Report Number: WUCSE-2003-53

2003-01-20

# Painting Lighting and Viewing Effects

Cindy Grimm

We present a system for painting how the appearance of an object changes under different lighting and viewing conditions. The user paints what the object should look like under different lighting conditions (dark, partially dark, fully lit, etc.) and (optionally) different viewing angles. The system renders the object under new lighting conditions and a new viewing angle by combining these paintings. We also provide a technique for constructing texture maps directly from the user's paintings.

Department of Computer Science & Engineering

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

2003-53

# Painting Lighting and Viewing Effects

Authors: Grimm, Cindy

January 20, 2003

Abstract: We present a system for painting how the appearance of an object changes under different lighting and viewing conditions. The user paints what the object should look like under different lighting conditions (dark, partially dark, fully lit, {\em etc.}) and (optionally) different viewing angles. The system renders the object under new lighting conditions and a new viewing angle by combining these paintings. We also provide a technique for constructing texture maps directly from the user's paintings.

# WUCSE-2003-53: Painting lighting and viewing effects

Cindy Grimm

***Abstract*—**
**We present a system for painting how the appearance of an object changes under different lighting and viewing conditions. The user paints what the object should look like under different lighting conditions (dark, partially dark, fully lit, *etc.*) and (optionally) different viewing angles. The system renders the object under new lighting conditions and a new viewing angle by combining these paintings. We also provide a technique for constructing texture maps directly from the user's paintings.**

## I. Introduction

In traditional 2D media an artist learns how to represent 3D forms on a 2D canvas using a combination of color, shading, and texture. Unlike photography, artists are free to render the world any way they like, whether it is physically "accurate" or not. They use the real world as a guide, but are not constrained by it.

In computer graphics, the artist controls the rendering process by changing lights, materials, textures, and shaders. This process lies somewhere between photography and painting; the artist has a great deal of indirect control over the way objects reflect light, but no *direct* control of the final image.

In this paper we describe a system that allows an artist to "paint" a 3D scene and what it should look under different lighting and viewing conditions. These paintings serve as an alternative method for specifying textures, shaders, and material properties. The goal is to let the artist use their traditional 2D skills in the 3D environment, an idea pioneered by 3D paint systems [8]. The original 3D painting systems were used to specify texture maps in an intuitive way; we extend this idea to the specification of shaders.

The artist begins by painting what the object should look like as if it were unlit, *i.e.*, completely in shadow. They next paint what the object should look like if it were fully lit. At this point, we have enough information to render the object, blending from the "dark" painting to the "light" painting as the shading on the object changes.

The artist is then free to add more paintings. These paintings may show what the object looks like when partially lit, what it should look like when viewed from a particular angle, under particular lighting conditions, or from far away. A whimsical example of a rendering using three paintings is shown in Figure 1.

The system is designed to be user-intensive, under the assumption that the user is a skilled artist and has a particular goal in mind. The effects that are created using the system could be duplicated using combinations of texture maps and shaders, and in fact, the rendering system is amenable to a hardware implementation. The advantage of this approach is, we believe, the directness of it.

We begin by putting this approach in context with existing work (Section II). We next discuss the system as seen from the user's point of view (Section III). The implementation section is broken into two pieces; the first describes how we use the artist's paintings as texture maps (Section IV-A). Next, we define the rendering process (Section IV-D). We close with results and conclusions.

C. Grimm is a professor at Washington University in St. Louis, USA (email: cmg@cs.wustl.edu)

## II. Previous work

This work continues the concept of using warm and cool colors [7] or painterly color models [18] to shade an object. We combine this with 3D painting [8], [19], [1] to let the user paint both the texture and the shade effects at the same time.

Several techniques exist for automatically shading models using common 2D techniques such as hatching [20], [17], [10], procedural pen-and-ink textures [21], and cartoon shading [11]. There are two primary challenges in stroke-based techniques. The first is to maintain constant shading tones and stroke thicknesses as the model is viewed from different distances. This is achieved by creating a set of "artistic mip-maps" [13]. Each layer of the mip-map contains all the strokes of the previous mip-map. The second problem is maintaining consistent strokes as the desired shading value changes; again, this is achieved by adding strokes to existing strokes, creating increasingly darker tones. Together, these stroke images form a 2D "spread sheet", where moving in one direction changes the perceived intensity, and the other direction adjusts for the number of pixels the model occupies. We adopt this "spread sheet" structure to store our paintings (see Figure 1).

In the non-photorealistic community there is a growing body of *stroke-based* rendering systems that are examining what it means to translate the concept of "brush stroke" to a 3D model. Early work let the user specify the model, the strokes, and how the strokes should be applied to the rendering [14]. Harold [3] was a system that directly captured the user's drawings and placed them in a 3D world. Recent work [12] has combined the automatic shading models with an interactive system for specifying the sample strokes and where they should go. We differ from this approach in that the user specifies the tone and the texture together.

Disney's Deep Canvas [4] was one of the first systems to convert an artist's 2D painting to 3D. Every stroke the artist made was "attached" to a 3D element in the scene. When the camera moved, the strokes were re-oriented and scaled to match the

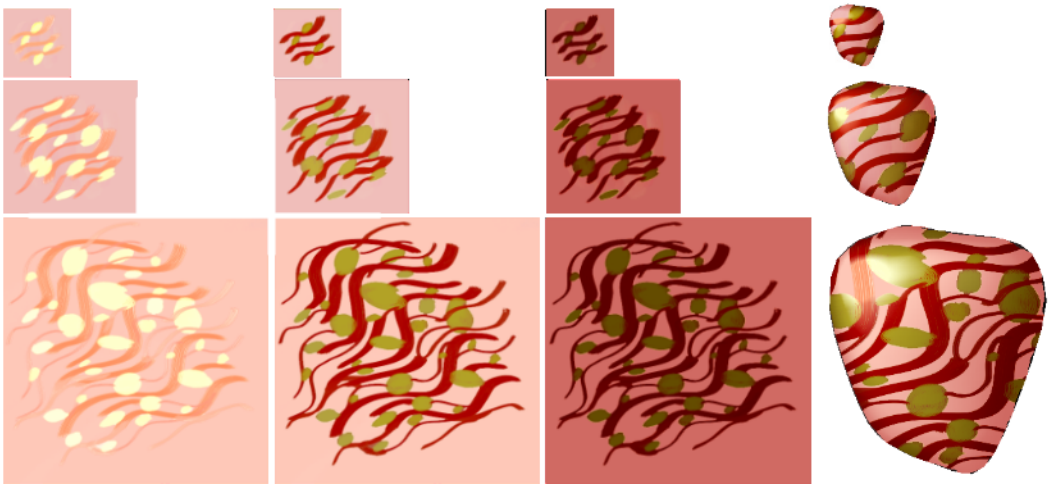

Fig. 1. An example using three shading values. The three paintings for the current camera view are shown in the upper left, a bump map in the upper right. Bottom row left: The intensity values on the object. Bottom row right: Using the three paintings to build a texture map by blending according to the intensity values. The dark painting is assigned the value 0, the lightest painting the value 1. From left to right we set the middle painting's value at 0.5, 0.9, and 0.95.

new viewpoint. When the viewpoint changed sufficiently, the artist would paint the scene from this new viewpoint. We adopt this notion of painting a series of viewpoints, but interpolate and blend in the texture map and not the strokes themselves.

3D painting requires a texture map, and a way to "reach" every point on the object with the paintbrush. A survey of the current approaches and problems can be found in a technical report by Low [15]. If a model has an existing texture map then we can use that. Takeo [9] introduced a method for creating a texture map "on the fly" by locally flattening out the mesh into the plane. This works well for simple non-occluding meshes, but becomes somewhat difficult for objects with handles. Lapped textures [16] provide a method for locally flattening out pieces of the mesh and texture mapping the pieces. One problem with using an existing texture map is that the user's paintings need to be resampled into the texture map; if the texture map resolution varies much from the sampled image this can create artifacts. For that reason, we introduce a texture mapping method that uses the paintings directly and can cope with self-occlusions.

View-dependent texture maps first arose in the context of image-based rendering [5]. In this case, photographs are aligned with the 3D model automatically. As the camera viewpoint changes, different sets of photographs are chosen and combined. We use the weighting scheme outlined in Buehler et. al. [2] to combine our paintings. This approach weights the blends based on how close rays are in angular distance and resolution (distance to the camera).

## III. User interface

In this section we describe the system from the user's point of view, leaving the details of the implementation for later sections.

When developing our system we chose to have the user use an external program, such as Painter$^{tm}$, to create the images (or, alternatively, they can scan hand-painted images in). This has the advantage that the user can use their favorite method for creating the 2D images, but it has the disadvantage of introducing an intermediate step between painting and viewing the results. We ameliorate this somewhat by copying data from existing paintings to new views before the user begins painting.

The system has two windows, a 3D one and a 2D one. In the 3D window the user can change the camera viewpoint and lights, see the results of one painting or a group of them, or what part of the object is currently un-painted. In the 2D window the user can page through the existing paintings, and add new shade values or mip-map levels.

Each painting consists of a set of images, the camera that was used to create those image's viewpoint, a set of shade values, a material shinyness, and an optional bump map. Each image in the set has a shade value and a mip-map level; the set of images forms an array indexed by shade and mip-map level (see Figure 1). To create a painting, the user first picks the camera viewpoint using the 3D window. In the 2D window they then name the painting and pick a shade value for the image. They can then optionally add new shade values and mip-map levels (which in turn creates more images).

We classify paintings into three classes; base-coat, view-dependent, and light-dependent. The base-coat paintings cover the visible part of the object and serve as the "base" texture. The view-dependent paintings only appear for a limited range of view angles (see Figure 3). The user has two sliders that control the view angle ranges; the first controls the total visible angular distance, the second controls how fast the painting fades out.

The light-dependent paintings are tied to the position of a particular light in the scene instead of the camera. The user has three sliders that control the angles over which the light-dependent painting appears. The first two are identical to the view-dependent sliders; the last one lets the user fade the painting out as the light moves away from the object.

A typical painting session begins with the user picking some number of base-coat views, typically 4-6. For each base-coat view the user specifies two shade values, one dark and one light, which creates corresponding dark and light images. These images initially contain a grey-scale rendering of the model. The user paints the images, then reads them back in and applies

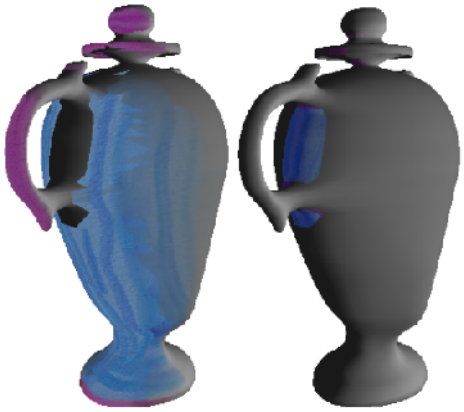

Fig. 2. Splitting the object into two paintings to avoid the self-occlusions. Left: The first layer contains the handle and the body of the vase, except for the part under the handle. Right: the part of the vase body that was covered by the handle. The uncovered portion of the mesh is shown in (smooth) grey.

Fig. 3. Left: The vase with just the base-coat. Middle: The angle at which the side view-dependent painting begins to appear. Right: The side view-dependent painting fully visible.

them to the model. The user then moves to the next painting viewpoint and writes out images that show the uncovered portion of the model as a grey scale image, and the covered portion showing the dark (or light) previous painting.

Once the initial base-coat is created the user has several options:

- Produce mip-map levels of the current paintings and edit them to create effects based on viewing distance and screen size.
- Add more shade levels to control the dark-to-light transitions.
- Add one or more view-dependent paintings (each of which contains one or more shade levels).
- Add one or more light-dependent paintings.
- Adjust the shinyness parameter. This is equivalent to the traditional shinyness parameter and controls how sharp the highlights are.
- Add a bump map. This is also equivalent to the traditional bump map and is used in the lighting calculation to adjust the surface normals of the texture map.

If the object is self-occluding then the user has the option of separating the object into pieces and painting each of the pieces with two or more paintings (see Figure 2). This is discussed in more detail in the texture section.

## IV. Implementation

In this section we provide implementation details for the texture maps (Section IV-A) and the rendering (Section IV-D). The texture map section describes how to use the painting images directly as texture maps. There are two issues here; first, how to cope with self-occluding models, and second, how to combine the paintings where they overlap on the object.

The rendering section defines how the paintings are shaded and then combined into a final rendering. We use image-based rendering techniques, based on the ones in Buehler et. al. [2], to combine the paintings. Although we use our texture mapping technique in this discussion, the methods themselves apply to any texture map representation.

### A. Texture maps from paintings

To create a texture map from a painting we project the vertices of the faces onto the image and use the projected locations as texture coordinates. Our algorithm addresses the two major problems with this approach, occlusion and shared faces.

For any reasonably complicated model there will be portions of the model that are occluded. This leads to two problems. First, if two faces map to the same pixel then they both get colored with that pixel's color. This is desirable for two neighboring faces but not so for two overlapping faces. Second, it may be difficult to find a view where the occluded faces are visible.

We approach the problem of occlusion by breaking the model's mesh into layers (see Figure 2). As a layer of the mesh is painted (with one or more paintings) we "peel off" that layer to expose the next set of faces to be painted. We also ensure that the occluded faces (even partially occluded ones) are not used in a painting. To make painting simpler, and to avoid texture blending artifacts, we enforce a pixel wide halo around faces that occlude other ones.

### B. Data structures

For each painting we store the layer, the list of faces associated with that painting, texture map coordinates for the vertices, the camera, and an alpha mask. The user provides the layer number, and the remaining data is calculated automatically.

### C. Algorithms

*1) Faces for a painting:* We start with the set of faces not assigned to a higher level. We run a modified two-pass scan-line algorithm to determine which faces are visible, which are occluded, and to calculate the point and normal for each pixel. In the first pass we perform the standard scan-line algorithm, keeping track of the points and normals, and which face they came from. Any face which falls across the edge of the image or is back-facing is eliminated at this stage.

In the second pass we increase the size of the polygon by half a pixel in all directions and keep track of all of the faces that map into each pixel, sorted by depth. For each pixel covered by more than two faces we look for possible occlusions. A face $f$ is occluded if there is a face $g$ that is closer and $g$ is not a neighbor of $f$ in the mesh.

To determine if $f$ and $g$ are neighbors we look for a path of adjacent faces $\{f_a\}$ that connect $f$ to $g$ such that every face in

$\{f_a\}$ maps to the current pixel. Usually $f$ and $g$ will either be adjacent or widely separated, but it is possible for several small faces to map to a single pixel.

If the mesh has intersecting polygons then the above algorithm will end up throwing both polygons out. As an alternative, we can sort the faces by their depth order (essentially the Painter's [6] algorithm) and perform occlusion testing on this ordered list. In this case, *any* face that overlaps and is not a neighbor is thrown out.

*2) Assigning faces to layers:* We need to assign each face to a primary texture; this is used at rendering time to ensure that there are no opacity gaps. We also use this algorithm to determine which layer a face should belong to. We begin with the set of faces that map to any layer zero painting. We assign the face to the painting where the face is most forward-facing. We then repeat with the set of faces covered by the second layer, but not the first. We continue until all of the faces are covered.

The above algorithms are interleaved with the user's creation of paintings. Usually the user will paint several views that cover the object, assign them to a single layer, then "strip off" those faces to begin painting the next layer. For example, layer zero for the vase was made first with six paintings that covered the top, bottom, and four sides. The scan-line algorithm left gaps in the areas behind the handles and around the lid. After painting these six views, the user ran the face assignment algorithm by clicking a button. They were then able to see just the faces left. They picked six more views, angled through the handle on each side and top and bottom, to fill in the back side of the handles, the vase body, and the remaining top and bottom of the lid.

*3) Combining paintings:* Faces will usually be covered by one or more paintings and we want to blend smoothly from one painting to the next. This is essentially an image-based rendering problem; we want to take the paintings that best cover a face and combine them based on the camera angle relative to that face. At this stage we currently blend only on camera angle, and not on resolution, because the paintings are usually at similar resolutions.

For each pixel in each face in each painting we store the percentage of that painting to use in the alpha mask. We then use OpenGL's blend routines to combine the results. To calculate the percentages we first find the angles, $\alpha_i$, between the face normal at that pixel and the ray from camera $i$ through that pixel. (We linearly interpolate the vertex normals across the face.) Let $\alpha_m$ be the maximum angle we wish to allow (slightly less than $90^{\text{deg}}$). We use a maximum angle rather than the largest angle because we may only have two paintings. The un-normalized weights are then:

$$w_i = \frac{1}{\alpha_i}(1 - \frac{\alpha_i}{\alpha_m})$$

We only use cameras where $\alpha_i < \alpha_m$. To normalize, we divide by the sum of all the weights. If $\alpha_i = 0$ for some camera, then we only use that camera.

### D. Rendering

In the rendering step we create a "shaded" texture map for each painting and combine the results together. We calculate the shaded texture map by finding the intensity value at each pixel and interpolating between the images that bracket that intensity value. All of the paintings are calculated in this manner; for the light-dependent paintings that depend on specularity we only use the specular component of the lighting calculation to determine the intensity.

The view-dependent and light-dependent paintings over-ride the base-coat paintings. We first calculate the percentage of each additional painting we wish to include; these numbers are derived from the user-specified maximum angle and fall-off. We then normalize the additional contributions, using the base-coat if the sum of the contributions is less than one.

*1) Shading the texture maps:* We calculate how much light the object should reflect, then use that number to linearly blend between the two bracketing shade values, creating a new "shaded" texture map. This calculation is performed on a per-pixel basis.

Suppose we have $N$ texture maps $t_i$ at shade values $0 \leq d_i \leq 1$, with $d_i < d_{i+1}$. For each pixel in the texture map we have stored a point $p$ and a normal $n$. The color of the pixel in the shaded texture map is found by first calculating the shade value $s$ at the pixel using the standard lighting calculation [6] ($l$ is the look vector, $I_a, I_d, I_s$ the ambient, diffuse, and specular light values, $d$ the distance to the light source, $l$ the vector to the light source):

$$s = I_a + \frac{1}{c_0 + c_1 d + c_2 d^2} \sum (I_d n \cdot l + I_s (r \cdot l)^e)$$

Next, we use that shade value to determine the two bracketing texture maps and how much of each to take:

$$i \quad \text{s.t.} \quad d_i \leq s \leq d_{i+1} \tag{1}$$

$$t_s(x,y) \quad = \quad \frac{d_{i+1} - s}{d_{i+1} - d_i} t_i + \frac{s - d_i}{d_{i+1} - d_i} t_{i+1} \tag{2}$$

We can either blend each of the color channels independently, or average the $RGB$ values in $s$ and use the same blend value for all channels.

We can easily incorporate bump mapping [6] and material shininess into this calculation. Since we are storing the per-pixel normals already, we can simply perturb these normals using the bump map and store the results. Material shininess is captured by the parameter $e$ in the lighting equation.

To create the point and normal information we use the standard scan-line [6] algorithm and store the results in two images. The maximum amount of data storage needed for the base coat (including the bump map and "scratch" texture space for the blended texture) is $(N + 4)(WH)$, where $W, H$ are the width and height of the texture map for the entire object.

*2) View-dependent maps:* View-dependent (VD) maps fade in and out based on the current viewing direction. Like the base coat, each VD map has one or more paintings at different shade values and point and normal information for those paintings. The final, shaded VD texture map is calculated by blending between the paintings on a per-pixel basis. Unlike the base coat, each VD map typically covers only a subset of the model (since there is no point in painting parts of the model that will not be

seen). To control the fade we weight the texture map's contribution by a value $w_v$ that depends on the current viewing direction (see Eq 3).

Each VD map has an associated viewing direction, represented by an `eye` point $p_e$ and an `at` point $p_a$. The `at` point lies along the look vector and in a plane containing the model. Given a new `eye` point $p'_e$ we can calculate $w_v$ as follows:

$$d = \frac{p_e - p_a}{||p_e - p_a||} \cdot \frac{p'_e - p_a}{||p'_e - p_a||} \quad (3)$$

$$w_v = \begin{cases} 0 & d \leq d_m \\ ((d - d_m)/(1 - d_m))^f & d > 0 \end{cases} \quad (4)$$

where $0 < d_m < 1$ is the cut-off angle specified by the user and $1 < f < \infty$ is the speed of the fall-off, also specified by the user. This is essentially a camera angle penalty [2]. This equation ignores the viewing distance (the appropriate mip-map level will be selected by OpenGL) and does not take into account where the object is in the field of view.

*3) Light-dependent maps:* Light-dependent (LD) maps are nearly identical to VD maps, except for the calculation of $w_v$. Instead of storing an `eye` point and an `at` point we store the source $p_s$ and direction $v_d$ of the associated light source. $d_m$ and $f$ control the rate of angular fall-off, as in the view-dependent maps. $m_o$ controls the distance at which the LD map falls off:

$$d_v = v_d \cdot v'_d \quad (5)$$

$$d_d = ||p_s - p'_s||/(m_o) \quad (6)$$

$$\alpha_v = \begin{cases} 0 & d_v \leq d_m \\ 0 & d_d \leq m_o \\ (1 - d_d)((d - d_m)/(1 - d_m))^f & otherwise \end{cases} \quad (7)$$

*4) Image-space size:* We use OpenGL's mip-mapping routines to account for changes in resolution. The user may override the default mip-maps, if desired (see Figure 1).

To reduce the computation time of the filtered images we can save and propagate down the shade values that were calculated at the top level.

*5) Blending additional paintings:* After calculating the weights for additional paintings, we sum the weights. If the sum is less than one then we make up the difference using the base-coat. If the sum of the weights is greater than one, then we normalize and do not use the base coat. The calculations are performed on a face-by-face basis.

## V. Results

In Figure 4 we see the same scene with different portions painted by two different artists. Most of the objects have between 6 and 8 paintings. The vase and the table both required slightly more paintings because of occlusion effects. The vase also has view-dependent effects, as can be seen in the accompanying video. The orange and table both have bump maps.

In Figure 5 we see two different plants, each with approximately 20,000 faces. The table, pot, and plant each have 6-8 paintings. For the plant we did not do any occlusion culling; all of the faces map to one of the paintings.

Rendering time for the scenes was between 1 and 5 seconds on a 2GHz Pentium processor.

## VI. Conclusions

We have presented a system for painting lighting and viewing effects that is a simple extension to existing texturing and lighting techniques. The approach is suitable for hardware acceleration. We also provide a method for building texture maps directly from user's paintings.

The system is currently being used by an artist with no computer science background. The artist is learning to use 3DS Max in addition to using in-house software. Unfortunately the artist has no experience with traditional 3D painting systems, so he cannot make any comparisons in that area. He does have this to say about the painting system versus the materials and shading system of 3DS Max:

> I am designing both the dark and light textures and the computer is putting them together for me. In 3DS Max I don't have that same direct control - I may be able to import a texture, but often end up spending hours tweaking lighting and material properties to find the dark and light images I'm looking for. This is a much simpler system to learn for someone coming from traditional media - 3DS Max is very powerful, and offers so many tools, but it doesn't let traditionally trained people take advantage of their learned skills.

We believe that "painting" provides a viable alternative to specifying lighting and viewing effects using traditional materials and shaders, especially for artists who are transitioning from traditional media to 3D computer graphics.

## References


[1] Agrawala Maneesh and. 3d painting on scanned surfaces. In *Symposium on Interactive 3D graphics*, 1995.

[2] Chris Buehler, Michael Bosse, Leonard McMillan, Steven J. Gortler, and Michael F. Cohen. Unstructured lumigraph rendering. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 425–432. ACM Press / ACM SIGGRAPH, August 2001. ISBN 1-58113-292-1.

[3] J. Cohen, J. Hughes, and R. Zeleznik. Harold: A world made of drawings, 2000.

[4] Eric Daniels. Deep canvas in disney's tarzan. In *ACM SIGGRAPH 99 Conference abstracts and applications*, page 200. ACM Press, 1999.

[5] Paul E. Debevec, Yizhou Yu, and George D. Borshukov. Efficient view-dependent image-based rendering with projective texture-mapping. In *Eurographics Rendering Workshop 1998*, pages 105–116, Vienna, Austria, June 1998. Springer Wein / Eurographics. ISBN 3-211-83213-0.

[6] James Foley, Andries van Dam, Steve Feiner, and John Hughes. *Computer Graphics: Principles and Practice*. Addison and Wesley, 1997.

[7] Bruce Gooch, Peter-Pike J. Sloan, Amy Gooch, Peter S. Shirley, and Rich Riesenfeld. Interactive technical illustration. *1999 ACM Symposium on Interactive 3D Graphics*, pages 31–38, April 1999. ISBN 1-58113-082-1.

[8] Pat Hanrahan and Paul Haeberli. Direct wysiwyg paiting and texturing on 3d shapes. In *Siggraph '90*, volume 24, pages 215—223, aug 1990.

[9] Takeo Igarashi and Dennis Cosgrove. Adaptive unwrapping for interactive texture painting. In *2001 ACM Symposium on Interactive 3D Graphics*, pages 209–216, March 2001. ISBN 1-58113-292-1.

[10] Pierre-Marc Jodoin, Emric Epstein, Martin Granger-Piché, and Victor Ostromoukhov. Hatching by example: a statistical approach. In *NPAR 2002: Second International Symposium on Non Photorealistic Rendering*, pages 29–36. ACM SIGGRAPH / Eurographics, June 2002. ISBN 1-58113-494-0.

Fig. 4. The entire still life. Each object was painted individually with between 8 and 12 paintings. Top row: Intensity values. Bottom row: Rendered images.

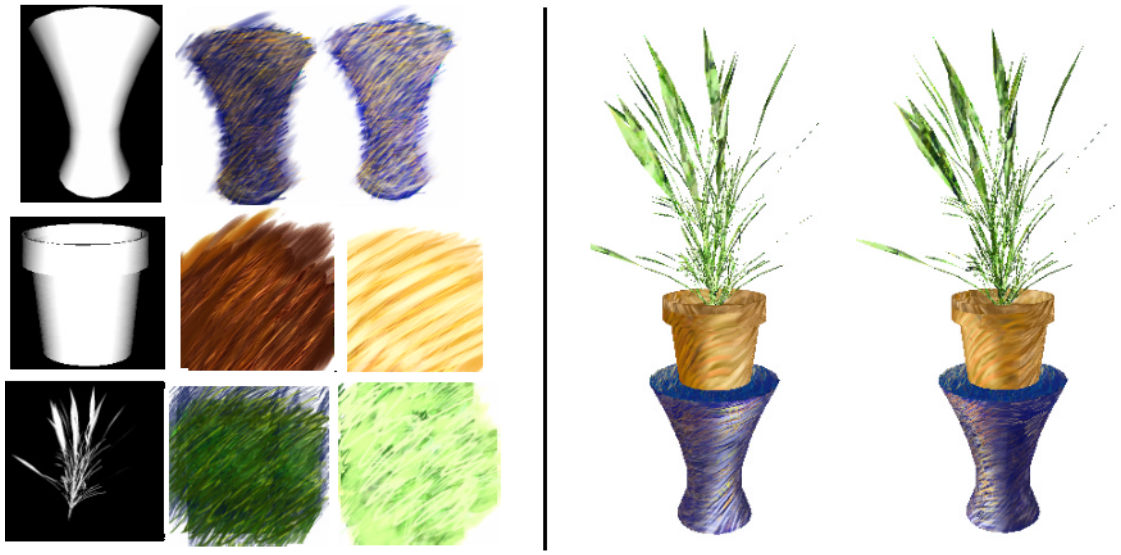

Fig. 5. Painting plants. Shown are example "dark" and "light" paintings for the table, pot, and plant. The images on the far left are the alpha masks for those paintings. On the right is two frames from an animation.


[11] Scott F. Johnston. Lumo: Illumination for cel animation. In *NPAR 2002: Second International Symposium on Non Photorealistic Rendering*, pages 45–52. ACM SIGGRAPH / Eurographics, June 2002. ISBN 1-58113-494-0.

[12] 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. *ACM Transactions on Graphics*, 21(3):755–762, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[13] Allison W. Klein, Wilmot W. Li, Michael M. Kazhdan, Wagner T. Correa, Adam Finkelstein, and Thomas A. Funkhouser. Non-photorealistic virtual environments. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 527–534. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, July 2000. ISBN 1-58113-208-5.

[14] Michael A. Kowalski, Lee Markosian, J. D. Northrup, Lubomir Bourdev, Ronen Barzel, Loring S. Holden, and John F. Hughes. Art-based rendering of fur, grass, and trees. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 433–438, Los Angeles, California, August 1999. ACM SIGGRAPH / Addison Wesley Longman. ISBN 0-20148-560-5.

[15] Kok-Lim Low. Simulated 3D painting. Technical Report TR01-022, 8 2001.

[16] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 465–470. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, July 2000. ISBN 1-58113-208-5.

[17] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. Real-time hatching. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 579–584. ACM Press / ACM SIGGRAPH, August 2001. ISBN 1-58113-292-1.

[18] Peter-Pike Sloan, William Martin, Amy Gooch, and Bruce Gooch. The lit sphere: A model for capturing NPR shading from art. In B. Watson and J. W. Buchanan, editors, *Proceedings of Graphics Interface 2001*, pages 143–150, 2001.

[19] Daniel Teece. 3d painting for non-photorealistic rendering. In *ACM SIGGRAPH 98 Conference abstracts and applications*, page 248. ACM Press, 1998.

[20] Matthew Webb, Emil Praun, Adam Finkelstein, and Hugues Hoppe. Fine tone control in hardware hatching. In *NPAR 2002: Second International*

*Symposium on Non Photorealistic Rendering*, pages 53–58. ACM SIGGRAPH / Eurographics, June 2002. ISBN 1-58113-494-0.

[21] Georges Winkenbach and David H. Salesin. Computer-generated pen-and-ink illustration. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 91–100, Orlando, Florida, July 1994. ACM SIGGRAPH / ACM Press. ISBN 0-89791-667-0.