Department of Computer Science & Engineering

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

WUCSE-2003-54

Using Texture Synthesis for Non-Photorealistic Shading from Paint Samples

Authors: Kulla, Christopher; Tucek, James; Bailey, Reynold; Grimm, Cindy

May 20, 2003

WUCSE-2003-54: Using Texture Synthesis for Non-Photorealistic Shading from Paint Samples

Christopher D. Kulla cdk1@cec.wustl.edu
James D. Tucek jdt1@cec.wustl.edu
Reynold J. Bailey rjb1@cs.wustl.edu
Cindy M. Grimm cmg@cs.wustl.edu

Department of Computer Science & Engineering
Washington University in St. Louis
One Brookings Drive, Campus Box 1045
St. Louis, MO, 63130-4899

Abstract

This paper presents several methods for shading meshes from scanned paint samples that represent dark to light transitions. Our techniques emphasize artistic control of brush stroke texture and color. We first demonstrate how the texture of the paint sample can be separated from its color gradient. We demonstrate three methods, two real-time and one off-line for producing rendered, shaded images from the texture samples. All three techniques use texture synthesis to generate additional paint samples. Finally, we develop metrics for evaluating how well each method achieves our goal in terms of texture similarity, shading correctness and temporal coherence.

1. Introduction

Traditional artists convey shading using both brush stroke texture and by varying the color of the paint. They also work exclusively on a flat canvas, requiring a good sense of abstract spatial visualization to convey believable lighting. On the other hand, computer graphics programs are very good at computing lighting, but are not able to make artistic decisions about color choice and paint texture. The algorithms and techniques we present in this paper attempt to address this disparity by letting the artist and software each do what they are best at. When 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. Our goal is to make this freedom accessible when shading 3D models, while leveraging the computer's processing power to paint complex meshes.

Many existing non-photorealistic shading techniques are applications of very stylized techniques such as hatching or stippling. Particular tone values are created by adding in strokes until the over-all effect is dark enough. Creating a procedural texture for these types of techniques is fairly straightforward. In this paper we explore the use of less structured texturing styles, giving the artist more freedom over brush texture and color changes.

In our system, 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 physical media or created using a 2D paint program. We then apply this user-defined shading style to a mesh, making it appear to be painted with the same technique (see Figure 1). Consider a typical sample as shown in Figure 8(a). It has two distinct properties that vary with shading: color and brush texture. The color transitions are typically smooth (albeit non-linear), which makes them easy to model. Extracting texture change is more difficult, both because the texture transitions are coarser than the lighting changes, and because creating "more" texture is not as simple as it was in the hatching or stippling case.

To generate more texture we rely on texture synthesis, which is not a perfect process since even the best known algorithms have cases in which they fail. We present a texture synthesis algorithm, based on the image quilting approach [3], that is suited to the generation of texture for shading.

There are a variety of ways one might apply shaded texture to an object, and even more ways to animate the result. We present three different techniques that range from purely image-based to the more traditional texture-mapping approaches. In order to quantify and compare these methods, we introduce a metric that qualitatively captures common texture distortions, shading errors and temporal coherence in animation.

We discuss previous work in section 2. In section 3 we show how to process paint samples to separate color transition from texture transition. In section 4 we describe three

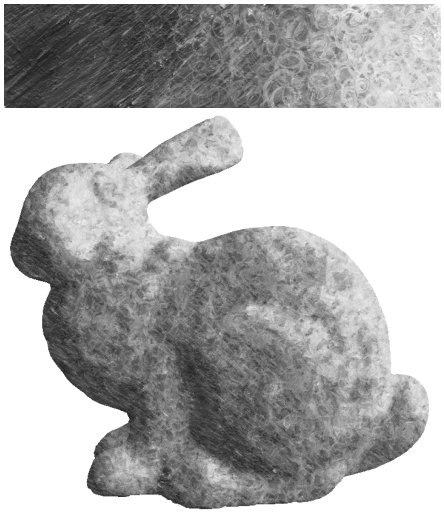


Figure 1. A typical shading paint sample, scanned by the user, applied to a mesh.

rendering methods that apply the paint sample to a lit mesh. Section 5 introduces our metrics and section 6 compares the quality of each rendering method using them. We discuss future directions of our work in section 7.

2. Previous Work

The problem of non-photorealistic shading is well studied. Technical illustration shading [5] introduced the use of warm-cool color blends to enhance the perception of shape and orientation. Artists indeed typically use colors that, while unrealistic, can enhance the perception of the image. The lit sphere approach [16] extracts 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 [9] which uses lighting values to apply a thresholded color map, giving a cartoon-like appearance.

Artists also make use of texture change across a surface to convey lighting. Several papers have addressed the technique of hatching [14, 21, 9] in which the lit appearance is conveyed by varying stroke densities and orientations. Other rendering styles that rely purely on texture change are charcoal rendering [10], engravings [12] or half-toning [4]. All of these systems use texture only and do not address the issue of color (which is reasonable for the particular mediums they convey). All of these techniques are very stylized and are therefore able to be captured procedurally or with minimal user input.

Stroke based techniques such as the WYSIWYG NPR system [7] 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 the model is also used in painterly rendering [11]. This approach is particularly interesting because it completely automates the painting process using a particle system to place the strokes and various surface shaders to determine their orientation, size, color and texture. The system also addresses frame-to-frame coherence by reusing strokes throughout the animation. Finally, Webb et al. [19] show how to convey both texture and color change with shading. They make use of the lapped texture method [13] to place several levels of texture onto a mesh, then blend between levels using 3D textures. This gives the illusion that the strokes are pinned to the mesh.

In this paper, we present an alternative approach 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. In particular, we were inspired by the texture transfer algorithm [3] which is able to render an image in the texture of a provided sample (for example a banana textured like an orange).

Several papers [18, 20] address the problem of synthesizing texture across a mesh, instead of across an image. These approaches require that the mesh be topologically "nice". We present two alternatives for texturing that operate on arbitrary meshes, and have different trade-offs for texture accuracy versus object frame-to-frame coherency.

3. Paint Sample Processing

A scanned paint sample has two distinct properties: texture and color. The color gradient is the global color change across the sample. We call this change the color trajectory, as it defines a path through color space. The color trajectory of a paint sample is typically non-linear and is stored as a 1D texture map. Brush texture can be described as local variations of the color gradient. We first show how a smooth color gradient can be extracted from a paint sample. Then, we show how the texture of the sample can be treated independently of the color trajectory, allowing the user to modify the perceived color transition while preserving the textured features of the sample.

3.1. Color Gradient Extraction

We assume that our paint sample is stored in a wide horizontal 2D image and that the shading changes from left to right. An example of such a paint sample is provided in Figure 8(a). As can be seen in Figure 9, plotting the color

values of the paint sample in RGB space reveals the rough shape of its non-linear color trajectory.

To extract a smooth 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 a 1D image strip that represents a path through color space. Unfortunately, the result contains a fair amount of streaking due to local texture variations (Figure 8(b)). We run the following recursive algorithm on the 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 diameter AB (see Figure 2). 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 of the input if necessary. Figure 8(c) shows how the output of the algorithm has effectively removed the streaking effect while maintaining the shape of the non-linear color blend we wanted to extract.

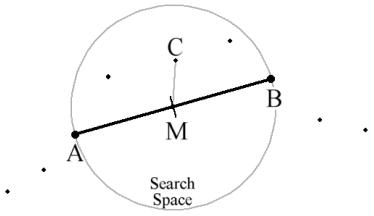


Figure 2. Searching for the sample point halfway between A and B

For trajectories of high curvature, this 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.

3.2. Separating Color and Texture

Texture change can be viewed as a local modulation of the color gradient. We can represent it separately by subtracting the color gradient from each pixel column of the original paint sample. This produces an intermediate difference image with RGB values ranging from -1 to 1 (Figure 8(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 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 separation reasonably well. Figure 10 shows a red-to-yellow sample being modified into 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 is also useful for rendering as described in the following section.

4. Rendering Methods

We present three different rendering methods to apply a paint sample onto a shaded mesh.

4.1. Image Based Texture Synthesis

This approach was inspired by the image quilting and texture transfer algorithm [3]. The image quilting algorithm synthesizes texture by cutting random blocks out of the sample and pasting them down in the target image in raster scan order. To ensure the continuity of the synthesized image, the pasted blocks overlap each other and are chosen in such a way as to minimize the color difference error of the overlapping areas. A second pass over the image processes these overlap regions and performs a "minimum error-cut" which produces an optimal irregular boundary between the blocks that further minimizes discontinuities. An additional contribution of image quilting is the notion of texture transfer, which makes it possible to render an image in the texture of a provided sample.

We start by rendering a grayscale shaded image 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. For the results presented in this paper, we used a directional light source $\vec{\mathbf{L}}$ and computed lighting as $(1+\vec{\mathbf{N}}\cdot\vec{\mathbf{L}})/2$ to get a full range of shading values across the mesh for analysis purposes. Nothing prevents the use of multiple lights or more sophisticated lighting algorithms. We supplement the shaded image with an object ID buffer [8] so that we know to which mesh each pixel belongs.

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 the block picking stage. In particular, we add 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). This constrains the search area to a region of the correct light level. This is somewhat different than what is suggested by Efros and Freeman for texture transfer. They add the constraint that the luminance of the block in the guide image must match the luminance of the block from the sample. Restricting the search range is more effective in our case because we do not want to restrict the luminance values of the paint sample to a linear ramp. For example, a paint sample that transitions from blue to red will have roughly constant luminance.

We made another important modification to the texture transfer 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 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: lighting changes are more important to capture with high frequency than texture changes. Getting the color gradient right is therefore more important than getting the texture to change along with the lighting. We generate the high frequency 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 by only using the texture difference image: the paint sample minus the color gradient as described in section 3.2. We recombine the two images by adding the RGB values together pixel by pixel, clipping any overflows that occur. 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 tweak the color gradient in real-time after the texture has been synthesized, increasing artistic control.

Naturally, the synthesis is only performed on the regions of the image that have the correct ID value in the ID buffer. Border cases where only a few pixels lying underneath a synthesis block are of the correct ID are handled by carefully counting those pixels and only adding in error terms for those pixels. Blocks that cover regions without any such valid pixels can be skipped, which speeds up image generation greatly.

Finally, we address the issue of creating animations with this method. Naively resynthesizing each frame from scratch produces a shower door effect [6]. 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 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. See the accompanying videos for examples of each method.

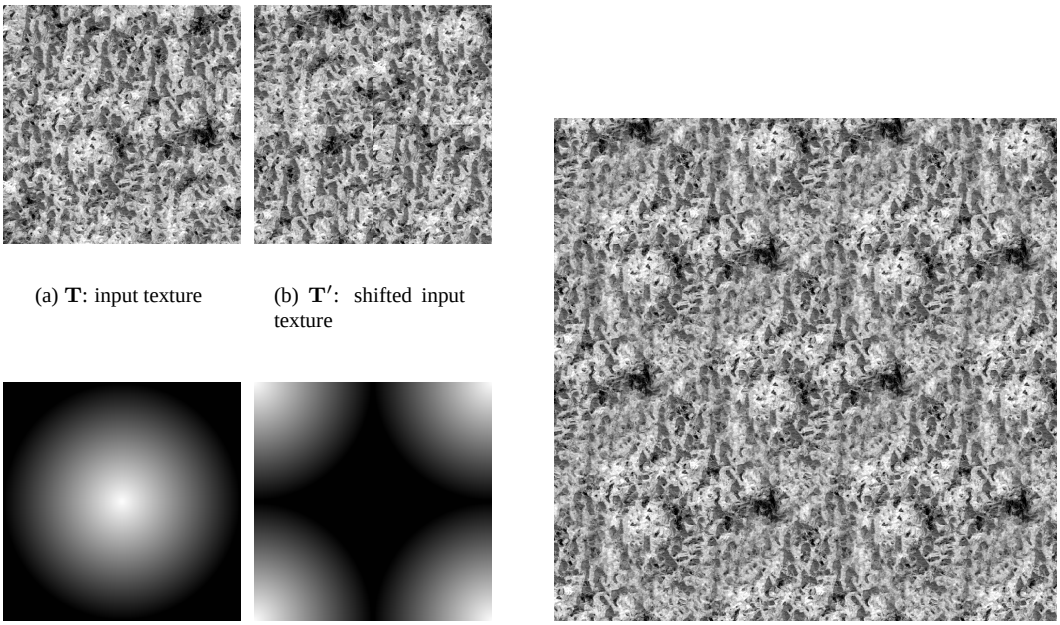
Stroke density in this method is directly related to the stroke density in the original sample. Therefore the ratio between image resolution and paint sample resolution is significant. Simply rescaling the input paint sample is sufficient to achieve a different stroke density. Since the image generation 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 to a minute depending on image resolution. The two following sections describe alternatives that run in real-time on commodity graphics hardware.

4.2. View Aligned 3D Texture Projection

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 of each section with image quilting. 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.

Our implementation runs on a GeForce 4 class graphics card with each level set to 512×512 pixels. In order to keep texture information separate from the color gradient, we subtract the average color of each section and only syn-



(a) $\mathbf{T}$: input texture

(b) $\mathbf{T}'$: shifted input texture

(c) $\mathbf{M}$: circular mask

(d) $\mathbf{M}'$: shifted mask

(e) $\mathbf{O}$: result tiled twice in each direction without any seams

Figure 3. Generating a tileable texture level.

thesize a texture difference image. We store this difference image in a regular bitmap by mapping the interval $[-1, 1]$ linearly to $[0, 1]$. Additionally, we guarantee that the texture be tileable. This constraint is not easily solved by adding constraints to the texture synthesis because of the raster scan order in which blocks are placed. Instead, we use a simple masking technique as a post process to achieve the desired result [2]. 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:

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

This method works very well in our case because the synthesized texture already has a repetitive structure, so the distortion introduced by blending is minimal. Figure 3 summarizes the process and demonstrates how the resulting texture can be tiled without any visible seams.

For rendering, we start by creating a 3D texture from each of the synthesized levels by stacking them in order of increasing shade level [19]. A simple pixel shader is used to access the 3D texture, expand the value back to the interval $[-1, 1]$ and add a color gradient indexed by the light value. We index the 3D texture using the screen coordinates of the pixel for s and t, and the lighting value for r, the depth texture coordinate. 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.

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 helps coherence for the most perceptually obvious cases.

4.3. View Dependent Interpolation

This technique is, in some sense, a combination of lapped textures [13] and texture synthesis on the mesh [18, 20]. Like lapped textures, we generate a small number of groups of mesh faces which are each covered with a texture. The groups of faces overlap, *i.e.* a face may be covered by more than one texture. Unlike lapped textures, we do not require that the face groups be disks, but only that the individual faces be within some epsilon distance of each other. This allows us to create a single texture map for the teeth of the skull, for example.

Like texture synthesis on a mesh, we can generate texture for one part of the mesh then propagate it to adjacent portions of the mesh and continue the texture synthesis there. However we do not require a connected mesh.

Once the texture maps are defined we use texture synthesis to create 3D textures, as was described in section 4.2.

To create the texture maps we chose a small number of views (typically 12-15) which surround the object. If the object has a preferred orientation we choose that direction as "up", if possible. We center the object in the view by automatically adjusting the zoom and panning the camera until the object is centered and as large as possible. We then use projection to generate texture map coordinates. We create an alpha mask using the dot product of the viewing direction and the face normals. To avoid "stretched" triangles we clip the dot product to a non zero epsilon.

The main issue to address with this method is self-occlusion. We scan-line render the mesh to determine which faces are visible, and to determine their depth-ordering. For every pixel that a face covers, we check if there is another face that covers that pixel and is closer in depth. If that face is covered, we remove it from the list of faces for that view. If the two faces are adjacent in the mesh then we do not mark the face as covered.

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.)

Because the views overlap, there is a subset of the faces that is visible in both views. The projection of the face creates a triangle in the image plane of each view; we can use this triangle correspondence to map pixels from one view to the other. During texture synthesis, as we finish the texture image for one view we copy the results into adjacent views to provide a smooth transition.

5. Metrics

In this section we outline our choice of metrics and provide empirical analysis of the performance on a test data set. The goal of these metrics is two-fold. First, a metric provides a quantitative way of comparing the results of different approaches. Second, defining an error metric can lead to insights about what it means to use a texture to shade.

Our error metric has three components:

- How much does the texture in the rendered image "look like" the texture in the sample?
- How accurately does the texture track the shade values? (Is the texture the correct one for that shade value?)
- How stable is the texture from frame-to-frame? There are two distinct choices here; either the texture is "pinned" to the object and moves with the object, or the texture is fixed to the image.

We define each of these below.

5.1. Texture similarity

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 paper. we 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 measure and then demonstrate its behavior on a small test set that is designed to capture the above distortions types.

The image similarity measures we use are common building blocks in image database retrieval algorithms [15]. The first measure is the difference in the color histograms. Next, we filter the image to locate edges in the horizontal, vertical, and diagonal directions. The second measure is the difference between the edge image histograms. Together, these two measures capture the distribution of color and edge directions within the image.

We create one histogram for each color channel of each image, for a total of fifteen histograms. Each histogram has 10 bins, with the divisions chosen so that the source texture pixels are evenly distributed in the bins.

The four edge images are created by running a 3×3 filter across the image. The four filters we use are:

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

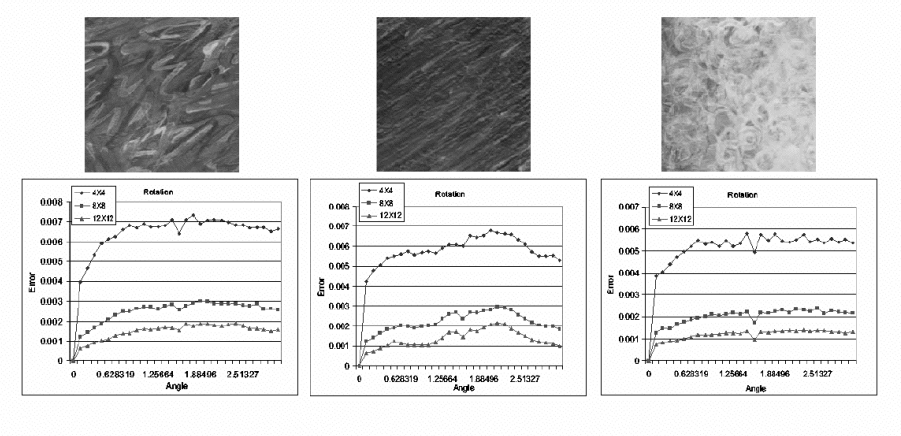


Figure 4. Rotation error for three different textures: random, diagonal and symmetric.

To compare two pixels we first find the $s \times s$ block surrounding the 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; the results are qualitatively similar, but quantitatively different. As the number of pixels increases, the distributions smooth out, so the total error decreases.

To compare a pixel to the source texture we find the best pixel match. To speed up this process, we pre-process the data and store it in a $k-d$ tree. This allows us to find the k nearest pixels in $O(\log^3 n)$ time [1].

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

The first test progressively rotates the texture and samples the error at 100 pixels for a 12×12 image block. 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 4 shows how our metric responds to rotation for three distinct textures.

The second test scales the texture in the x and y directions individually, and in both directions simultaneously. The error should increase as the image is shrunk and expanded. Figure 5 shows our metric measuring the distortion introduced by scaling in the x and y directions.

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 synthesis algorithm of Section 4.1. A block pasted into location (x, y) in the discontinuity 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

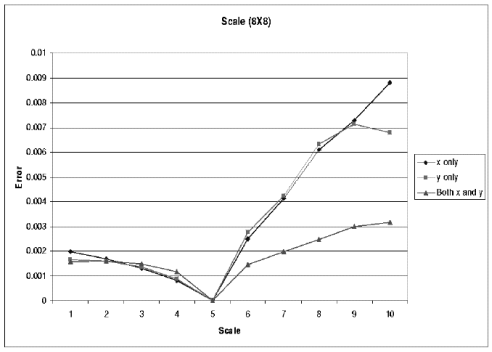


Figure 5. Scale error for the random texture.

size. The minimum edge cut between blocks is then applied. As the block size increases the distortion decreases, since there are fewer boundaries. We generated 8 distortion images, with block sizes from 4 to 32. Figure 6 shows these three synthesis results along with their response to the metric.

As shown in Figures 4, the metric behaves as expected on the test set, 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.

5.2. Shading error

This metric calculates how close the texture at a pixel is to the desired texture for that shade value. We first find the k pixels in the source texture that are the closest to the test pixel, using the metric outlined above. We then average the shade values corresponding to those k source pixels and compare it to the real shade value. By using k matched pixels (where k is typically 3) instead of a single pixel we get

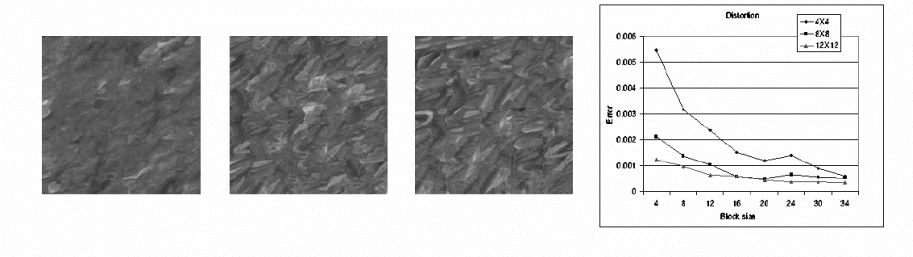


Figure 6. Sample distortions, block sizes 4, 20, and 32, for the random texture.

a better average shade measure, since texture can be fairly similar across a range of shade values.

5.3. Frame-to-frame coherency

We measure frame-to-frame coherency in image 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 some number of 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.

6. Results

Each of the rendering techniques presented in this paper has its distinct set of advantages and drawbacks. The image based texture synthesis method is the best for individual frame quality, but takes a long time to render. Additionally, frame-to-frame coherence is difficult to achieve, especially for samples that contain a lot of texture variation. The view aligned 3D texture method is very attractive because it can almost match the quality of the image based 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 mesh on screen. The view dependent technique also runs in real time, but keeps the texture attached to the object's surface. This gives excellent frame to frame coherence, but we lose texture quality because the texture is distorted to fit the contours of the mesh.

We use the metrics outlined in section 5 to compare our renderings. Our metric shows that the texture synthesis provides the greatest amount of texture fidelity (data is for green to yellow texture, other paint samples give similar results). All methods capture shading with the same amount of error, which is the most important result since our goal is to convey shading. Temporal coherence was measured 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.

	Similarity	Shading	Temporal
Texture Synth.	0.07539	0.00536	0.00297
3D Texture	0.08048	0.00521	0.00016
View dep.	0.10197	0.00582	0.00547

Figure 7. Evaluating the rendering methods of section 4 with error metrics from section 5.

7. Conclusion and Future Work

We have demonstrated three rendering algorithms for shading a mesh using a provided paint sample. Each technique has its particular advantages, in terms of texture fidelity and texture coherence. All methods accurately capture both color and texture change with lighting. Recent work in mesh texture synthesis [17] could be coupled with 3D texture blending. Artists also typically use brush strokes to convey more than just shading. We could imagine constraining our synthesis as to capture surface curvatures and silhouettes in styles provided by the user.

References

[1] S. Arya and D. 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] P. Bourke. Tiling textures on the plane (part 2) http://astronomy.swin.edu.au/ pbourke/texture/tiling2/.

[3] A. A. Efros and W. 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.

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

[5] A. Gooch, B. Gooch, P. Shirley, and E. 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.

[6] A. Hertzmann and K. 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.

[7] R. D. Kalnins, L. Markosian, B. J. Meier, M. A. Kowalski, J. C. Lee, P. L. Davidson, M. Webb, J. F. Hughes, and A. 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.

[8] M. A. Kowalski, L. Markosian, J. D. Northrup, L. Bourdev, R. Barzel, L. S. Holden, and J. 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.

[9] A. Lake, C. Marshall, M. Harris, and M. 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.

[10] A. 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.

[11] B. 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.

[12] V. 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.

[13] E. Praun, A. Finkelstein, and H. 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.

[14] E. Praun, H. Hoppe, M. Webb, and A. Finkelstein. Real-time hatching. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, page 581. ACM Press, 2001.

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

[16] P.-P. Sloan, W. Martin, A. Gooch, and B. Gooch. The lit sphere: A model for capturing npr shading from art. In *GI 2001*, pages 143–150, June 2001.

[17] C. Soler, M.-P. Cani, and A. Angelidis. Hierarchical pattern mapping. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 673–680. ACM Press, 2002.

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

[19] M. Webb, E. Praun, A. Finkelstein, and H. 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.

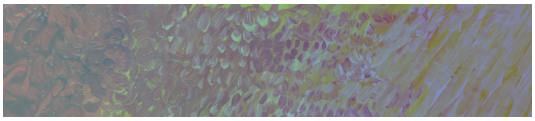
[20] L.-Y. Wei and M. Levoy. Texture synthesis over arbitrary manifold surfaces. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 355–360. ACM Press, 2001.

[21] G. Winkenbach and D. 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.

(a) Red to yellow paint sample

(b) Color gradient before sorting

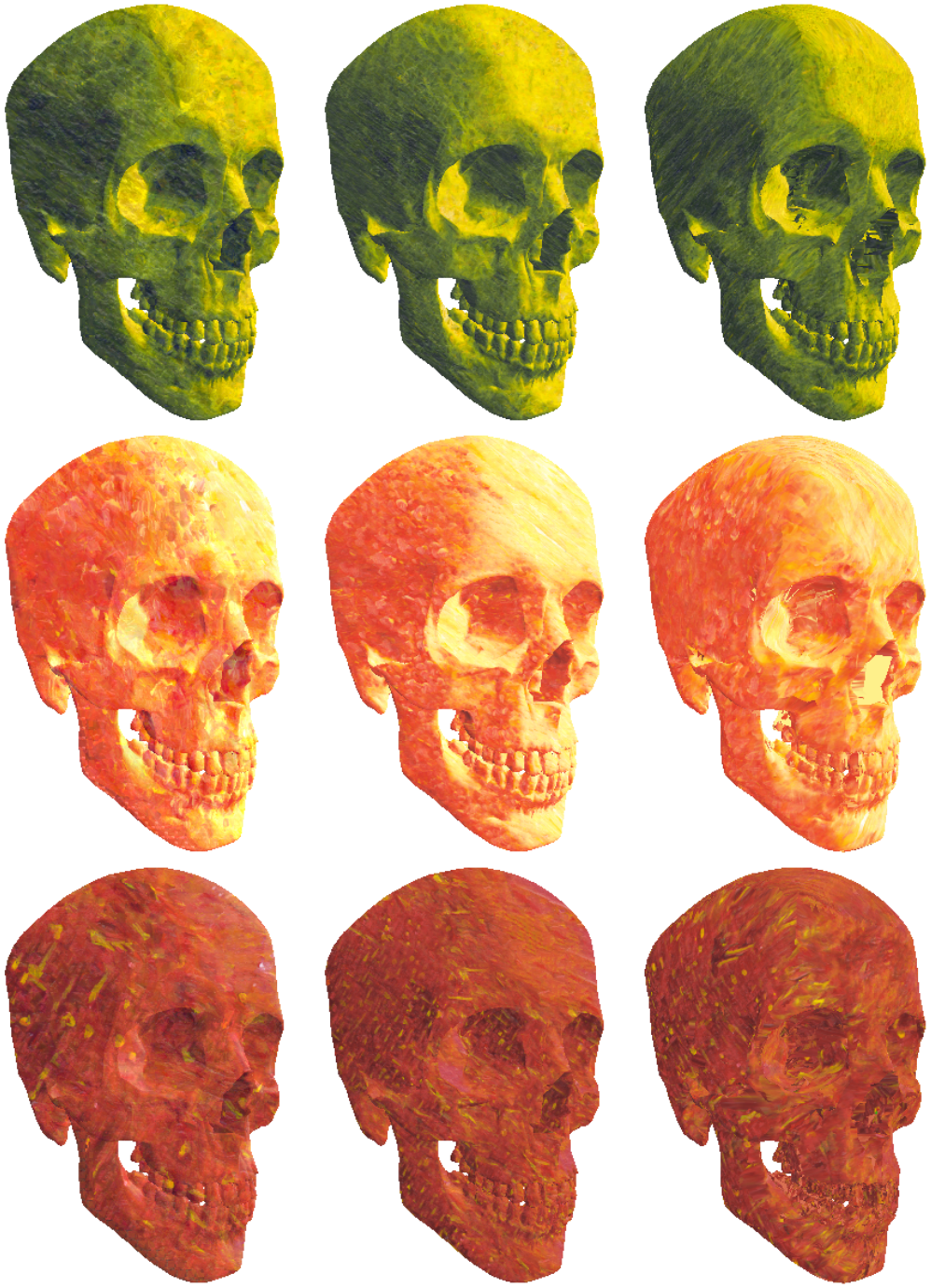
(c) Color gradient after sorting



(d) Extracted texture

Figure 8. Extracting color gradient and texture from a typical paint sample.

(a) Paint samples

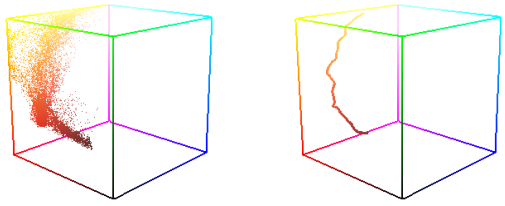


(b) Image Based Synthesis

(c) View aligned 3D Texture Projection

(d) View dependent interpolation

Figure 11. Rendering a skull mesh with various paint samples.



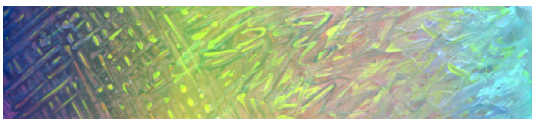
(a) Color distribution

(b) Color trajectory

Figure 9. Path through RGB space for the sample of Figure 8(a).

(a) A scanned paint sample

(b) User specified color gradient



(c) Applying the new gradient to the paint sample

Figure 10. Changing the color transition of Figure 8(a) without changing its texture.