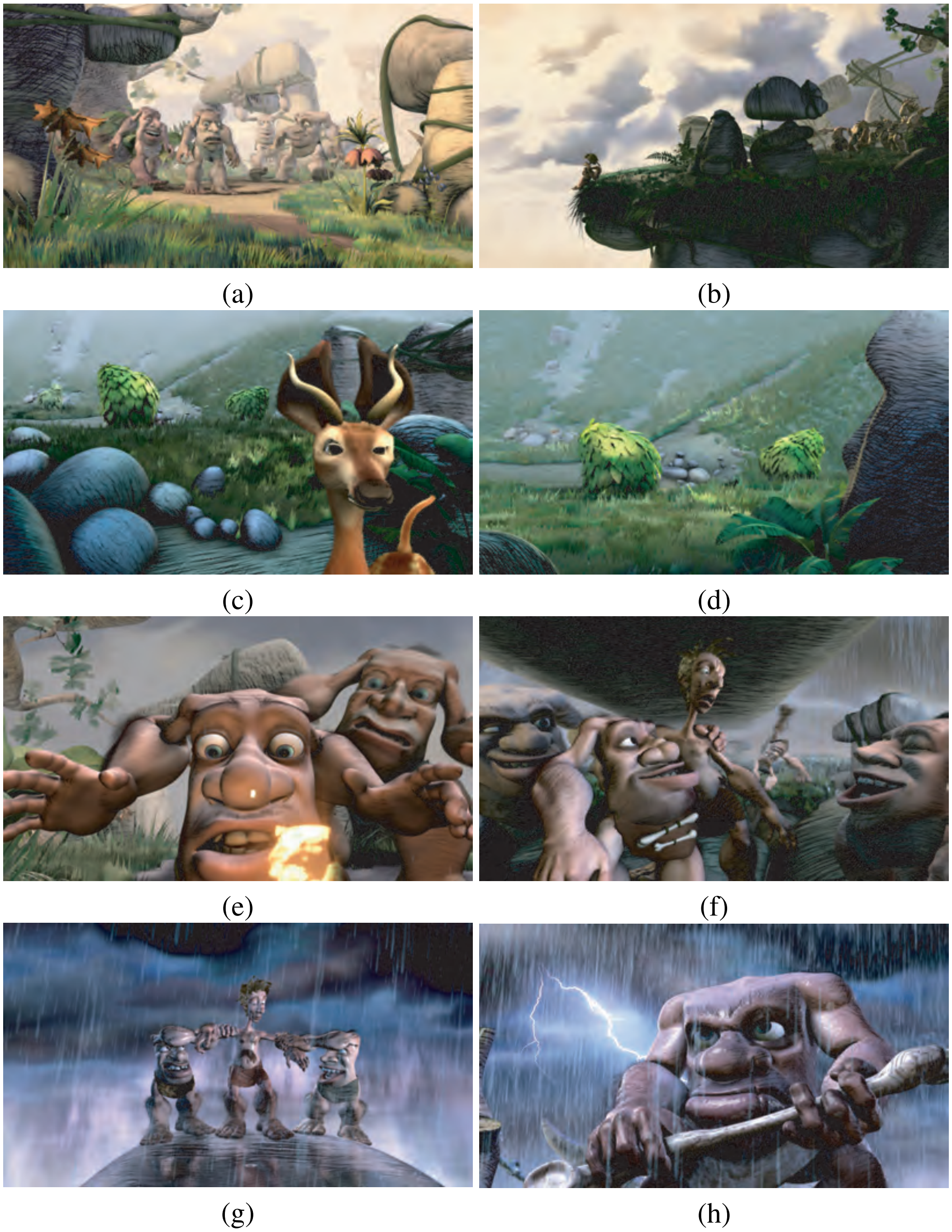


Figure 2.35: Additional images from various shots in Noggin that show the flexibility in the use of the hatching throughout the film.

Chapter 3

Conclusion

In this paper we have presented a new NPR method to stylize the look of a 3D animated film. Our method is unique in that it adds a pen-and-ink style of hatching to the film that gives it a hand-drawn feel while preserving its colors from shading and lighting. The hatches add an additional level of interest to the film by adding more contrast in the darker regions and by producing patterns that change smoothly with changes in lighting. The options and tools provided to fine-tune the hatching allow interesting and natural patterns to be created across multiple shots in a reasonable amount of time.

The most important aspects of our work are the technical solutions it provides to problems that are highly creative in nature. To define exactly what it means to make a film look interesting is difficult. Many NPR methods are very successful in providing solutions to mimicking artistic styles that accurately portray the style and that run faster and more efficiently than previous methods. However, most do not give the proper emphasis to addressing the problems that limit the use of these methods in real-world applications. Instead of only using simple objects with basic camera moves or other simple animations in our test cases, we show the use of our method on an actual 3D film with multiple shots and complex models. Also, we do not limit our method to specific types of models or special case scenarios, but provide functionality so that it may be used to enhance the look of any film no matter what models it includes or how complex they are. The art of filmmaking contains its own set of unique challenges that must be specifically addressed in order to allow new techniques and styles to be practical. The work presented in this thesis contributes to this field by providing styles for animated films that enhance their visual impact in new ways.

Bibliography

[AG00] Anthony Apodaca and Larry Gritz. Advanced renderman: Creating cgi for motion pictures. pages 457–468. Morgan Kuafmanm, 2000.

[ARS79] Arthur Appel, F. James Rohlf, and Arthur J. Stein. The haloed line effect for hidden line elimination. In *SIGGRAPH '79: Proceedings of the 6th annual conference on Computer graphics and interactive techniques*, pages 151–157, New York, NY, USA, 1979. ACM Press.

[BW03a] Kevin Buchin and Maike Walther. Hatching, stroke styles, and pointillism. In *ShaderX2 - Shader Tips and Tricks*, pages 340–347. Wordware Publishing, Inc., 2003.

[BW03b] Kevin Buchin and Maike Walther. Real-time per-pixel rendering with stroke textures. In *SCCG '03: Proceedings of the 19th spring conference on Computer graphics*, pages 125–129, New York, NY, USA, 2003. ACM Press.

[BWL] William Baxter, Jeremy Wendt, and Ming C. Lin. IMPaSTo: A realistic, interactive model for paint. In *Proceedings of NPAR 2004, The 3rd International Symposium on Non-Photorealistic Animation and Rendering*.

[CRH05] John P. Collomosse, David Rowntree, and Peter M. Hall. Stroke surfaces: Temporally coherent artistic animations from video. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):540–549, 2005.

[CSUN05] Youngha Chang, Suguru Saito, Keiji Uchikawa, and Masayuki Nakajima. Example-based color stylization of images. *ACM Trans. Appl. Percept.*, 2(3):322–345, 2005.

[Cur98] Cassidy J. Curtis. Loose and sketchy animation. In *SIGGRAPH '98: ACM SIGGRAPH 98 Electronic art and animation catalog*, page 145, New York, NY, USA, 1998. ACM Press.

[DC90] Debra Dooley and Michael F. Cohen. Automatic illustration of 3d geometric models: lines. In *SI3D '90: Proceedings of the 1990 symposium on Interactive 3D graphics*, pages 77–82, New York, NY, USA, 1990. ACM Press.

[DFR04] Doug DeCarlo, Adam Finkelstein, and Szymon Rusinkiewicz. Interactive rendering of suggestive contours with temporal coherence. In *NPAR '04: Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*, pages 15–145, New York, NY, USA, 2004. ACM Press.

[DS00] Oliver Deussen and Thomas Strothotte. Computer-generated pen-and-ink illustration of trees. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 13–18, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[EMP+03] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. Texturing and modeling: A procedural approach. pages 157–176. Morgan Kaufmann Publishers, 2003.

[FMS04] Bert Freudenberg, Maic Masuch, and Thomas Strothotte. Real-Time Halftoning: Fast and Simple Stylized Shading. In Andrew Kirmse, editor, *Game Programming Gems 4*, pages 443–440. Charles River Media, 2004.

[Fre01] B. Freudenberg. Real-time stroke textures. In *SIGGRAPH 2001 Conference Abstracts and Applications*, page 252. ACM Press, 2001.

[FV03] Jennifer Fung and Oleg Veryovka. Pen-and-ink textures for real-time rendering. In *Graphics Interface*. A K Peters, LTD., June 2003.

[GG01] Bruce Gooch and Amy Gooch. In *Non-photorealistic Rendering*. A. K. Peters, Ltd., 2001.

[Her98] Aaron Hertzmann. Painterly rendering with curved brush strokes of multiple sizes. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 453–460, New York, NY, USA, 1998. ACM Press.

[HHD04] Michael Haller, Christian Hanl, and Jeremiah Diephuis. Non-photorealistic rendering techniques for motion in computer games. *Comput. Entertain.*, 2(4):11–11, 2004.

[HZ00] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 517–526, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[LD04] Thomas Luft and Oliver Deussen. Non-Photorealistic Real-Time Rendering of Characteristic Faces. In *Proceedings of Pacific Graphics 2004*, pages 339–347, 2004.

[LMHB00] A. Lake, C. Marshall, M. Harris, and M. Blackstein. Stylized rendering techniques for scalable real-time 3d animation, 2000.

[May96] Stephen F. May. Rmannotes. In *http://accad.osu.edu/~smay/RManNotes/*. Stephen F. May, 1996.

[Mei96] Barbara J. Meier. Painterly rendering for animation. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 477–484, New York, NY, USA, 1996. ACM Press.

[MKT$^+$97] Lee Markosian, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes. Real-time nonphotorealistic rendering. *Computer Graphics*, 31(Annual Conference Series):415–420, 1997.

[ND04] Marc Nienhaus and Jrgen Dllner. Sketchy drawings. In *AFRIGRAPH '04: Proceedings of the 3rd international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 73–81, New York, NY, USA, 2004. ACM Press.

[PHWF01] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 579–584, 2001.

[SABS94] Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive pen-and-ink illustration. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 101–108, New York, NY, USA, 1994. ACM Press.

[SALS96] Mike Salisbury, Corin Anderson, Dani Lischinski, and David H. Salesin. Scale-dependent reproduction of pen-and-ink illustrations. In *Computer Graphics, Proceedings of ACM SIGGRAPH 1996*, pages 461–468, 1996.

[SLK05] Kaleigh Smith, Yunjun Liu, and Allison Klein. Animosaics. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 201–208, New York, NY, USA, 2005. ACM Press.

[VB99] Oleg Veryovka and John W. Buchanan. Comprehensive Halftoning of 3D Scenes. In Pere Brunet and Roberto Scopigno, editors, *Proceedings of EuroGraphics'99 (Milano, Italy, sep 1999)*, volume 18, pages 13–22, Oxford, September 1999. NCC Blackwell Ltd.

[Ver02] O. Veryovka. Animation with threshold textures. In *Graphics Interface 2002*, 2002.

[WB03] Holger Winnemoller and Shaun Bangay. Rendering optimisations for stylised sketching. In *AFRIGRAPH '03: Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and interaction in Africa*, pages 117–122, New York, NY, USA, 2003. ACM Press.

[WM04] Brett Wilson and Kwan-Liu Ma. Rendering complexity in computer-generated pen-and-ink illustrations. In *NPAR '04: Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*, pages 129–137, New York, NY, USA, 2004. ACM Press.

[WS94] Georges Winkenbach and David H. Salesin. Computer-generated pen-and-ink illustration. *Computer Graphics*, 28(Annual Conference Series):91–100, 1994.

[WS96] Georges Winkenbach and David H. Salesin. Rendering parametric surfaces in pen and ink. *Computer Graphics*, 30(Annual Conference Series):469–476, 1996.