Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

WUCSE-2003-62

A Thesis on Techniques for Non-Photorealistic Shading Using Real Paint - Master's Thesis, August 2003

Authors: Bailey, Reynold J.

June 27, 2003

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

A THESIS ON TECHNIQUES FOR NON-PHOTOREALISTIC SHADING USING REAL PAINT

by

Reynold J. Bailey, B.S. Computer Science & Mathematics

Prepared under the direction of Professor Cindy Grimm

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

May, 2004

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

ABSTRACT

A THESIS ON TECHNIQUES FOR NON-PHOTOREALISTIC SHADING USING REAL PAINT

by Reynold J. Bailey

ADVISOR: Professor Cindy Grimm

May, 2004
Saint Louis, Missouri

The goal of this research is to explore techniques for shading 3D computer generated models using scanned images of actual paint samples. The techniques presented emphasize artistic control of brush stroke texture and color. We first demonstrate how the texture of a paint sample can be separated from its color transition. Four methods, three real-time and one off-line, for producing rendered images from the paint samples are then presented. Finally, we develop metrics for evaluating how well each method achieves our goal in terms of texture similarity, shading correctness, and temporal coherence.

To my family

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis paper describes several techniques for shading 3D computer generated models using scanned images of actual paint samples. Figure 1.1 illustrates the basic objective of our research. We start with a lit 3D model. An artist provides an example of a shading change from dark to light in an image strip. This paint sample can either be scanned in from traditional art media or created using a 2D paint program. We then apply this user-defined shading style to the model, making it appear to be painted with the same technique.

Figure 1.1: Basic research goal.

When traditional artists compose a scene, they have complete control over the final image. They can alter the shading, brush stroke, and color as they see fit. They also work exclusively on a flat canvas, requiring some degree of skill and experience to convey believable lighting. On the other hand, computer graphics programs are very good at computing accurate lighting, but are not able to make artistic decisions about color choice or brush stroke texture. The techniques presented here address this disparity by letting the artist and software each do what they are best at. We seek to preserve the artist's freedom while harnessing the processing power of the computer to automate the time consuming details of applying the painting style to the model.

We present four techniques, each with a distinct set of advantages and drawbacks for rendering the model in the desired painting style. In order to quantify and compare these methods, we introduce a metric that qualitatively captures common texture distortions, shading errors, and frame-to-frame coherence in animation.

A discussion of previous work relating to this research is presented in Chapter 2. Background information on the fundamental concepts and existing techniques that we use in our research is given in Chapter 3. In Chapter 4 we demonstrate how to process paint samples to extract desired information. We show how processing paint in this way is useful, both for increasing artistic control and for improving our rendering techniques. In Chapter 5, we present our *object-based texture mapping* technique; the first of our four rendering methods. Our second technique, *image-based texture synthesis*, is presented in Chapter 6. Our third technique, *view-aligned 3D texture projection*, is presented in Chapter 7. Our final rendering technique, *view-dependent interpolation*, is presented in Chapter 8. Chapter 9 introduces our metrics and in Chapter 10 we compare the quality of each rendering method using these metrics. We also summarize the strengths and weaknesses of each technique. We conclude and discuss future work in Chapter 11.

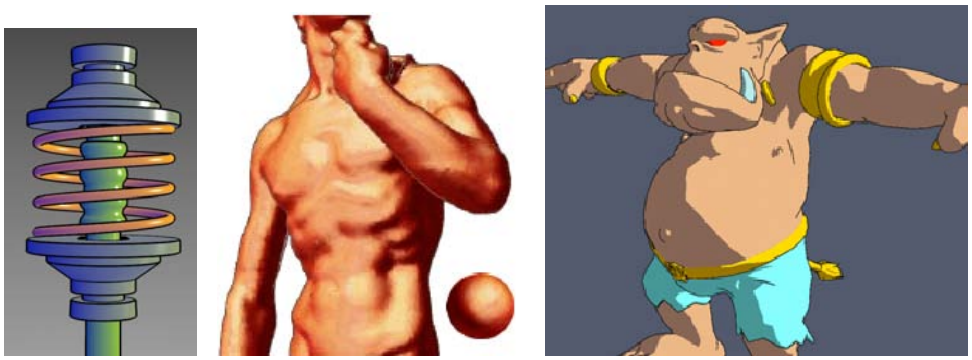
Chapter 2

Previous Work

The rendering techniques that we present can be categorized as *non-photorealistic rendering*. As the name suggests, non-photorealistic rendering refers to any technique that produces images in a style other than realism. There has been extensive research in the area of non-photorealistic shading.

Technical illustration shading [14] introduced the use of warm-cool color blends to enhance the perception of shape and orientation. This type of shading has been used extensively in technical manuals, illustrated textbooks and encyclopedias. The lit sphere approach [29] expands on this technique by providing a method to extract artistic shading models from actual paintings. This allows a wide range of effects from traditional painting to be reproduced. Unfortunately, much of the original brush texture is lost as the shading gradient is captured. Another popular shading style is cartoon shading [20]. The goal of cartoon shading is to render a 3D scene in a style which resembles that of traditionally animated films. These three techniques perform shading using color only and are illustrated in Figure 2.1.

Artists also make use of texture change across a surface to convey lighting. Several papers have addressed the technique of hatching [26, 35, 20]. Hatching is a drawing technique that uses groups of lines in close proximity to convey lighting, suggest material properties, and reveal shape. Another popular technique is stippling [8] which uses dots of varying density to convey shading changes. Creating a desired tone using either stippling or hatching is straightforward; strokes are added until the overall effect is dark enough. Other rendering styles that rely purely on texture change are charcoal rendering [21], engravings [24], and half-toning [13]. All of these systems use texture only and do not address the issue of color (which is reasonable for the particular mediums they mimic). Additionally, all of these techniques



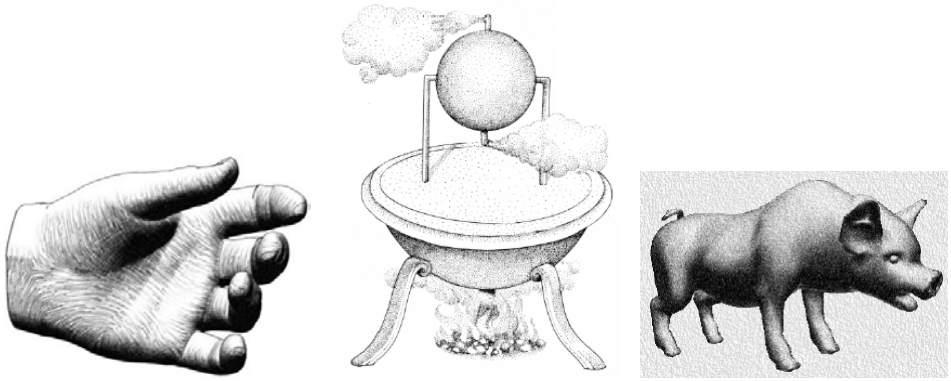
(a) Technical Illustration (b) Lit sphere (c) Cartoon shading

Figure 2.1: Color only techniques. (a) Courtesy of A. Gooch et al., A Non-Photorealistic Lighting Model for Automatic Technical Illustration. (b) Courtesy of P. Sloan et al., The Lit Sphere: A Model for Capturing NPR Shading from Art. (c) Courtesy of A. Lake et al., Stylized Rendering Techniques for Scalable Real-Time 3D Animation.

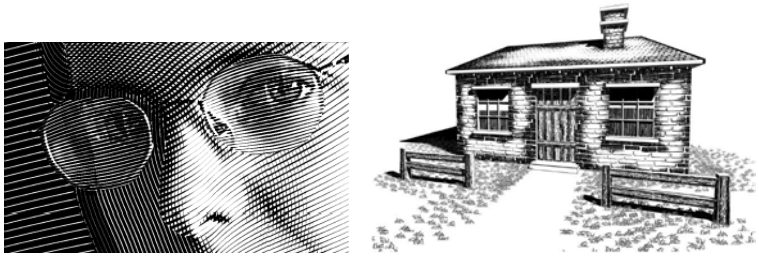
are very stylized and are therefore able to be captured procedurally or with minimal user input. We explore the use of less structured texturing styles, giving the artist more freedom over brush texture and color changes. These techniques are illustrated in Figure 2.2.

Stroke-based techniques such as the WYSIWYG NPR system [18] take a different approach by attaching paint strokes on the surface of the object. These strokes can convey fixed features of the model, or move over the surface in response to lighting changes. The idea of attaching paint strokes to a model is also used in painterly rendering [22]. This approach is particularly interesting because it completely automates the painting process by using various techniques to determine the position, orientation, size, color, and texture of the brush strokes. The system also maintains frame-to-frame coherence by reusing strokes throughout the animation. These stroke-based techniques are illustrated in Figure 2.3.

Finally, Webb et al. [33] show how to convey both texture and color change with shading. They use a volume texture approach that is based on the lapped texture method [25], to place several levels of texture onto a model. They then blend between these levels to give the illusion that the strokes are pinned to the model. Figure 2.4 shows a typical rendering obtained using this approach.

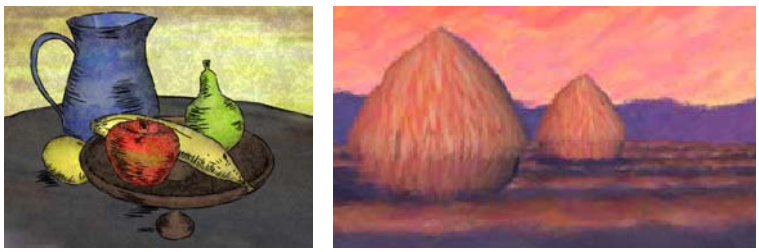


(a) Hatching (b) Stippling (c) Charcoal



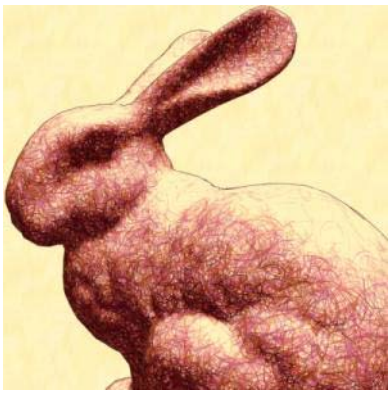
(d) Engraving (e) Half-toning

Figure 2.2: Texture only techniques. (a) Courtesy of E. Praun et al., Real-Time Hatching. (b) Courtesy of O. Deussen et al., Floating Points: A Method for Computing Stipple Drawings. (c) Courtesy of A. Majumder and M. Gopi, Hardware Accelerated Real-Time Charcoal Rendering. (d) Courtesy of V. Ostromoukhov, Digital Facial Engraving. (e)Courtesy of B. Freudenberg et al., Real-Time Halftoning: A Primitive for Non-Photorealistic Shading.



(a) WYSIWYG NPR (b) Painterly rendering

Figure 2.3: Stroke-based techniques. (a) Courtesy of R. Kalnins et al., WYSIWYG NPR. (b) Courtesy B. Meier, Painterly Rendering for Animation.



(a) Volume texture rendering

Figure 2.4: Technique that captures both color and texture with shading. (a) Courtesy of M. Webb et al., Fine Tone Control in Hardware Hatching.

In this paper, we present several alternative approaches to the problem of non-photorealistic shading. When artists paint an image in real life, they often must replicate identical brush strokes over and over so as to completely cover the surface they are trying to capture. We model this process using texture synthesis. The goal of texture synthesis is to create more texture that is perceptually similar (but not identical to) some input sample. Texture synthesis is not a perfect process since even the best known algorithms have cases in which they fail. We use a texture synthesis algorithm, based on the image quilting approach [9]. An explanation of this technique is given in Chapter 3.

Several papers [32, 34] address the problem of synthesizing texture directly on a model. These approaches require that the model be topologically well-structured. Our techniques operate on arbitrary models, and have different trade-offs of texture accuracy versus frame-to-frame coherency.

Chapter 3

Background

This chapter provides the background for this thesis. In Section 3.1, we present the fundamental concepts and general terminology used in our research. Additionally, our research makes use of (and in some cases, modify) existing techniques in order to achieve our goal. A discussion of these techniques is given in Section 3.2.

3.1 Fundamental concepts

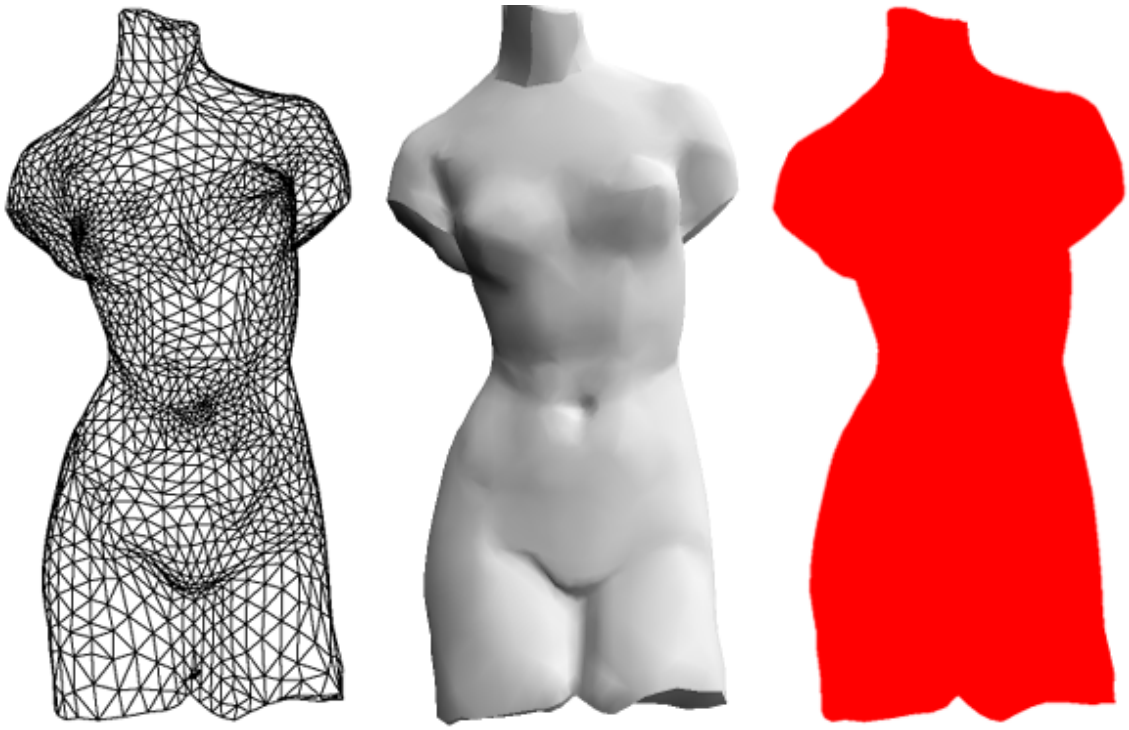
3.1.1 Models

The 3D models that we use were either obtained from 3D laser scans of real world objects or created using 3D modeling software. The models are stored as large text files containing the coordinates of all the vertices in the model along with information about how these vertices are connected. If we render this raw data, we obtain a wire-frame depiction of the model. This is known as the *mesh* of the model and reveals the underlying shape of the model. Figure 3.1(a) shows the mesh of a typical model.

3.1.2 Lighting

In order to accurately shade a given mesh, we must perform lighting calculations. For the examples presented in this paper, we used a single directional light source and computed the amount of light at every vertex as:

$$s = \frac{1 + \vec{N} \cdot \vec{L}}{2} \tag{3.1}$$



(a) Raw mesh (b) Shaded mesh (c) ID buffer

Figure 3.1: Venus de Milo: raw mesh, shaded mesh, and ID buffer

where $\vec{N}$ is the surface normal and $\vec{L}$ is the light vector. Multiple lights or more sophisticated lighting algorithms may also be used. The possible values for s range from 0 to 1. The model's assigned color is scaled by this value to result in a shaded model. Figure 3.1(b) shows an example of a shaded model.

3.1.3 ID buffer

We assign a unique identifier to every model in a scene. These identifiers are stored in an *ID buffer*. ID buffers are used primarily for improving efficiency. We use it as a lookup table to quickly determine which model in the scene a given pixel belongs to. This improves the performance of our rendering techniques. If we assign a unique color to each identifier and render the contents of the ID buffer, we obtain an image showing how the scene is segmented. Figure 3.1(c) demonstrates this for a simple scene consisting of a single model.

3.1.4 Object-based operations vs. image-based operations

At present, almost all affordable computer display devices are limited to 2D images. Displaying a 3D object in 2D is accomplished by projecting the 3D object onto a 2D plane. Numerous projection techniques exist [12]. Figure 3.1.4 illustrates the basic principles of performing a projection. Throughout this thesis, we refer to object-based

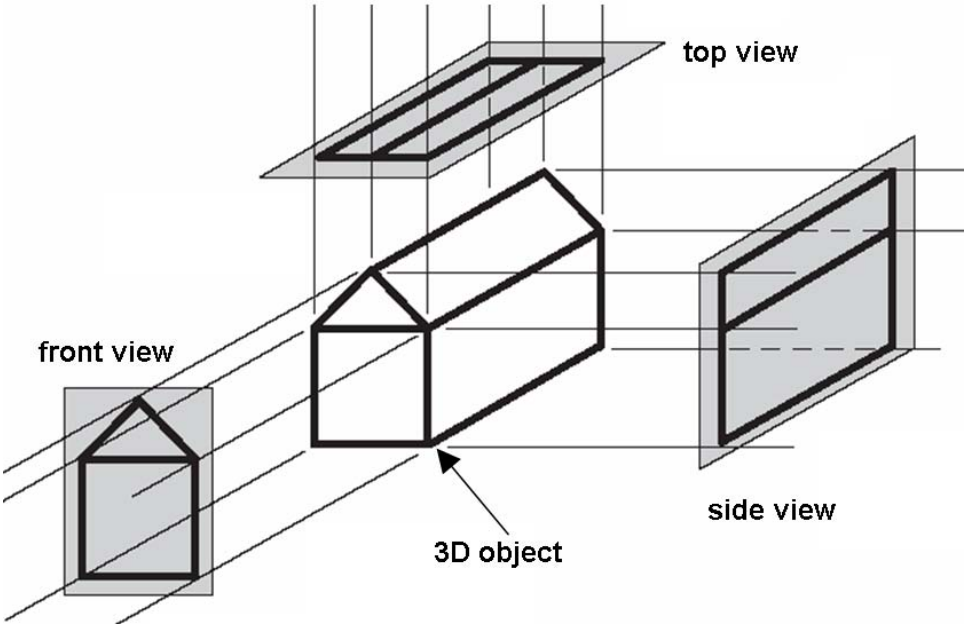


Figure 3.2: Projecting a 3D object onto a 2D plane. Courtesy of J. Foley et al., Computer Graphics: Principles and Practice.

operations and image-based operations. Understanding the distinction between them is important. Object-based operations refer to operations that use and manipulate the properties of the 3D object before the projection takes place. Image-based operations are those performed on the 2D image that results from the projection.

3.1.5 Color

We represent color using the RGB (Red-Green-Blue) color model. The RGB color model is used in color CRT monitors. A given color is expressed as (r, g, b) where the values are normalized. That is, all color values are restricted to the range of 0 to 1 inclusive. The RGB color model is additive, meaning that the values represent the contribution of red, green and blue respectively. The RGB color space can be visualized by plotting the values of red, green, and blue in a Cartesian coordinate system as shown in Figure 3.3.

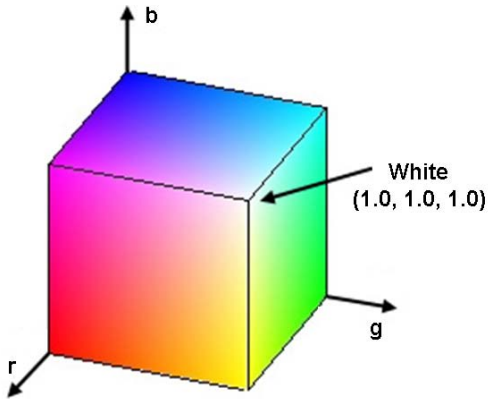


Figure 3.3: The RGB color space.

3.1.6 Paint

Figure 3.4 shows a scanned image of a typical paint sample used in our research. Paint samples have two distinct properties: a color transition and a brush texture.

Figure 3.4: Example of a typical paint sample.

The color transition is the global color change across the sample. This change is more correctly referred to as the color trajectory, as it defines a path through color space. Brush texture can be described as local variations within the color trajectory. In Chapter 4 we describe our method for separating the texture of a paint sample from the color trajectory. For rendering purposes, we use the convention that a transition from left to right across the sample represents a change from dark to light.

3.2 Techniques

3.2.1 Texture mapping

Texture mapping [6, 3, 15] is a popular technique used in computer graphics to add detail to a scene. The basic idea of texture mapping is to map a texture image onto some surface. In many computer games, for instance, the surface representing the

terrain is texture mapped with images of grass and rocks. Likewise, the surfaces used for walls are texture mapped with images of bricks. This adds to the realism and overall visual appeal of the game. Additionally, even low-end commodity graphics cards are now equipped with texture mapping capabilities allowing texture mapping to be performed without any degradation in rendering time. Figure 3.2.1 illustrates the texture mapping process. Once a 2D texture pattern is obtained, it is stored as

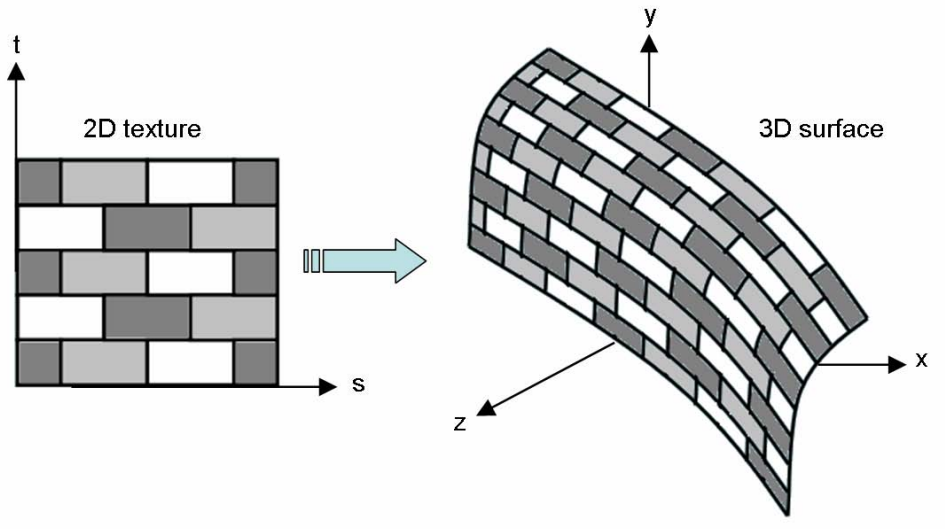


Figure 3.5: Texture mapping. Courtesy of J. Foley et al., Computer Graphics: Principles and Practice.

an $n \times m$ array of texture elements or *texels*. The variables s and t are used to index the rows and columns respectively of this array. These variables are called *texture coordinates* and are usually scaled to values in the interval [0,1]. Texture mapping associates a unique point from the texture with each point on the 3D surface. The rendered image will appear as if the texture pattern is "glued" to the surface.

Despite the obvious benefits of texture mapping, there are some difficulties that must be addressed. The situation often arises where the surface that we are mapping to is larger than the texture pattern. There are several approaches that may be used to solve this problem. One option is to stretch the pattern to fit the surface. This stretching may be undesirable as it can cause the pattern to become distorted. Another approach is to use multiple copies of the texture to cover the surface as in Figure 3.2.1. This works well for textures that are tileable. Tileable textures have the property that multiple copies can be placed side by side without revealing seams where two copies meet. Tileable textures are discussed in Section 3.2.3. A

third approach is to generate a larger texture pattern through texture synthesis. A discussion of texture synthesis is presented in the Section 3.2.2.

Another problem that is encountered with texture mapping is that of mapping a rectangular texture pattern onto a non-planar surface such as a sphere. This results in the texture being distorted. To add to this problem, the resulting texture mapped 3D surface must be projected onto a 2D plane to be displayed as explained in Section 3.1.4. This projection causes further distortion to the texture pattern. Solving this problem requires the use of complex mapping functions.

3.2.2 Texture synthesis

Texture synthesis refers to any process that generates texture that is perceptually similar (but not identical to) some input sample. Figure 3.2.2 illustrates the texture synthesis process. Early texture synthesis algorithms [16, 4] captured statistical mea-

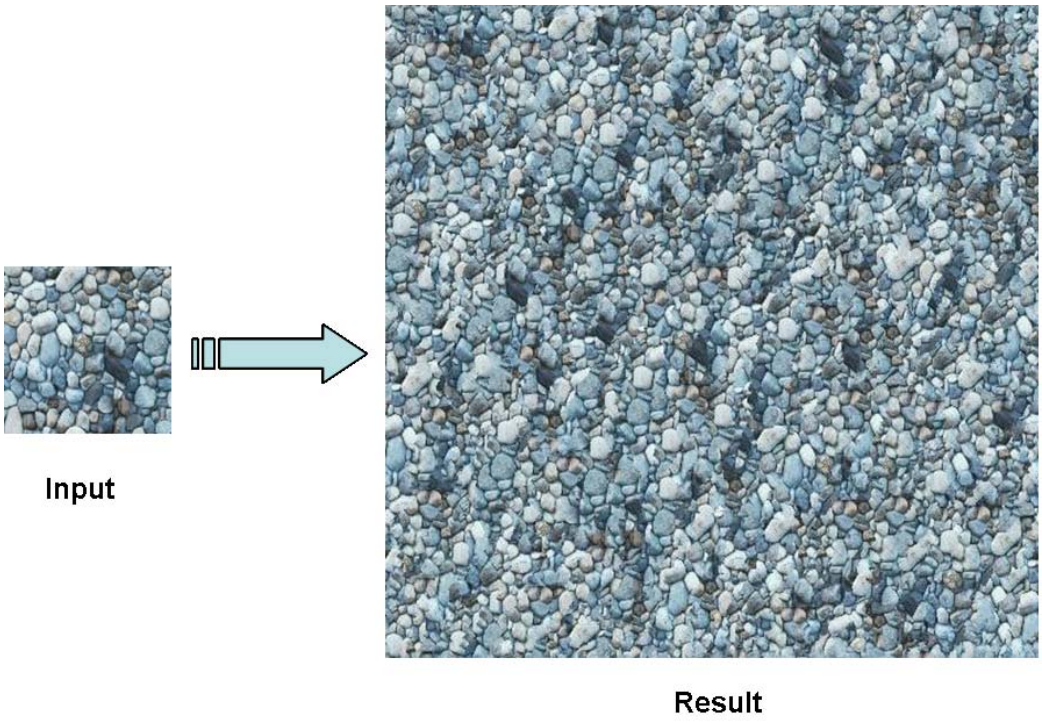
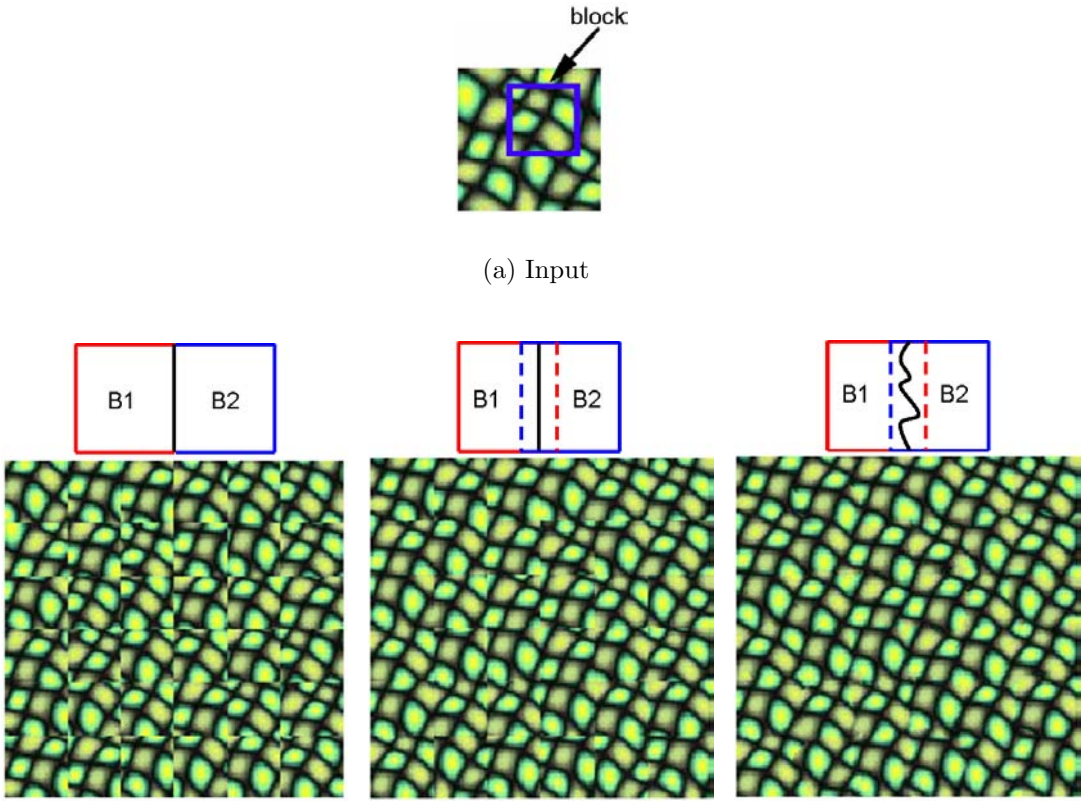


Figure 3.6: Texture synthesis. Courtesy of M. Ashikhmin, Synthesizing Natural Textures.

surements from the input texture and used it as a guide for the synthesis process. The resulting texture was generated in order to maintain the statistical properties that were present in the input texture. This approach worked well for stochastic textures

but was not suitable for more structured textures. Later approaches [10, 2] performed synthesis one pixel at a time. This was typically done in raster scan order (left to right, top to bottom). In general, these approaches determine the color for a given pixel (called a target pixel) in the resulting texture by searching the input texture for a suitable candidate pixel and copying its color. The search for a candidate pixel proceeds by searching for a region in the input texture that most closely matches the region around the target pixel that has already been synthesized. The candidate pixel is then chosen from this region of the input texture. While these approaches obtained excellent results for a wider range of textures, they were computationally expensive. Every synthesized pixel in the resulting image required that its neighboring pixels be compared to every possible region in the input image. More efficient and simpler texture synthesis techniques eventually emerged. One such technique, which we use extensively in our research is image quilting [9].

The image quilting technique generates a resulting texture by stitching together small square patches of the input texture. This approach is more efficient because entire patches of texture from the input are copied and pasted into the resulting image instead of a pixel by pixel transfer. Figure 3.7 illustrates how this is done. Figure 3.7(a) shows the input texture and also highlights a block that has been selected. This block will be used as a patch in the quilting process. Blocks are selected in order to meet certain constraints. Typical constraints include ensuring that texture features such as edges are maintained across patches and ensuring that the difference between the average color of neighboring patches is within some threshold. Additional constraints can be introduced to make the synthesis process more specific. Figure 3.7(b) shows the results of simply placing the patches side by side in a grid pattern. As expected this leaves visible seams where two patches meet. A better approach is to allow the patches to overlap each other. An additional constraint that the overlapping regions of two patches have some degree of similarity is necessary. The use of these overlapped patches in Figure 3.7(c) shows a clear improvement over the naive side by side patch placement. This reduces the visual discontinuity but does not eliminate it. A second pass over the image processes these overlapped regions and performs a *minimum error cut*. The minimum error cut works by finding the path through the overlapped regions that minimizes the the total error. This path produces the optimal irregular boundary between the patches and further minimizes



(a) Input

(b) Grid-like placement (c) Overlapped placement (d) Minimum error cut

Figure 3.7: Image quilting. Courtesy of A. Efros and W. Freeman, Image Quilting for Texture Synthesis and Transfer.

discontinuities. Figure 3.7(d) shows the results of using a minimum error cut. Computing the minimum error cut can be done using dynamic programming or by using Dijkstra's algorithm [7].

3.2.3 Generating a tileable texture

In some cases we require that a given texture be tileable. A tileable texture is one that has a seamless appearance if multiple copies are placed side by side. The problem of creating tileable textures is not easily solved by adding constraints to the image quilting technique because of the raster scan order in which the blocks are placed. We instead, utilize a simpler masking technique to achieve the desired result [5]. Given a square input image $\mathbf{T}$ of size $n \times n$ pixels, we first generate a shifted image $\mathbf{T}'$ as

follows:

$$\mathbf{T}'[x][y] = \mathbf{T}[(x + n/2) \bmod n][(y + n/2) \bmod n]$$

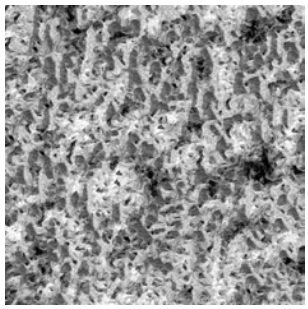
This shifts the edges of the image to the center, revealing the vertical and horizontal seams it produces when tiled. The pixels that were in the center of $\mathbf{T}$ are also now on the edges of $\mathbf{T}'$ so it can be tiled. Next, we use the following mask to blend the center of $\mathbf{T}$ with the edges of $\mathbf{T}'$. A circular mask works well in our case:

$$\mathbf{M}[x][y] = 255 - 255 \cdot \frac{\sqrt{(x - n/2)^2 + (y - n/2)^2}}{n/2}$$

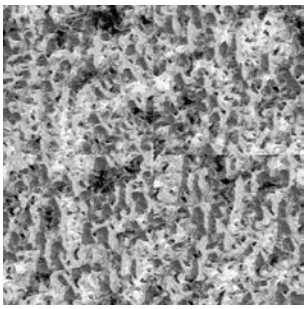
We clip the values of $\mathbf{M}$ to the interval $[1, 255]$, and generate $\mathbf{M}'$ as we did $\mathbf{T}'$. The output image $\mathbf{O}$ is finally computed as:

$$\mathbf{O} = \frac{\mathbf{T} \cdot \mathbf{M} + \mathbf{T}' \cdot \mathbf{M}'}{\mathbf{M} + \mathbf{M}'}$$

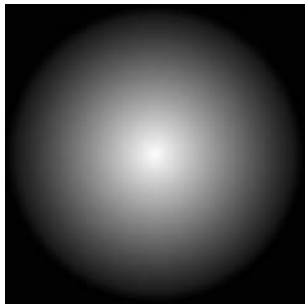
Figure 3.8 summarizes the process and demonstrates how the resulting texture can be tiled without any visible seams.



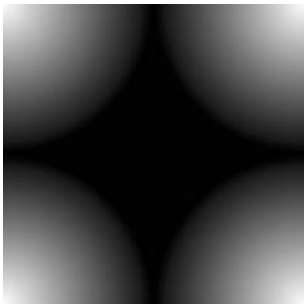
(a) **T**: input texture



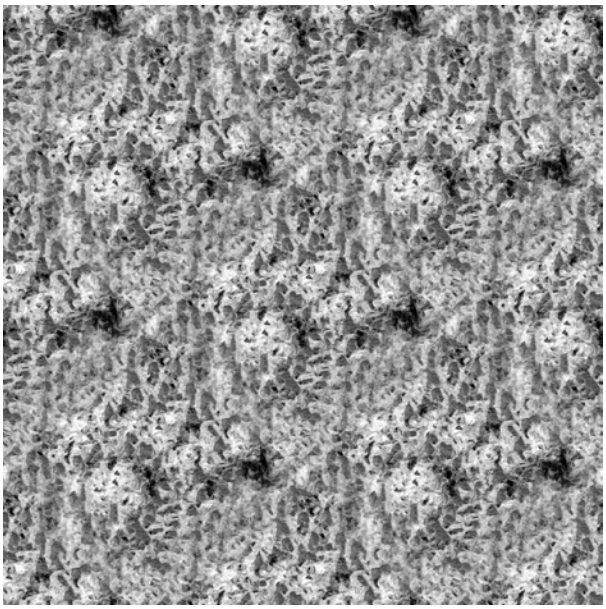
(b) **T′**: shifted texture



(c) **M**: circular mask



(d) **M′**: shifted mask



(e) **O**: result tiled twice in each direction without any seams

Figure 3.8: Generating a tileable texture level.

Chapter 4

Paint Sample Processing

In this chapter, we describe our technique for processing a user provided paint sample. Paint processing is necessary for extracting information that will be used by our rendering techniques. Additionally, paint processing is also beneficial in that it provides an avenue for increasing artistic control. We first show how a smooth color trajectory can be extracted from a paint sample in Section 4.1. In Section 4.2 we show how the texture of the sample can be treated independently of the color trajectory. This allows the user to modify the color trajectory while preserving the texture features of the sample.

4.1 Color trajectory extraction

Consider the paint sample shown in Figure 4.1(a). If we plot the color values of each pixel of this sample in RGB color space, we obtain Figure 4.2(a). Notice that this plot reveals a rough shape of the color trajectory.

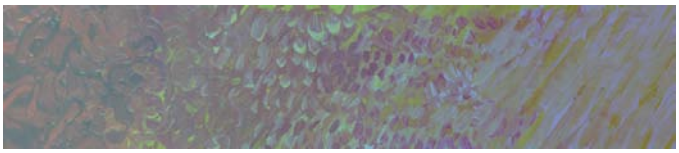
To extract an initial color trajectory from a given sample, we simply average the colors of each pixel column of the sample image. This effectively filters the 2D sample into 1D image strip that represents a path through color space. Unfortunately, the result contains a fair amount of streaking due to local texture variation. We stretch the 1D image vertically to better illustrate the streaking as shown in Figure 4.1(b). We run the following recursive algorithm on this trajectory's set of RGB points in order to sort the colors into a smooth, continuous gradient.

We first seed the algorithm with the two endpoints of the unsorted trajectory. Given two colors A and B of the trajectory in RGB space, we let M be their midpoint. We search for the point C that is closest to M, but contained in the sphere of the

(a) Red to yellow paint sample

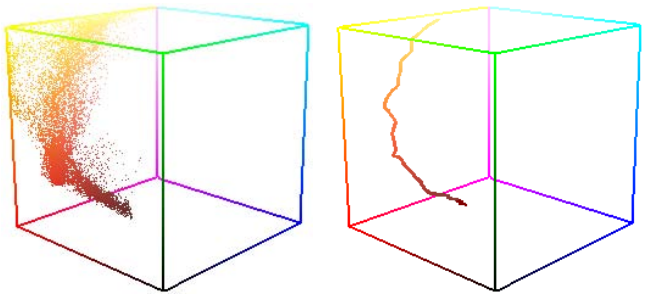
(b) Color trajectory before sorting

(c) Color trajectory after sorting



(d) Extracted texture

Figure 4.1: Extracting the color trajectory and texture from a typical paint sample.



(a) Color distribution (b) Trajectory

Figure 4.2: Color distribution and non-linear trajectory for the paint sample in Figure 4.1(a).

diameter AB (see Figure 4.3). If such a point is found, the algorithm runs recursively on A and C and on C and B. If no such point is found, we know that A and B are close enough to be considered neighbors and the recursion stops by adding A and B into a linked list representing the sorted color trajectory. The output trajectory may contain fewer colors than the input because samples which deviate too much from the path are removed. Simple linear interpolation can stretch the output gradient back to the size the input if necessary. Figure 4.1(c) shows how the output of the algorithm has effectively removed the streaking effect while maintaining the shape of the color trajectory we wanted to extract. Figure 4.2(b) shows the resulting smooth trajectory. It is important to note that the resulting trajectory is non-linear. Most existing computer based techniques represent color transitions using some form of a linear blending. This is unnatural in practice and clearly does not model how real paint behaves. An extracted color trajectory, using our approach, is a more realistic representation of the behavior of actual color blends.

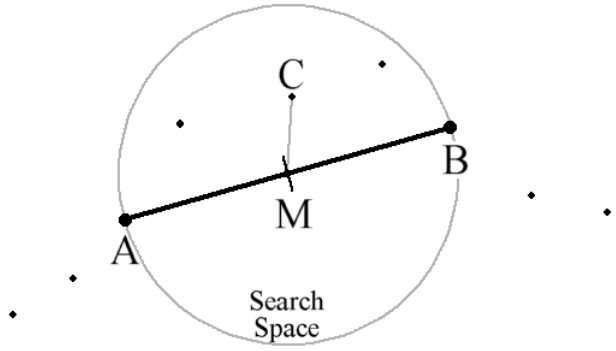


Figure 4.3: Searching for a sample point halfway between A and B.

For trajectories of high curvature, our sorting algorithm may reject too many points and clip the most curved section from the output. We compensate for this by expanding the search area for C by a user-defined scale factor. The output still may have a different rate of color change because of discarded color segments. We correct for this by allowing the artist to use a simple click-and-drag interface to control the speed of the color trajectory.

4.2 Separating color and texture

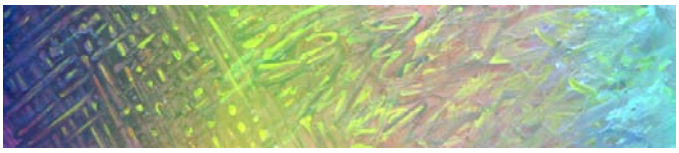
As explained in Chapter 3 texture can be viewed as local modulation within the color trajectory. We extract this local modulation by subtracting the color trajectory from

each pixel row of the original paint sample. This produces an intermediate difference image with RGB values ranging from -1 to 1 (Figure 4.1(d)).

The advantage of this separation is that we can then add an arbitrary color trajectory back into the difference image (clipping any RGB values that fall outside of the 0 to 1 range) to obtain a sample with different colors but similar texture. While this approach is purely heuristic, it does in fact achieve the desired separation reasonably well. Figure 4.4 shows a paint sample being modified to produce a much more creative color blend by a user-specified path through color space. The approach is not perfect. Indeed, we can observe some hints of green in the recreated blend that may be undesirable, but the main stroke features are preserved which is our main objective. This approach allows an artist to create a wide range of textured blends from a single initial paint sample. The separation of the color trajectory and texture from a paint sample is also useful for rendering as described in the following chapters.

(a) Input paint sample

(b) User specified color gradient



(c) Resulting user created “paint sample”

Figure 4.4: Changing the color trajectory of a paint sample without changing its texture.

Chapter 5

Object-Based Texture Mapping

The idea behind our object-based texture mapping technique is to dynamically determine the texture coordinates (s, t) for every vertex in the model. Since we are using the convention that moving from left to right across the texture corresponds to a change from dark to light, it means that the s coordinate, which represents the horizontal index into the texture, caries shading information, and must be updated every frame to reflect lighting changes. The t coordinate, the vertical index into the texture, does not affect the shading, but rather impacts the perceived texture coherence. We want vertices that are close to each other in the model to be mapped close together in the texture. Once we have chosen a set of t values for a model, we keep them fixed, and allow only the s coordinate to change. We do this because we want the texture to remain consistent from frame to frame.

In order to assign the s texture coordinate for a particular vertex, we determine the the amount of light received at the vertex as explained in Section 3.1.2. We use the resulting value as the s texture coordinate.

To assign t values to the vertices, we start by picking a random t for a randomly selected vertex v. Next, we place all the neighbors of v into a queue. We keep processing from the front of the queue. For every vertex that leaves the queue, we compute its t value based on the closest neighbor that has already been processed. We queue all unprocessed neighbors of the current vertex and repeat the process until the queue becomes empty.

The actual calculation for forwarding t values to a new vertex is as follows:

$$t_v = t_{\text{nearest}} + \frac{\delta y}{\Delta y}\beta + \gamma \tag{5.1}$$

where t_v is the t value currently being computed, t_{nearest} is the t value of its nearest neighbor, δy is the difference between the y coordinates of the two vertices and Δy is the maximum y difference for the entire mesh. The choice of the y axis is arbitrary. A user specified vector field could be used as in [25] to change the relative orientation of the texture mapping. β is the number of times the texture repeats over the object and γ is a random value that introduces some statistical variation.

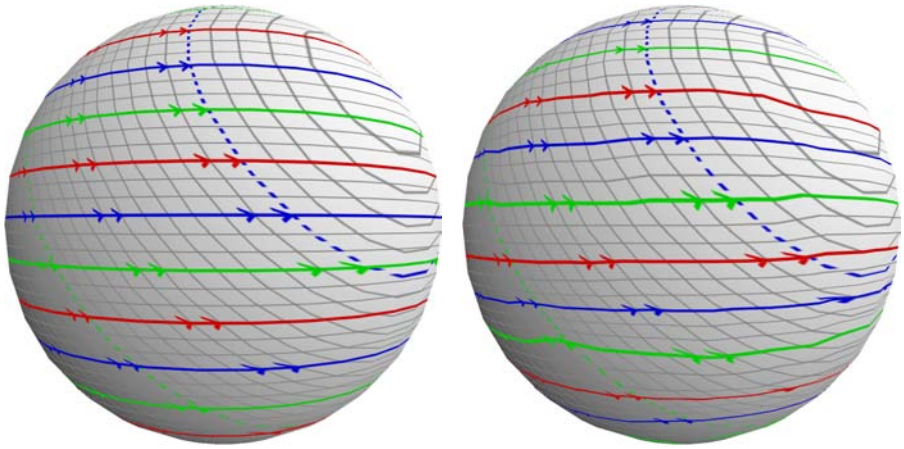
Figure 5.1 illustrates our general method for texture application. We use a simple grid to show how parameters can be adjusted to obtain desired results. The sphere in Figure 5.1(a) shows the direct mapping of 4 strips of the grid with no variation ($\gamma = 0$). Figure 5.1(b) and 5.1(c) show the mapping of 4 strips of the grid with increasing variation in γ. Notice that in all cases the dotted blue line, representing a line of constant lighting intensity, remains fixed. Figure 5.1(d) shows the application of an actual paint blend.

Some problems arise if the mesh has seams. We can handle the problem in one of two ways. We can identify the unprocessed disconnected vertex that is closest to a processed vertex and simply forward that t value over. This is most useful in cases where the seam falls within a region that should be continuous. The other approach is to ignore the seam and generate a new random t value for a random unprocessed vertex in each disconnected piece of the mesh. This tends to highlight the seam.

The calculation of the t values occurs only once, whereas the s values are updated each frame. This guarantees that the scene correctly reflects changing lighting conditions and that paint strokes stay consistent within a single frame and coherent from frame to frame.

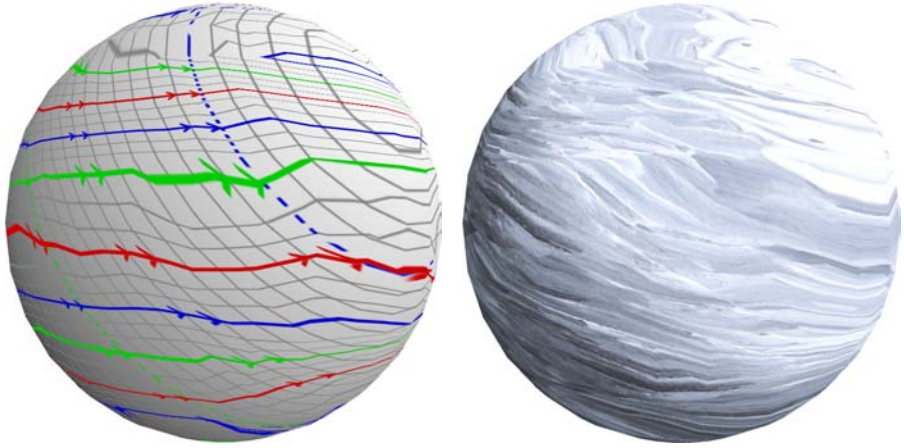
The main advantage of the object-based approach is the fact that it runs in real-time. We only need to perform the random texture coordinate generation once at startup, and can reuse the results for every subsequent frame. Since we only use basic texturing capabilities present on any 3D capable graphics hardware, this method is perfectly suited to real-time applications. Additionally, temporal coherence is achieved because the spatial texture coherence is established separately from the shading calculations and is carried over from one frame to the next.

There are, however, some serious drawbacks to this method. Most importantly, the texture quality is unstable under changing lighting conditions. Figure 5.2 illustrates how the quality of the resulting texture diminishes as the light in the scene is repositioned. When the lighting direction is parallel to the direction that the paint sample is applied we obtain excellent texture quality. This is so because the shading



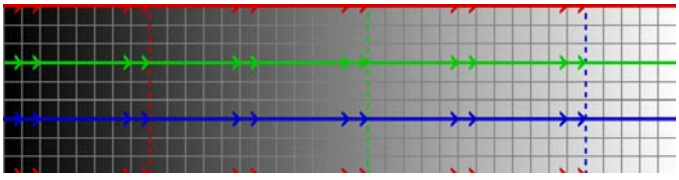
(a) No perturbation

(b) Small perturbation



(c) Large perturbation

(d) Same parameters as 5.1(c)



(e) Grid texture to test continuity and alignment

Figure 5.1: Object-based texture application

Figure 5.2: Texture distorts as direction of light changes

change across the model matches the left-to-right user specified shading across the paint. As we increase the angle between the lighting direction and the direction that the texture is applied, the texture becomes distorted in order to maintain the correct shading across the model. The texture is completely lost when the lighting direction is perpendicular to the direction that the paint is applied. One possible, but rather complicated and inefficient, method for fixing this problem is distort the texture before it is applied to the model in such a manner that the amount of distortion caused by the lighting cancels the texture distortion.

Other problems with this approach are also encountered. Since texture coordinates can only be assigned to vertices, the algorithm has little control over meshes that are sparsely tessellated. In these cases, the texture may be noticeably distorted or stretched. Subdividing the mesh only helps for non-planar objects. This is because a flat surface typically receives a constant amount of light. Figure 5.3(a) illustrates this fact. Notice that the shading varies across the teapot but is constant on the table top. This means that all the vertices for the table top will be assigned the same s texture coordinate, which effectively hides the texture and loses the "painted look". The image obtained by applying paint to both the teapot and the table is shown in Figure 5.3(b).

(a) Shaded scene (b) Painted scene

Figure 5.3: Rendering flat surfaces.

Chapter 6

Image-Based Texture Synthesis

In our image-based texture synthesis approach, we seek to generate and apply paint to the region covered by the model in image space while preserving the correct shading of the scene. We modify the image quilting algorithm presented in Section 3.2.2 in order to achieve this goal.

Figure 6.1 illustrates the image-based texture synthesis technique. We start by rendering a grayscale shaded image, as explained in Section 3.1.2, that will serve as a guide for the synthesis. Since we are letting the artist make decisions about color and texture for the final appearance of the model, the fact that illumination is only computed in levels of gray is not a problem.

The advantage of using the image quilting algorithm to perform the texture synthesis is that we can control the appearance of the output by placing additional constraints on which blocks we pick. In particular, we wish to constrain the search area for a block to a region of the paint sample that corresponds to the correct shade value. This is done by adding the constraint that the x coordinate of the block must be no more than k block sizes away from $x_0 = L \cdot W_{\text{sample}}$, L being the average light value of the block in question, W_{sample} being the width of the paint sample image and k being a texture dependent parameter ($k = 4$ worked best for us in most cases).

We made another important modification to the image quilting algorithm. As more constraints are added to the algorithm, it is more likely that the underlying block structure becomes visible due to the increased difficulty in finding a suitable block. Efros and Freeman, the creators of the image quilting technique, suggest running the synthesis algorithm multiple times, decreasing the block size at each step, and trying to match the previous level as much as possible. This increases the computation time

significantly yet does not completely solve the problem because regions where the shading changes sharply still appear blocky.

We are able to take advantage of our knowledge of the data to solve this problem differently. In particular: it is more important to capture the correct lighting changes than it is to capture the correct texture changes. This is especially true for small objects within a scene where shading conveys more important information about shape and orientation than texture does. Getting the color gradient right is therefore more important than getting the texture to change along with the lighting. We generate the lighting component of the image by looking up the color gradient with the shade value at each pixel. We can then synthesize the texture separately using only the texture difference image: the paint sample minus the color gradient as described in Section 4.2. Figure 6.1(d) shows a rendering using only the color component of the paint sample in Figure 6.1(b) while Figure 6.1(e) shows a rendering using only the texture component. We recombine these two images by adding the RGB values together pixel by pixel, clipping any overflows that occur (see result in Figure 6.1(f)). This effectively removes the blocking artifacts without increasing the processing time because we can use only one pass of a relatively coarse block size for texture synthesis. Another benefit of this separation is that we can let the artist manipulate the color gradient in real-time after the texture has been synthesized, thereby increasing artistic control.

Texture synthesis is only performed on the regions of the image that have the correct ID value in the ID buffer (see Section 3.1.3). Blocks that cover regions without any such valid pixels can be skipped, which speeds up the image generation process. The ID buffer for our example is shown in Figure 6.1(c).

Finally, we address the issue of creating animations with this method. Naively resynthesizing each frame from scratch produces a shower-door effect [17] (the model looks as if it was viewed through a shower-door). To improve temporal coherence we add an additional constraint: we require each block to match the previous frame as much as possible (computed as a squared pixel difference error of the synthesized texture image). For small lighting or camera movements, this added constraint works very well at keeping texture coherent over time. The shower door effect is not completely eliminated, but is reduced to an unobtrusive level. Naturally, for paint samples that exhibit drastic texture variations this constraint will make it impossible for the synthesis to find suitable blocks after a few frames. There is no way for the synthesis to turn a hatch mark into a curve, for example. For these more difficult cases, we

(a) Shaded mesh (b) Paint sample (c) ID buffer

(d) Color only (e) Texture only

(f) Color and texture

Figure 6.1: Rendering created using image-based texture synthesis.

must rely on blending to improve coherence. We synthesize an entirely new set of texture every nth frame and blend texture values between these keyframes while recomputing the shading at every frame. Again, the separation of color transition from texture is very beneficial.

Stroke density in this method is directly related to the stroke density in the original sample. Therefore the ratio between image resolution and paint sample resolution is significant. Simply rescaling the input paint sample is sufficient to achieve a different stroke density. Since the image generation happens off-line, the target resolution is known ahead of time and the paint sample can be prepared accordingly.

The off-line nature of this algorithm is its main disadvantage. Rendering takes between 20 seconds and a minute depending on image resolution. The quality of the results produced, is however, very good (this approach produces the best individual frame quality of the four techniques). The underlying shading of the model is captured and the overall painting style is consistent with that of the input sample. The two following chapters describe alternatives that run in real-time on commodity graphics hardware.

Chapter 7

View-Aligned 3D Texture Projection

Recent advances in graphics hardware technology has allowed for the use of *volume textures*. A volume texture is simply a stack of 2D textures. A third texture coordinate, r is used in addition to s and t to perform a lookup in the 3D texture space. Since their introduction, volume textures has been used for MRIs and other scanning programs. In our view-aligned 3D texture projection technique, we make use of the volume texturing capabilities of GeForce 4 class graphics cards that are capable or storing 3D textures up to size 512×512×512.

This approach uses texture synthesis only as a preprocessing step. We divide the input paint sample into 8 sections of roughly constant shade level. We generate larger versions (512×512) of each section with image quilting. Consider the paint sample shown in Figure 7.1. Figure 7.2 shows the resulting 8 synthesized blocks.

Figure 7.1: Input sample.

We found that generating 8 levels was adequate for the particular size of our paint samples given that this is about how often the texture changes. We experimented with generating more levels and with trying to keep stokes coherent from level to level, but observed no substantial gain. In order to keep texture information separate from

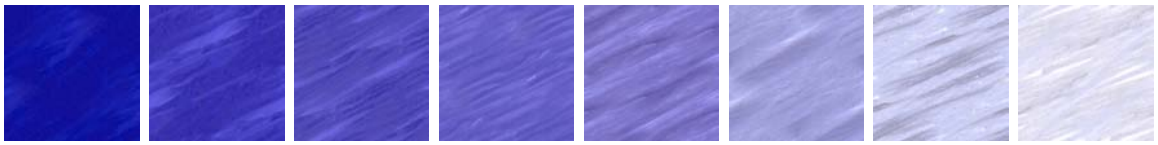


Figure 7.2: Resulting synthesized images.

the color gradient, we subtract the average color of each section and only synthesize a texture difference image. We store this difference image in a regular bitmap by mapping the interval $[-1, 1]$ linearly to $[0, 1]$. Once this is done we make a second pass over the textures to guarantee that they are tileable as explained in Section 3.2.3.

For rendering, we start by creating a 3D texture from each of the synthesized levels by stacking them in order of increasing shade level [33]. Figure 7.3 illustrates the 3D texture. We access the 3D texture, expanding the value back to the interval

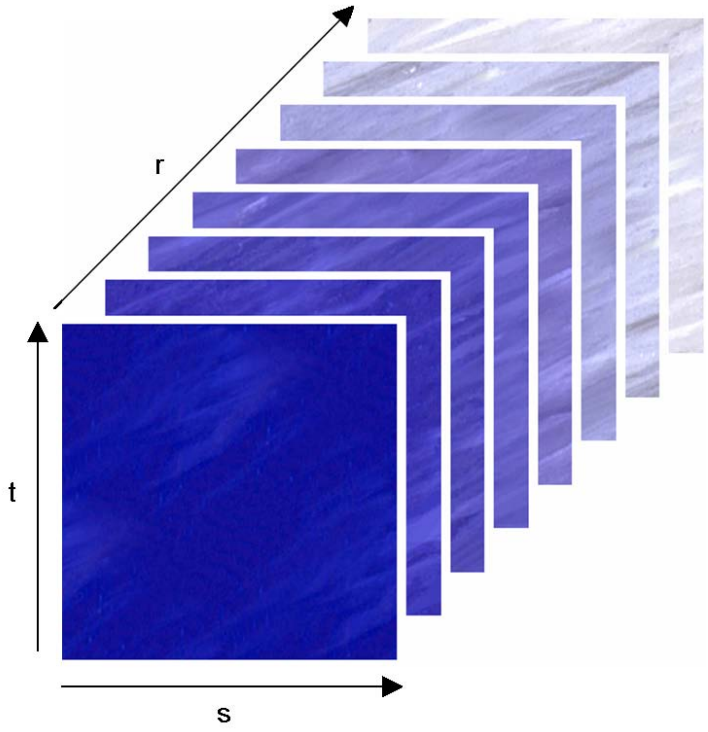


Figure 7.3: 3D texture.

$[-1, 1]$ and adding a color gradient indexed by the light value. The s and t texture coordinates are generated by mapping the horizontal and vertical screen coordinates to the interval [0,511] respectively. We use the lighting value mapped to the interval [0,7] as the r texture coordinate. The hardware automatically blends between the

texture levels for values of r that are not whole numbers. The (s, t, r) triplet is generated automatically by a vertex shader. Stroke density can be adjusted by a simple scale factor on s and t. This is where the advantage of having a tileable texture comes in, as no seams are visible when the texture repeats over the image. Figure 7.4 shows a typical rendering using this approach.

Figure 7.4: Model of a bunny rendered using 3D textures.

In order to avoid the impression that the texture is fixed to the screen and that the mesh is "sliding" through it, we keep track of an offset in s and t that we adjust when moving the model. We increment this offset by the average screen space displacement of the vertices most directly facing the camera. This gives the illusion that the texture follows the movement of the object, at least for the polygons that occupy most of the screen space. It is impossible to perfectly move the texture along with the mesh since it is attached to the view plane, but this approximation increases visual coherence.

The view-aligned 3D texture projection method is very attractive because it can almost match the quality of the image-based texture synthesis technique, but runs in real time. The hardware does, however, introduce error when interpolating across levels in the 3D texture. Frame-to-frame coherence is again only approximated

by trying to shift the texture in the view plane to match the motion of the model on screen. The main disadvantage to this approach is, however, the preprocessing time. Synthesizing 8 512×512 textures then processing each of them to ensure that they are tileable may take as long as 15 minutes depending on the block size used during texture synthesis.

Chapter 8

View-Dependent Interpolation

Our view-aligned 3D texture projection technique presented in Chapter 7 automatically blends between the levels of texture for shade values that are not whole numbers. This happens for every possible view of the model, which in some sense means that every view is equally important. The situation may arise, however, where we desire that certain views of the model have a specific texture instead of blending between textures. Our view-dependent interpolation technique achieves this goal.

The basic idea behind our view-dependent interpolation technique is to assign specific textures to the important views of the model and perform blending between these textures for all other views. Determining which views are important is left to the discretion of the user. The only restriction we impose is that every face in the mesh of the model must appear in at least one of these views. This is necessary to ensure that there are no gaps (un-rendered faces) in the resulting image.

Typically, between 12 and 15 views which surround the model are selected. We must determine which faces of the model's mesh are in each view. To do this, we center the mesh in the view by automatically adjusting the zoom and panning the camera until the mesh is centered and as large as possible. We then project the mesh onto the 2D plane of the current view and record which faces are present. We also record the projected locations (s, t) of each vertex of each face in the 2D plane. This will be used as the texture coordinates when the actual texture is applied.

For any reasonably complicated model, there will be portions of the model that are occluded or partially occluded for a particular view. This means that two or more faces overlap when projected. We want only the faces that are closest to the projection plane to be recorded. We handle this issue by processing the overlapped

faces to determine their depth-ordering and removing any face that is covered from the list of faces for that view.

The above approach can leave a face uncovered if there is no view for which that face is un-occluded. If this is the case, we mark all of the faces that are covered by the current view and generate another set of views, this time with only the faces that were not covered in the first pass. (We only need to keep the subset of these new views that actually contain visible faces in the uncovered subset.)

Once we have recorded all the necessary information for each view, we use texture synthesis to create a 3D texture as explained in Chapter 7. The 3D texture will be comprised of n 2D textures where n is the number of views selected by the user. For rendering a given view of the model, we first obtain the list of faces present in that view. For each of these faces, we know which subset of the textures covers that face and also the texture coordinates of the face in a particular texture. We assign a weight to each of these textures based on how much the viewing direction of that texture differs from the current viewing direction (we assign the highest weight to the texture that has a viewing direction that most closely matches the current viewing direction). We then use these weights to blend between the textures.

As we animate our scene and move from viewing direction v to viewing direction $v + 1$, there is a subset of the faces that is visible in both views. To ensure frame-to-frame coherency, we copy the resulting texture from the faces in v to the corresponding faces in $v + 1$. This provides a smooth transition but may result in some texture distortion as a face in view v will not look exactly the same in view $v + 1$ because of the curvature of the model.

The view-dependent interpolation technique runs in real time, and maintains excellent frame-to-frame coherence since the texture is pinned to the surface of the model. There is however a loss of texture quality because the texture is distorted to fit the contours of the mesh.

Chapter 9

Metrics

In this chapter we outline our choice of metrics. The goal of these metrics is two-fold. First, a metric provides a quantitative way of comparing the results of the different rendering approaches. Second, analyzing the results using metrics may lead to insights that can be used to improve our techniques. We developed three metrics:

- Texture similarity metric: Measures how much the texture in the rendered image "looks like" the texture in the sample.
- Shading error metric: Calculates how close the texture at a given pixel in the rendered image is to the the desired texture from the input sample.
- Frame-to-frame coherency metric: Measures how stable the texture from frame-to-frame. There are two distinct choices here; either the texture is "pinned" to the model and moves with the model, or the texture is fixed to the projected 2D image.

We define each of these metrics below.

9.1 Texture similarity metric

There are many communities that are examining the question of how to measure visual similarity, such as human perception researchers, image database querying, image recognition, and texture synthesis. Developing a metric for general human perception is beyond the scope of this research. We instead focus on a metric that is capable of measuring the types of texture distortion we expect to be present. These distortions can be categorized as stretch, rotation or shearing effects, and discontinuities or poor

texture sampling. We first define the similarity metric and then demonstrate its behavior on a small test set that is designed to capture the above distortions types.

Our texture similarity metric combines several measures that are common building blocks in image database retrieval algorithms [27]. The first measure is the difference in the color histograms of the two images being compared. Next, we run 3×3 filters across each of the 3 color channels of each image to locate edges. The four filters, which locate horizontal, vertical, and diagonal edges respectively are given by the following matrices:

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 3 \\ 0 & 0 & 0 \\ -3 & 0 & 0 \end{bmatrix} \begin{bmatrix} -3 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

We create a total of fifteen histograms for each image (1 color histogram and 4 edge histograms for each of the 3 color channels of the image). Each histogram has 10 bins of equal size (the choice of 10 bins is arbitrary).

To compare a pixel from the source texture with a pixel from the rendered image, we first find the $s \times s$ block surrounding each pixel, then build the histograms using that block. We then measure the Euclidean distance between each pair of histograms, and normalize by dividing by $s \times s \times 15$. We have experimented with block sizes ranging from 4×4 to 12×12.

In order to perform meaningful comparisons, we must determine where a given pixel in the rendered image came from in the input sample. During rendering, the blocks around a given pixel may become distorted or the pixel itself may be the result of blending. It is therefore inappropriate to claim that we can be certain of where the resulting pixel came from. To improve the statistical soundness of our approach, we instead search for the k most likely matches and average between them (we typically use $k = 10$ in this research). To speed up this process, we preprocess the data and store it in a $k-d$ tree. This allows us to find the closest matching pixels in the input sample in $O(\log^3 n)$ time where n is the number of pixels in the input sample [1].

To check that this metric correctly captures texture distortion, we evaluated it on three test cases. For all tests we sampled the error at 100 randomly sampled pixels from the rendered image, with histogram image block sizes of 4, 8, and 12. These tests are explained below:

9.1.1 Test 1: Rotation

For the first test, we progressively rotate a copy of a square input texture and compared it with the original square. After each rotation, we sampled the error at 100 randomly chosen pixels. For relatively symmetric textures we would expect to see a small error for all angles. For textures with a strong diagonal element, the error achieves a maximum at $\pi/2$, but drops back as the angle approaches π. Figure 9.1 shows how our metric responds to rotation for three distinct textures. Notice that the results obtained by varying the block size are qualitatively similar, but quantitatively different. As the block size increases, the distributions generally smooth out, so the total error decreases. This turns out to be true for all the tests.

9.1.2 Test 2: Scale

For the second test, we progressively scale the texture in the x and y directions individually, and in both directions simultaneously. As in the previous test, we sample the error at at 100 randomly selected pixels. Figure 9.2 shows our metric measuring the error introduced by scaling. To simplify the graph, we only show the results for an 8×8 block size. The metric performs as expected since the error should increase as the image is shrunk and expanded.

9.1.3 Test 3: Poor texture sampling.

For the third test we introduce an increasing number of texture discontinuities. To create the discontinuity image we run a slightly modified version of the image based texture synthesis algorithm of Chapter 6. A block pasted into location (x, y) in the target image is taken randomly from a vertical stripe centered around x in the original texture image. The width of this stripe is $2n$, where $n \times n$ is the pasted block size. The minimum error cut between blocks is then applied. As the block size increases the error should decrease, since there are fewer boundaries where a cut is necessary. We generated 8 distortion images, with block sizes ranging from 4 to 32. Figure 9.3 shows three of these (block sizes 4, 20, and 32) for illustration purposes, along with a chart of the general (expected) response to the metric.

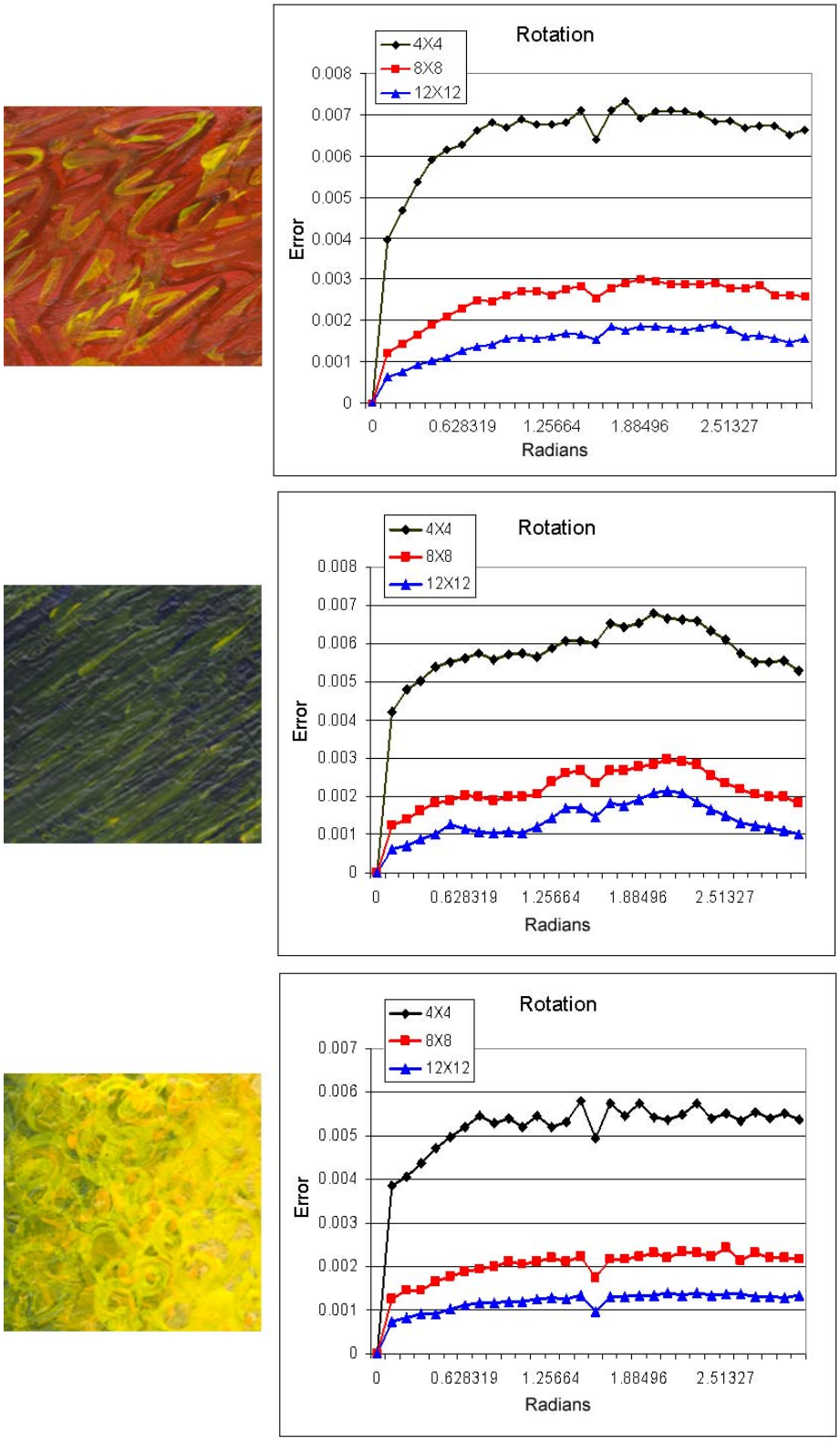


Figure 9.1: Rotation error for three different textures: random, diagonal, symmetric.

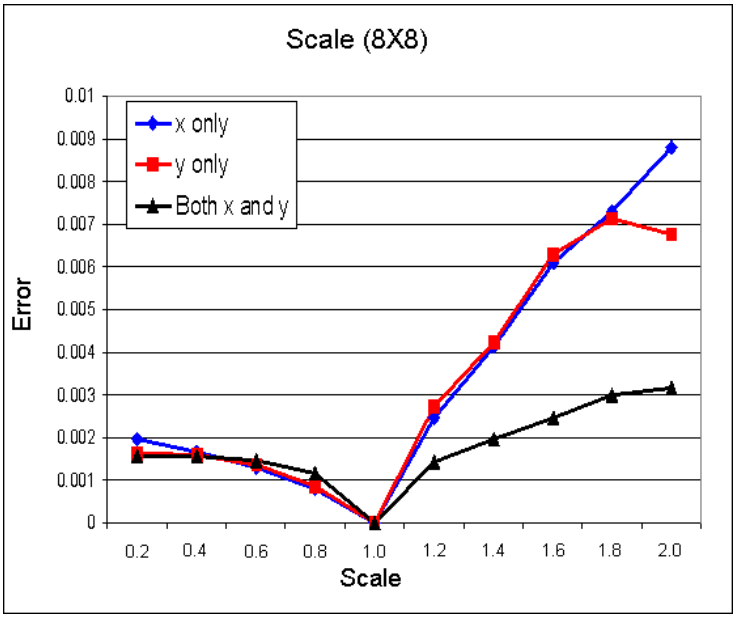


Figure 9.2: Scale error for the random texture.

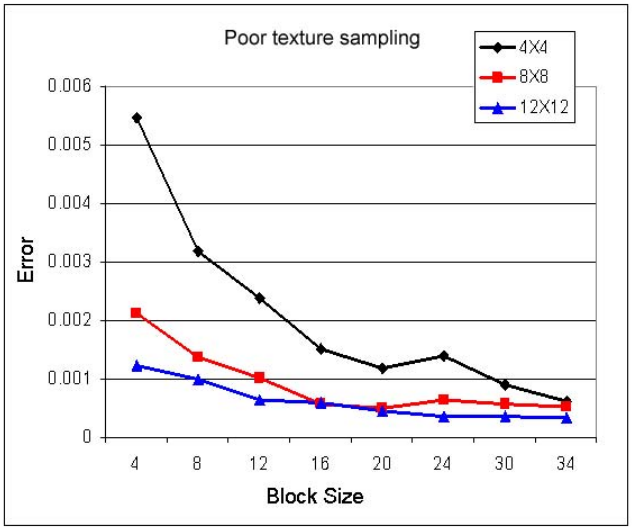


Figure 9.3: Error introduced by poor texture sampling. (block sizes 4, 20, and 32 also shown).

We have shown that the metric behaves as expected under different test conditions, and is also fairly robust to block size. The test set also provides us with an expected absolute measure of error for a given texture sample and block size.

9.2 Shading error metric

This metric calculates how close the texture at a pixel in the resulting image is to the desired texture for that shade value in the input sample. We first find the k pixels in the source texture that are the closest to the test pixel using the $k-d$ tree explained above. We then average the shade values corresponding to those k source pixels and compare it to the real shade value.

9.3 Frame-to-frame coherency metric

We measure frame-to-frame coherency in either image space or object space. For image-space coherency, we compare the histogram difference between the same pixel location in frame i and frame $i+1$. We measure this error for 100 randomly chosen pixels, making sure the sample blocks lie inside the rendered object for both frames. To measure object-space coherency we pick a point on the 3D object that is visible in both frames, and compare the pixels in the two frames.

Chapter 10

Results

In this chapter, we summarize the advantages and disadvantages of each of the rendering techniques in Section 10.1. We then evaluate our rendering techniques using our metrics in Section 10.2. We conclude this chapter by showing how a model skull is rendered using each of our techniques with a variety of paint paint samples.

10.1 Summary of the rendering techniques

10.1.1 Object-based texture mapping

Advantages:

- Runs in real time.
- Zero preprocessing time.
- Excellent frame-to-frame coherence.

Disadvantages:

- Texture quality varies as light position changes.
- Texture is completely lost on flat surfaces.

10.1.2 Image-based texture synthesis

Advantages:

- Resulting images are consistent with input paint sample (best quality of the four techniques).

Disadvantages:

- Slow rendering time (between 20sec and 1 minute per frame).
- Poor frame-to-frame coherence (suffers from the shower-door effect).

10.1.3 View aligned 3D texture projection

Advantages:

- Runs in real time.
- Quality of rendered images is close to that of the image-based texture synthesis.
- Fair degree of frame-to-frame coherence.

Disadvantages:

- Large preprocessing time (performing texture synthesis and ensuring that the textures are tileable may take as long as 15 minutes).

10.1.4 View dependent interpolation

Advantages:

- Runs in real time.
- Good frame-to-frame coherence.

Disadvantages:

- Loss of texture quality.

10.2 Evaluation of the rendering techniques

We use the metrics outlined in chapter 9 to compare our renderings (We omit the object-based texture mapping technique since the quality of its results are dependent on the position of the light as explained above). Table 10.1 shows how our well our

Table 10.1: Evaluation of the rendering techniques using the metrics from Chapter 9

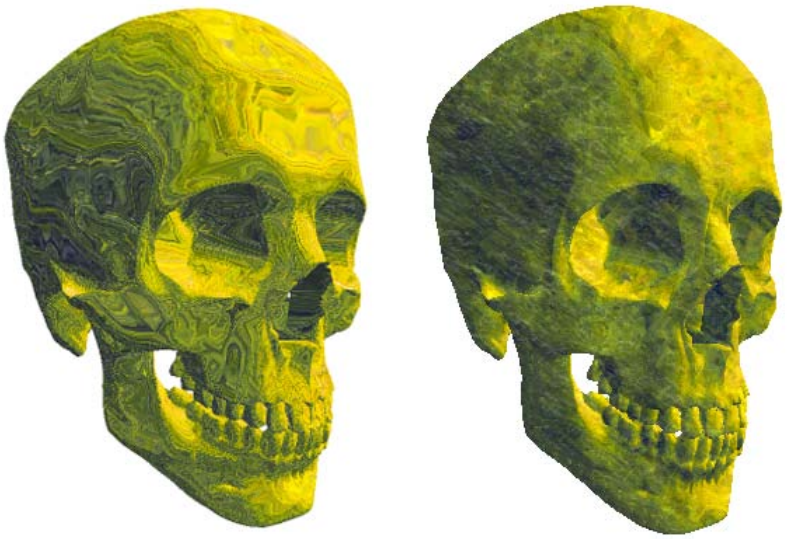
Technique	*Similarity*	*Shading*	*Frame-to-frame*
Image Based Texture Synthesis	0.07539	0.00536	0.00297
View Aligned 3D Texture Projection	0.08048	0.00521	0.00016
View Dependent Interpolation	0.10197	0.00582	0.00547

rendering techniques performed in terms of the texture similarity metric, the shading error metric, and the frame-to-frame coherency metric. The measurements were taken from the images in Figure 10.1. Other paint samples produced similar results.

Our metric shows that the image-based texture synthesis technique introduces the smallest amount of error in terms of texture similarity. This means that image based texture synthesis provides the greatest amount of texture fidelity. Also, the rendering methods capture shading with roughly the same amount of error. This is important since our goal is to correctly convey shading. We measured frame-to-frame coherence in image space. In this context, 3D texturing works best because the texture is only translated from one frame to the next, whereas texture synthesis must do blending to provide coherence. The view dependent method, while coherent in object space, is not at all coherent when measured in image space as the texture may be distorted by the curvature of the mesh.

10.3 Rendering Examples

Figure 10.1, Figure 10.2, and Figure 10.3 show the results of applying three different textures to a model skull, using each of our techniques.

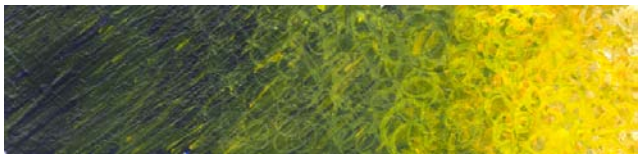


(a) Object-based

(b) Image-based

(c) View-aligned

(d) View-dependent



(e) Green to yellow paint sample

Figure 10.1: Results for green to yellow paint sample.

(a) Object-based

(b) Image-based

(c) View-aligned

(d) View-dependent

(e) Dark red paint sample

Figure 10.2: Results for dark red paint sample.

(a) Object-based

(b) Image-based

(c) View-aligned

(d) View-dependent

(e) Red to yellow paint sample

Figure 10.3: Results for red to yellow paint sample.

Chapter 11

Conclusion and Future Work

This thesis paper presented several techniques, each with distinct advantages and drawbacks for shading 3D computer generated models using scanned images of actual paint samples. A method for separating the color transition from the texture of a paint sample was also demonstrated. We also introduced metrics that evaluate how well each of our rendering techniques performed in terms of texture similarity, shading correctness, and temporal coherence.

We have divided our ideas for future work into two categories. The first category looks at improving the current rendering techniques. Despite the drawbacks of our object-based texture mapping technique, it has the shortest overall running time (zero preprocessing time and real-time rendering time). Our other two techniques which render in real-time (view-aligned 3D texture projection and view-dependent interpolation) require a large preprocessing time due to the use of texture synthesis. Finally, our image-based texture synthesis approach, which produces the best quality results, does not run in real-time. This encourages us to explore techniques that provide the efficiency of the object-based texture mapping approach with the quality of the image-based texture synthesis approach. Recent work in performing texture synthesis directly on the surface of a model [30] could be coupled with 3D texture blending in order to achieve this goal. We also wish to extend each of our rendering techniques to capture artistic silhouettes [23] in styles provided by the user. We chose to do this because the boundaries of the rendered images from all of our techniques are sharp, which is uncharacteristic of artistic works.

Our second category for future work entails developing techniques that automate the process of composing a scene. Traditional scene composition involves manually placing the models in a scene. Additionally, the user manually places and

determines the parameters for the camera and the lights. This is a time consuming process as it is necessary to make modifications to the scene, evaluate the results and repeat the process until the desired scene/lighting is obtained. We hope to automate this process.

One of the central problems in performing automatic scene composition is developing an intuitive method for communicating to the computer what the desired scene should "look like". Recent work by D. Lischinski has led to the development of an automatic lighting design tool [28] and a method for automatic camera placement [11]. We hope to use these results as a base for our future work.

References

[1] Sunil Arya and David M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 271–280. ACM Press, 1993.

[2] Michael Ashikhmin. Synthesizing natural textures. In *Symposium on Interactive 3D Graphics*, pages 217–226, 2001.

[3] James F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Comm. ACM*, 19(10):362–367, July 1976.

[4] J. S. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. In *Computer Graphics*, pages 361–368. ACM SIGGRAPH, 1997.

[5] Paul Bourke. Tiling textures on the plane (part 2) http://astronomy.swin.edu.au/ pbourke/texture/tiling2/.

[6] E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah, 1974.

[7] T. Corman, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[8] Oliver Deussen, Stefan Hiller, Cornelius van Overveld, and Thomas Strothotte. Floating points: A method for computing stipple drawings. *Computer Graphics Forum*, 19(3), 2000.

[9] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 341–346. ACM Press, 2001.

[10] Alexei A. Efros and Thomas K. Leung. Texture synthesis by non-parametric sampling. In *IEEE International Conference on Computer Vision*, pages 1033–1038, Corfu, Greece, September 1999.

[11] Shachar Fleishman, Daniel Cohen-Or, and Dani Lischinski. Automatic camera placement for image-based modeling. *Computer Graphics Forum*, 19(2):101–110, Jun 2000.

[12] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics — Principles and Practice.* The Systems Programming Series. Addison-Wesley, second edition in c edition, 1996.

[13] Bert Freudenberg, Maic Masuch, and Thomas Strothotte. Real-time halftoning: a primitive for non-photorealistic shading. In *Proceedings of the 13th workshop on Rendering*, pages 227–232. Eurographics Association, 2002.

[14] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 447–452. ACM Press, 1998.

[15] Paul S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, November 1986.

[16] David J. Heeger and James R. Bergen. Pyramid-based texture analysis/synthesis. In *SIGGRAPH*, pages 229–238, 1995.

[17] Aaron Hertzmann and Ken Perlin. Painterly rendering for video and interaction. In *Proceedings of the first international symposium on Non-photorealistic animation and rendering*, pages 7–12. ACM Press, 2000.

[18] Robert D. Kalnins, Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Joseph C. Lee, Philip L. Davidson, Matthew Webb, John F. Hughes, and Adam Finkelstein. Wysiwyg npr: drawing strokes directly on 3d models. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 755–762. ACM Press, 2002.

[19] Michael A. Kowalski, Lee Markosian, J. D. Northrup, Lubomir Bourdev, Ronen Barzel, Loring S. Holden, and John F. Hughes. Art-based rendering of fur,

grass, and trees. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 433–438. ACM Press/Addison-Wesley Publishing Co., 1999.

[20] Adam Lake, Carl Marshall, Mark Harris, and Marc Blackstein. Stylized rendering techniques for scalable real-time 3d animation. In *Proceedings of the first international symposium on Non-photorealistic animation and rendering*, pages 13–20. ACM Press, 2000.

[21] Aditi Majumder and M. Gopi. Hardware accelerated real time charcoal rendering. In *Proceedings of the second international symposium on Non-photorealistic animation and rendering*, pages 59–66. ACM Press, 2002.

[22] Barbara J. Meier. Painterly rendering for animation. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 477–484. ACM Press, 1996.

[23] J. D. Northrup and Lee Markosian. Artistic silhouettes: A hybrid approach. In *Proceedings of the First International Symposium on Non Photorealistic Animation and Rendering (NPAR) for Art and Entertainment*, June 2000. To be held in Annecy, France.

[24] Victor Ostromoukhov. Digital facial engraving. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 417–424. ACM Press/Addison-Wesley Publishing Co., 1999.

[25] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 465–470. ACM Press/Addison-Wesley Publishing Co., 2000.

[26] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, page 581. ACM Press, 2001.

[27] Y. Rui, T. Huang, and S. Chang. Image retrieval: current techniques, promising directions and open issues, 1999.

[28] Ram Shacked and Dani Lischinski. Automatic lighting design using a perceptual quality metric. 20(3):??–??, September 2001.

[29] Peter-Pike Sloan, William Martin, Amy Gooch, and Bruce Gooch. The lit sphere: A model for capturing npr shading from art. In *GI 2001*, pages 143–150, June 2001.

[30] Cyril Soler, Marie-Paule Cani, and Alexis Angelidis. Hierarchical pattern mapping. In *Siggraph'02*, July 2002. to appear.

[31] Cyril Soler, Marie-Paule Cani, and Alexis Angelidis. Hierarchical pattern mapping. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 673–680. ACM Press, 2002.

[32] Greg Turk. Texture synthesis on surfaces. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 347–354. ACM Press, 2001.

[33] Matthew Webb, Emil Praun, Adam Finkelstein, and Hugues Hoppe. Fine tone control in hardware hatching. In *Proceedings of the second international symposium on Non-photorealistic animation and rendering*, pages 53–ff. ACM Press, 2002.

[34] Li-Yi Wei and Marc Levoy. Texture synthesis over arbitrary manifold surfaces. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 355–360. ACM Press, 2001.

[35] Georges Winkenbach and David H. Salesin. Computer-generated pen-and-ink illustration. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 91–100. ACM Press, 1994.

Vita

Reynold J. Bailey

Date of Birth April 23, 1977

Place of Birth Kingstown, St. Vincent & the Grenadines

Degrees B.S. Computer Science & Mathematics,
Magna Cum Laude, May 2001

Publications Reynold Bailey, Cindy Grimm, Christopher Kulla, and James Tucek. "Using Texture Synthesis for Non-photorealistic Shading from Paint Samples", to appear in *Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, Alberta, Canada, October 2003.

Reynold Bailey, Ranette Halverson, Nelson Passos, Richard Simpson, and Delvin Defoe. "Theoretical Constraints on Multi-Dimensional Retiming Design Techniques", in *Proceedings of the 13th International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, August 2000.

Reynold Bailey, Ranette Halverson, Nelson Passos, Richard Simpson, and Delvin Defoe. "A Study of Software Pipelining for Multi-dimensional Problems", in *Proceedings of the AeroSense-Aerospace/Defense Sensing, Simulation and Controls*, Orlando, FL, April 2001.

May 2004