Processing Images and Video for An Impressionist Effect

Peter Litwinowicz

Apple Computer, Inc.

ABSTRACT

This paper describes a technique that transforms ordinary video segments into animations that have a hand-painted look. Our method is the first to exploit temporal coherence in video clips to design an automatic filter with a hand-drawn animation quality, in this case, one that produces an Impressionist effect. Off-the-shelf image processing and rendering techniques are employed, modified and combined in a novel way. This paper proceeds through the process step by step, providing helpful hints for tuning the off-the-shelf parts as well as describing the new techniques and bookkeeping used to glue the parts together.

1. INTRODUCTION

In the 1800's, Claude Monet created paintings that attempted to "catch the fleeting impression of sunlight on objects. And it was this out-of-doors world he wanted to capture in paint -- as it actually was at the moment of seeing it, not worked up in the studio from sketches." [Kingston80].

Impressionist paintings provide the inspiration for the work presented here. We have produced images that are *impressions* of an input image sequence, that give a sense of an original image without reproducing it. These images have a “painterly” feel; that is, they appear as if they have been hand-painted. Furthermore, we have produced entire animations with these same qualities.

Producing painterly animations from video clips *automatically* was the goal of this work. Our technique requires that the user specify a few parameters at the start of the process. After the first frame is produced to the user's liking, our technique processes a whole video segment without further user intervention. Previous painterly techniques require much user interaction and have only been presented in the context of modifying a single frame (with the exception of a technique applied to 3D animated scenes).

While this technique is not the first to produce images with an Impressionist look, our method has several advantages. Most significantly, this paper presents a process that uses optical flow fields to push brush strokes from frame to frame in the direction of pixel movements. This is the first time pixel motion has been tracked to produce a temporally coherent painterly style animation from an input video sequence. Brush strokes are distributed over an input image and then drawn with antialiased lines or with supplied textures to produce an image in the Impressionist style. Randomness is used to perturb the brush stroke's length, color and orientation to enhance the hand-touched look. Strokes are clipped to edges detected in the original image, thus preserving object silhouettes and fine detail. A new technique is described to orient strokes using gradient-based techniques. In the course of being moved from frame to frame, brush strokes may become too sparse to cover the image. Conversely, brush strokes may become overly dense. The necessary algorithms for adding and deleting brush strokes as they are pushed too close or too far apart by the optical flow field are described within this paper.

The following section describes previously presented painterly techniques. Then we present the details of our technique:

1) the stroke rendering and clipping technique,
2) the algorithm for producing brush stroke orientations,
3) the algorithm for moving, adding and deleting brush strokes from frame to frame.

In conclusion, we discuss limitations of the algorithm and possible future directions.

2. BACKGROUND

Techniques for computer-assisted transformations of pictures are presented in [Haeberli90]. Many of those techniques involve extensive human interaction to produce the final images. More specifically, the user determines the number of strokes as well as their positions. The user controls the orientation, size and color of the strokes using combinations of interactive and non-interactive input. Examples of interactive input include cursor location, pressure and velocity; and non-interactive input include the gradient of the original image or other secondary images. Brush strokes can be selected from a palette and include both 2D and 3D brushes.

Painting each image in a sequence is labor intensive, and even more work is necessary to produce a sequence that is temporally coherent. "Obvious" modifications can be made to Haeberli’s technique, but each has their drawbacks. For example, imagine keeping the same strokes from frame to frame and modifying the color and direction of the strokes as the underlying image sequence dictates. Doing so produces a final animation that looks as if it has been shot through a pane of glass because brush strokes don't follow the movement of the objects in the scene. Conversely, generating the random strokes from scratch for each frame often produces animation with too much jitter. Modifying Haeberli's approach to produce temporally coherent animations is a primary focus for the work presented here.

Peter Litwinowicz
1 Infinite Loop, MS 301-3J
Cupertino, CA 95014
email: litwinow@apple.com

An interactive paint-like system for producing pen-and-ink illustrations is described in [Salisbury94]. The user specifies regions that are filled with chosen pen-and-ink patterns. Regions can be determined by hand, or specified as portions of a supplied secondary image. Similarly, tone can be supplied by hand or from some portion of an underlying reference image. Random variations are added to help produce a hand-drawn look. Building upon this work, [Salisbury96] presents a computer-assisted technique for producing scale-dependent pen-and-ink reproductions of images. In this work, tone from an image is used in conjunction with edges (detected from the same image) to produce a pen-and-ink format that is resolution-independent. The final rendered pen-and-ink images seek to preserve discontinuous shading across the edges that appeared in the original image, and to produce continuous shading in other areas. As with the techniques presented in [Haeberli90], applying the pen-and-ink techniques to sequences of images to produce a temporally coherent result is not straightforward. Motivated by this work, our technique also preserves perceived edges when transforming an input image.

The aforementioned techniques were only applied to single images. A system for producing 2-1/2D animations using "skeletal strokes" was presented in [Hsu94]. "Skeletal strokes" is a term used by the authors to describe a brush and stroke metaphor that uses arbitrary pictures as ink. However, all animation is key-framed by the user; that is, there is no automatic processing of an underlying image sequence.

In [Meier96], a system for transforming 3D geometry into animations with a painterly look is presented. 3D objects are animated and "particles" on the 3D surfaces are tracked. After the objects are projected into 2D (via a camera transformation), the particles are sorted by depth from the eye and then serve as positions for 2D brush strokes (painted back to front). Orientations are determined by using the surface normals as projected into the image plane. Brush size and texture is specified by the user. If desired, brush size may vary across a particular 3D object. This work demonstrates that temporal coherence of brush strokes is both interesting and important, but did not use video sequences as its input.

3. THE PROCESS

In this section we present our algorithm for overcoming some of the shortcomings of the previously described techniques. First, we describe the rendering technique, then the orientation algorithm, and finally the technique used to move brush strokes from frame to frame to produce temporally coherent animations. Color Plate 1 shows an example image that is used in demonstrating the process. (All color plates are located near the end of the paper so the reader may compare successive stages of the algorithm).

In order to create a final image there are many facets to our technique that work in concert with each other. However, explaining all the details at once would be confusing. We will first describe a very simple method to generate an image. As the paper progresses we will continue to describe modifications until the entire process has been explained.

A. Rendering strokes

Stroke Generation

To create the image shown in Color Plate 2, brush strokes are generated which, when rendered, cover the output image. Assume that each brush stroke is rendered with an antialiased line centered at (**cx**,**cy**), with a given length **length**, a given brush thickness **radius**, and a given orientation, **theta**. Assume that the brush strokes are generated with centers (**cx**,**cy**) positioned every two pixels in both the X and Y directions for the image. This spacing will assure coverage of the entire image with rendered brush strokes (brush radii and lengths shown in Table 1). In practice, the user sets the initial spacing distance. Then **cx** and **cy** are stored as floating point numbers for subpixel positioning. An orientation, **theta**, for each stroke is also needed. Discussion of the orientation calculation is deferred, so assume a constant direction of 45° (an arbitrary orientation chosen for demonstration purposes). The color for a particular stroke is assigned the bilinearly interpolated color of the original image at (**cx**,**cy**). Color components (r,g,b) are in the range [0,255]. Last, the order that the strokes are drawn is randomized to help create a hand-touched look (it helps break up the spatial coherence that would otherwise occur).

	Color Plates 2-5	Color Plates 6-8
Brush stroke radius (or offset for textured brushes)	1.5-2.0	4.0-4.5
Length	4-10	8-20

Table 1. Ranges for brush stroke radius and length.

Random Perturbations

Adding random variations and perturbations to a stroke helps to create a hand-crafted look. Much of the previous work on painterly renderings contain some form of random variation. We assign random amounts to **length** and **radius** in ranges supplied by the user (see Table 1). We perturb the color by a random amount for Δ**r**, Δ**g** and Δ**b**, each in the range [-15,15] (a range empirically found to be useful). We scale the perturbed color by a random amount, Δ**intensity**, in the range [.85,1.15]. After the originally sampled color is modified, the

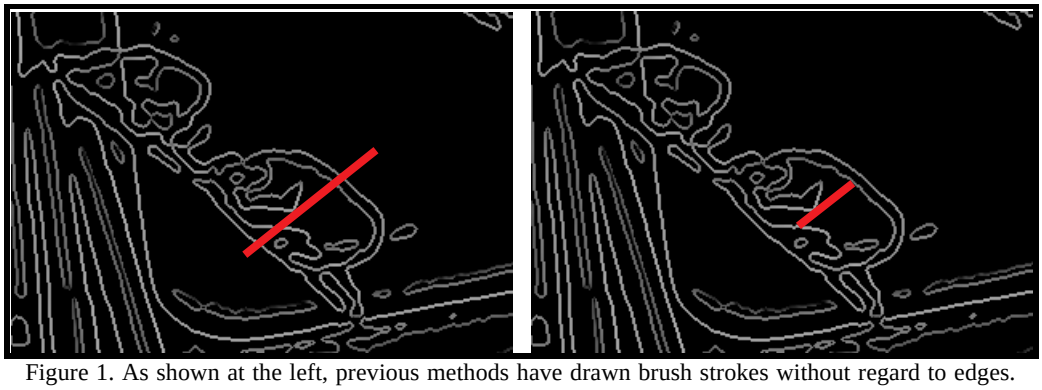


Figure 1. As shown at the left, previous methods have drawn brush strokes without regard to edges. The same brush stroke clipped to edges in the image is shown on the right.

resulting color is clamped to the valid range [0,255]. We also perturb **theta**, the orientation for the stroke, and do so by adding Δ**theta**, a random amount in the range [-15°,15°] (a range used for all images shown in the paper). **length**, **radius**, Δ**r**, Δ**g**, Δ**b**, Δ**intensity** and Δ**theta** are stored in the brush stroke's data structure and are used from frame to frame. We do not generate new values for each frame as this results in animations with too much jitter.

Clipping and Rendering

To render a brush stroke, an antialiased line is drawn through its center in the appropriate orientation. In order to preserve detail and silhouettes, strokes are clipped to edges that they encounter in the original image (see Figure 1). In this way, edges in the original image are more or less preserved. This is accomplished by starting at the center of the stroke and growing the stroke (in its orientation) until edges in the original image are detected. Color Plate 2 shows the results of drawing strokes while *not* trying to preserve edges, in contrast to Color Plate 3, which demonstrates the results of clipping the strokes against edges in the original image.

The stoke clipping technique is motivated in part by the work presented in [Salisbury94], in which stroke textures are clipped to edges provided by the user. Edges may be drawn or derived from an underlying image. [Salisbury94] presents an interactive system; the regions for stroke textures are specified by the user (using the original and secondary images, such as an edge enhanced version of the image). In our system there is no user interaction to specify edges; we rely solely on standard image processing techniques to locate edges.

The line clipping and drawing process proceeds as follows:

1. An intensity image is derived from the original color image. If the color value at a pixel is stored as red, green and blue components (r,g,b) in the range [0,255], then the intensity at each pixel is calculated as (30*r + 59*g + 11*b)/100 (standard conversion of r,g,b values to intensity value [Foley84]).

2. The intensity image is blurred with a Gaussian kernel [Jain95]. This blur helps reduce noise in the original video images. A larger kernel reduces noise, but at the expense of losing fine detail. A smaller kernel helps preserve fine detail, but may retain unwanted noise in the image. In this implementation, a B-spline approximation to the Gaussian is used. The width of the kernel should really depend on the content of the original sequence, so we let the user choose the kernel width. (A blurred image is shown in Figure 2, and uses a kernel that goes to zero at a radius of 11 pixels).

3. The resulting blurred image is Sobel filtered [Jain95]. The gradient (**Gx**,**Gy**) is calculated at each pixel and the value of the Sobel filter at any given pixel is:

Sobel(x,y) = Magnitude (**Gx**, **Gy**)

See Figure 2 for the Sobel filtered image of the example image.

4. Given the center (**cx**,**cy**), the orientation of the stroke, theta and the Sobel filtered image, endpoints of the stroke (**x1**,**y1**) and (**x2**,**y2**) need to be determined. The process starts at (**cx**,**cy**) and "grows" the line in its orientation until the maximum length is reached or an edge is detected in the smoothed image. An edge is considered found if the magnitude of the gradient (the Sobel value) decreases in the direction the stroke is being grown. See Appendix A for the pseudo-code for stroke clipping. This is similar to the edge detection process used in the Canny operator. For details of the Canny operator the reader is referred to [Jain95].

Determination of the stroke orientation, **theta**, is described in the following section. For Color Plate 3 we used a constant 45° orientation, with perturbations added.

5. The stroke is rendered with endpoints (**x1,y1**) and (**x2**,**y2**). The color of the stroke is assigned the color of the original image at the center of the stroke. The stored perturbations are used to modify this color, and then the color is clamped. The strokes in Color Plates 2 through 5 are rendered as antialiased lines using a stroke radius in the range [1.5,2.0] (again, a number used for example purposes). A linear falloff is used in a 1.0 pixel radius region (alpha, for compositing, transitions from a value of 1.0 to 0.0 linearly). Figure 3 shows how the values are used to render the stroke.

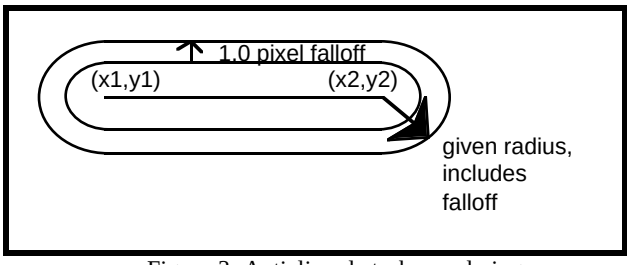


Figure 3. Antialiased stroke rendering.

Note that the drawing process touches pixels past the clipped endpoints of the line. Drawing slightly past the endpoints, along with the random ordering of strokes, creates a non-perfect line along edges. This helps to produce the hand-

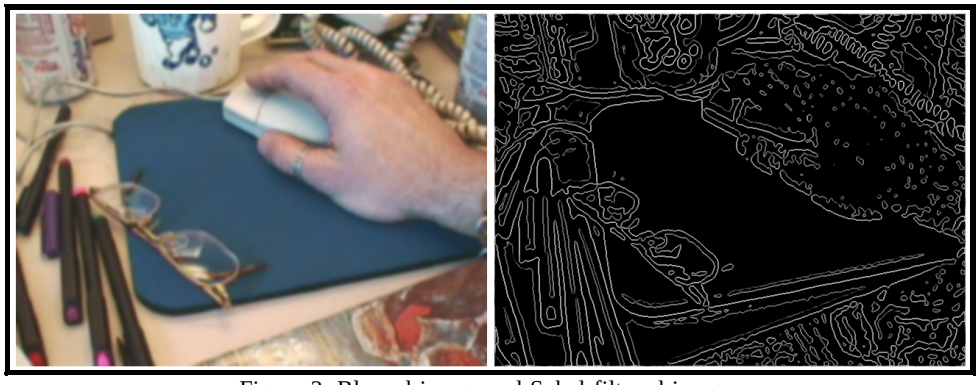


Figure 2. Blurred image and Sobel filtered image.

touched slightly wandering edges in the final image (shown in Color Plate 3) rather than an absolute hard edge.
It is important to note that, at the very least, a circle is drawn of the given brush radius, even if there are edges in the image that clip a particular stroke to zero length. This means that something will be drawn for each stroke even if the stroke's center is surrounded by edges in the original image.

Using Brush Textures

Brush strokes may also be rendered with textured brush images that have r,g,b and alpha components. A given offset is provided (akin to the radius given for the antialiased lines) and a rectangle surrounding the clipped line is constructed (see Figure 4). The brush stroke texture is then rendered into this rectangle. Each component of the color of the brush stroke texture is multiplied by the color assigned to the stroke. In the current implementation, the given offset is used regardless of the length of the clipped line. Another approach would be to scale the offset based on the length of the stroke. Color Plates 7 and 8 demonstrate the use of brush stroke textures.

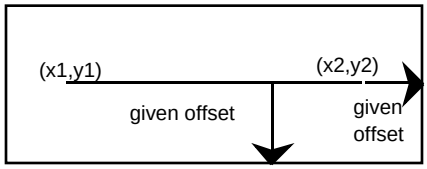


Figure 4. Rectangle for rendering textured brushes

B. Brush Stroke Orientation

In the previous section, a 45° constant orientation was used for all strokes (before the random variations were added). An artist, however, may not want to have all the strokes drawn in the same direction. We provide the user the option of drawing brush strokes in the direction of constant color, or near constant color, of the original image. This has the real world metaphor of strokes painted in a medium where a stroke does not change color as it is painted. This orientation can be approximated automatically by drawing strokes normal to the gradient direction (of the intensity image). Differentially, the gradient is the direction of most change, and normal to the gradient is the direction of zero change. Using this information, we assume that the image can be approximated locally with a relatively short stoke of constant color in the gradient-normal direction.

In our first implementation, the same Gaussian kernel used in the edge finding process was used for smoothing the image for the gradient calculation. However, using the same kernel did not produce a gradient that was smooth enough. A greater kernel width is used for the orientation calculation; in fact, the Gaussian filter used has a radius that is 4 pixels greater than the filter used for the edge finding process. Of course the user could supply this parameter, but for the video sequences we processed, the slightly larger kernel provided an adequately smoothed orientation field (and eliminated one more choice for the user).

The gradient direction is used to guide brush strokes in the work presented in [Haeberli90]. However, the user must interactively supply the position and length of the strokes. Using equally spaced brush strokes and the normal to the gradient of a smoothed version of the original image, Color Plate 4 was produced. However, when the magnitude of gradient is near zero we cannot rely on the gradient direction to be useful. We introduce a novel technique which modifies the gradient field so that brush strokes in a region of constant color (or near constant color) smoothly interpolate the directions defined at the region's boundaries (the difference is shown between Color Plates 4 and 5).

To accomplish this, gradient values are "thrown out" at pixel locations where the Magnitude((**Gx**,**Gy**)) is near zero. In this implementation, this is approximated by the test: |**Gx**|<3.0 and |**Gy**|<3.0, which was empirically found to be useful. The gradient at pixels with near zero gradient magnitude are then replaced by interpolating surrounding "good" gradients. The "good" values do not necessarily lie on a uniform grid, so generating points with cubic interpolation (or other closed-form solution) does not work here. An interpolant that does not assume uniformly spaced data in both directions is needed. In our implementation, **Gx** and **Gy** are interpolated using a thin-plate spline [Franke79], which is chosen for its smoothness characteristics.

Finally, at each brush stroke center (**cx**,**cy**), the modified gradient field components (**Gx**,**Gy**) are bilinearly interpolated. A direction angle is computed from this vector as arctan(**Gy**/**Gx**), 90° is added to it (to draw it normal to the gradient direction), and the Δ**theta** stored with the stroke is added to produce **theta**, the orientation of the drawn stroke (see Color Plate 5).

Using the normal to the gradient causes strokes to look glued to objects in a scene (it helps define their shape), especially when the objects are moving, rotating or changing shape. Keeping stroke orientation the same from frame to frame does not provide the same amount of perceived spatial and temporal coherence that is provided by using the normal to the gradient direction. Of course, keeping the strokes oriented along a particular constant direction remains an option.

C. Frame-to-Frame Coherence

In [Meier96], a temporally coherent technique is presented

Figure 5. Two frames and the optical flow field that maps pixels from one frame to another.

which employed “particles” on 3D objects as centers for brush strokes. By transforming these points into 2D, using the normal direction of 3D surfaces as guides for brush stroke orientations, and rendering strokes back to front as seen from the camera, temporally coherent animations were produced. However, input to our process is a video clip with no a priori information about pixel movement in the scene. Our technique uses standard vision techniques to produce an automatic technique to guide brush strokes in the direction of pixel movement.

To render the first frame, we use the process described in the previous two sections. In order to move the brush strokes from one frame to the next, we first calculate the optical flow between the two images. Optical flow methods are a subclass of motion estimation techniques and are based on the assumptions that illumination is constant and that occlusion can be ignored, that is, that the observed intensity changes are only due to the motion of the underlying objects. It should be noted that this assumption is quite invalid for many of our test sequences. However, the artifacts of these assumptions produce interesting results even when the assumptions aren't true. When objects appear or disappear, optical flow methods tend to mush together or stretch apart the image portions corresponding to these objects. This provides a pleasing temporal coherence when portions of objects appear or disappear.

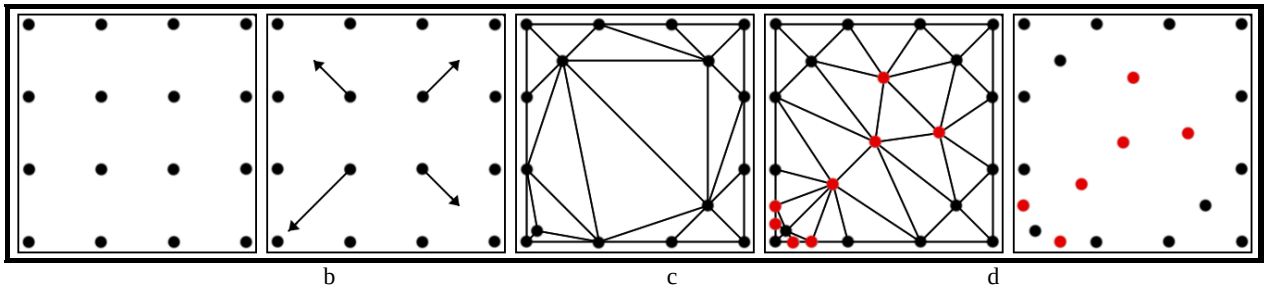
We chose the algorithm presented in [Bergen90] for its speed. This algorithm uses a gradient-based multi-resolution technique, employing a pyramid of successively low-passed versions of the gradient to help compute the optical flow. Presenting details concerning this optical flow method is beyond the scope of this paper.

The optical flow vector field is used as a displacement field to move the brush strokes (specifically, their centers) to new locations in a subsequent frame. The optical flow technique we implemented provides subpixel positioning, and this feature is exploited by moving brush strokes to subpixel locations. See Figure 5 for two images and the flow field that maps pixels in the first one to pixels in the second. After application of the displacements, some of the strokes may have been pushed from the edge of the image. The best match for a pixel will not be outside the image, but the algorithm may map edge pixels in one frame to an interior point in the next. We must make sure to generate new strokes near the image boundaries when this happens.
After application of the flow field to move the strokes, there also may be regions away from image boundaries that become unnecessarily dense with brush strokes or not dense enough. We want full coverage of the image with rendered brush strokes, so brush strokes are "too sparse" in our algorithm when there are pixels left untouched in the final rendered image.

To generate new brush strokes in regions that are too sparse, a Delaunay triangulation [Preparata85] using the previous frame's brush stroke centers (after application of the optical flow field) is generated using the methods described in [Shewchuk96] using source code available at [TriangleCode] (see Figures 6a,b,c). The particulars of the Delaunay triangulation is beyond the scope of this paper; however, it is important to know that the Delaunay triangulation covers the convex hull of the submitted points with triangles. By including the corners of the image in the point set, it is assured that the entire image will be covered with triangles (remember, as stated above, the optical flow may push strokes far away from the image boundaries).

The Delaunay triangulation by itself does not generate new points for brush strokes. However, after the Delaunay triangulation is performed, the mesh is subdivided so that there are no triangles with an area larger than maximum supplied area (as presented in [Shewchuk96]). By supplying an appropriate maximal area, new vertices are created which fill in the sparse areas and are subsequently used as new brush stroke centers. To produce Color Plate 5, the specific maximal area we supplied was 2.0 in pixels units squared (the antialiased lines for this plate were rendered with brush radii with a range of 1.5 to 2.0), a number found empirically to provide dense enough vertices. The maximum area may be tuned by the user if desired; for example, if the user wishes to have areas of the final image untouched by strokes. New brush strokes are created for the new vertices and a new random length and new variations for angle, color, intensity are determined and stored. See Figure 6d for the subsequent subdivision of the initial triangulation shown in Figure 6c.

Eliminating brush strokes in regions that are overly dense is desirable. After pushing strokes around frame after frame, brush strokes collect in image regions that "shrink." Over time this results in overly dense brush stroke regions, which then causes the rendering process to slow down tremendously. The amount of brush buildup depends of course on the specific video sequence. To dispose of brush strokes, the edge list of the triangulation is traversed (remember, each point in the



a b c d e

Figure 6.

a) Initial brush stroke positioning.
b) The four middle strokes are to be moved as shown.
c) Delaunay triangulation of the moved strokes
d) Red points show new vertices introduced as a result of satisfying the maximal area constraint.
e) The updated list of brush strokes. The original lower left corner brush stoke has been deleted because the distance between it and another original stroke satisfies the closeness test. Two of the potentially added new brush strokes have also been removed from the list.

triangulation is the center for an associated brush stroke). If the distance between the two points of an edge is less than a user-specified length, the corresponding brush stroke that is drawn closer to the back is discarded (a display list of strokes is kept, so there is an implicit front to back ordering). For Figure 6e and Color Plate 5, strokes were discarded when their centers were closer than 0.25 pixel units. As the edge list of the triangulation is traversed, if a point has been discarded we must be sure to perform the distance calculation with the point (and associated stroke) that replaced it. The triangulation provides the closest neighboring points to a given point, enabling a great reduction in the number of distance and comparison calculations.

At this point there are two lists of brush strokes: a list of "old" strokes (strokes moved and subsequently kept) from the previous frame, and the "new" strokes generated in sparse regions. Old stroke ordering (after throwing out unwanted strokes) is kept to provide temporal coherence. To place the new strokes on the list with the old strokes, the new strokes' order is randomized with respect to themselves. Then the new strokes are uniformly distributed among the old strokes. If the new strokes are simply painted behind the old strokes, undesirable effects can occur.

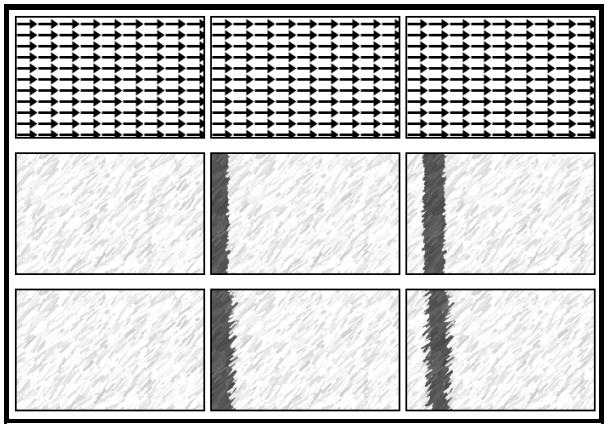


Figure 7. Top row shows a flow field. The second row shows the effects of placing new strokes behind old strokes, where new strokes are alternately coded dark and light. The third row demonstrates the effects of uniformly distributing the new strokes among the old ones.

For instance, in Figure 7 shows a flow field representing a pan of a video camera. The same figure demonstrates the results of putting new strokes behind old ones as well as uniformly distributing them. New strokes from frame to frame are alternately coded light and dark. Very clear edges appear if the new strokes are drawn behind the old ones. This is a problem, producing edges in the rendered image that may not be present in the original image. Uniformly distributing the new strokes produces much better results, effectively eliminating the problems encountered by painting new strokes behind the old ones. Distributing new strokes uniformly produces some temporal scintillation (strokes popping on top) but this was found to be preferable to the spatial anomalies that may otherwise occur.

After strokes are created, deleted and placed in their new positions, the base color of the stroke is retrieved from the image at the stroke center. The gradient field is determined for the new image in the sequence and used to calculate each brush's orientation. The stored delta values are then used to perturb these sampled values as described above and the next image in the rendered sequence is produced.

4. DISCUSSION

An algorithm for producing painterly animations from video clips has been presented. Brush strokes are clipped to edges detected in the original image sequence in an attempt to maintain silhouettes and other details present in the original images. Brush strokes are oriented normal to the gradient direction of the original image; a scattered data interpolation technique is used to interpolate the gradient field in areas where the magnitude of the gradient is near zero. Finally, a brush stroke list is maintained and manipulated through the use of optical flow fields to enhance temporal coherence.

The numbers presented in the paper represent a particular implementation. For the image sequence represented by the technique to produce Color Plate 5 (brush radii in the range [1.5-2.0], brush lengths in the range [4,10] and a maximal area constraint of 2.0), 76800 strokes were used to start the process (= 640/2 * 480/2). As the process continued, the stroke count averaged 120,000. Time to produce each frame averaged 81 seconds on a Macintosh 8500 running at 180 MHz.

Of course the specific parameters to the brush and the specific image processing and rendering techniques may be manipulated to produce different results. A fatter brush stroke radius of 8 produced the image in Color Plate 6, and textured brush strokes produced Color Plates 7 and 8. In the future different color assignment techniques are planned (such as averaging the colors under a particular brush stroke to generate its color).

We see this algorithm as an important step in automatically producing temporally coherent "painterly" animations. However, because we paint some of the new strokes in front of old strokes, the animations can scintillate. Whether we can avoid this and not introduce spatial anomalies remains to be determined. Also, because we clip lines to edges in the original video sequence, the presence of noise in the original video will cause the derived edges to scintillate, which in turn causes the brush stokes to scintillate. The brush stroke placement from frame to frame is not perfect either, and is only as good as the underlying motion estimation technique used. The technique we used does fairly well but can only do so much without any advanced knowledge of the objects in the scene. In particular, brush strokes can sometimes seem to swim in areas of near constant intensity.

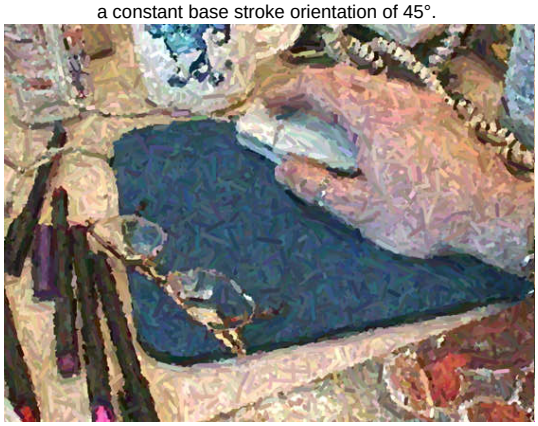
Further directions may include implementing other rendering, image processing and vision techniques to produce other artistic styles. Applying the techniques to 3D objects to produce painterly renderings would be interesting (as in [Meier96]), and would enable animations with much greater temporal coherence since object movement is known a priori.

For the first time, temporal coherence in video segments is used to drive brush stoke placement for a painterly style effect. After a few initial decisions, such as what the brush stroke length, radius and texture should be; whether or not to use the gradient for brush stroke orientation; what filter kernels should be used; providing distances and areas for the closeness and sparseness tests, our system process the video *automatically*. Hopefully this technique proves easy enough for those who do not have the time, desire, or talent to hand-animate a

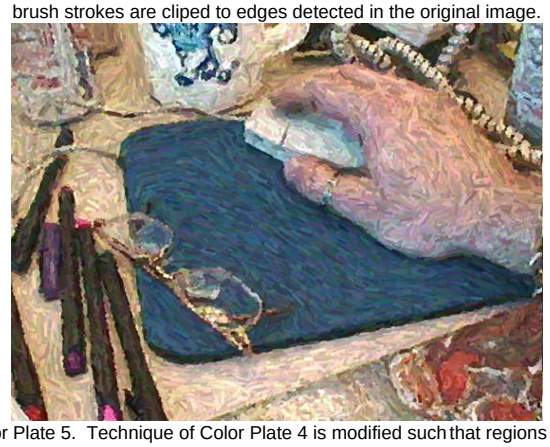
Color Plate 1. An original image. Plates 1-8 are 640x480 pixels.

Color Plate 2. Processed image using no brush stroke clipping and a constant base stroke orientation of 45°.

Color Plate 3. Technique of Color Plate 2 is modified so that brush strokes are cliped to edges detected in the original image.

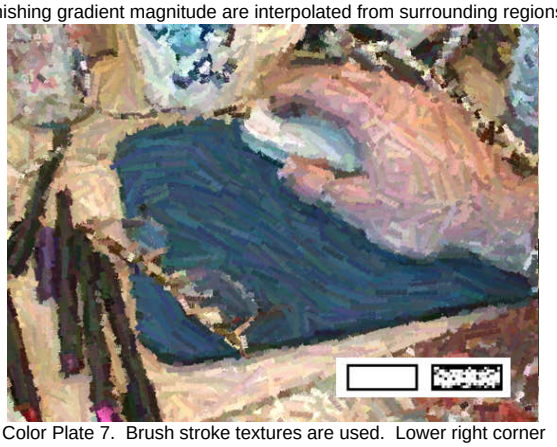


Color Plate 4. Technique of Color Plate 3 is modified to orient strokes using a gradient-based technique.

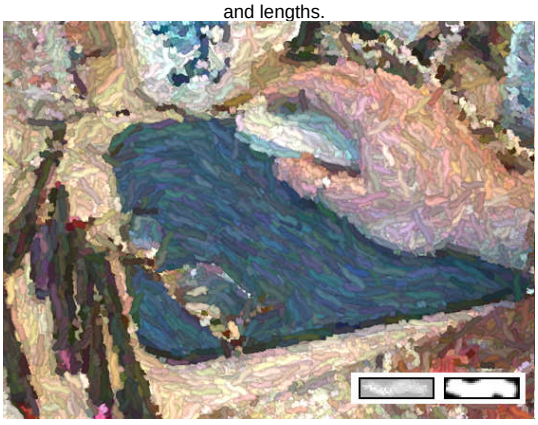


Color Plate 5. Technique of Color Plate 4 is modified such that regions with vanishing gradient magnitude are interpolated from surrounding regions.

Color Plate 6. Image produced using larger brush stroke radii and lengths.



Color Plate 7. Brush stroke textures are used. Lower right corner shows basic brush intensity and alpha.



Color Plate 8. Another brush stroke texture is demonstrated.

sequence, but is also powerful enough to be part of the battery of tools a trained artist might use.

5. ACKNOWLEDGEMENTS

Thanks to Apple Research Labs for sponsoring this work, Gavin Miller for many ideas, and the reviewers who provided a careful reading of this paper.

6. REFERENCES

[Bergen90] Bergen, J. R. and R. Hingorani. "Hierarchical motion-based frame rate conversion," David Sarnoff Research Center, Princeton, N. J.

[Foley84] Foley, James and Adries Van Dam. Fundamentals of Interactive Computer Graphics. Addison-Wesley, Reading, Massachusetts, 1984.

[Franke79] Franke, F. "A Critical Comparison of Some Methods for Interpolation of Scattered Data," Report NPS-53-79-03 of the Naval Postgraduate School, Monterey, CA. Obtained from the U.S Department of Commerce, National Technical Information Service.

[Haeberli90] Haeberli, Paul. "Paint By Numbers: Abstract Image Representations," Computer Graphics, SIGGRAPH Annual Conference Proceedings 1990, pp. 207-214.

[Hsu94] Hsu, Siu Chi and Irene Lee. "Drawing and Animation Using Skeletal Strokes," Computer Graphics, SIGGRAPH Annual Conference Proceedings 1994, pp. 109-118.

[Jain95] Jain, Ramesh, Rangachar Kasturi, and Brian Schunck. Machine Vision. McGraw-Hill, Inc. New York, 1995.

[Kingston80] Kingston, Jeremy. Arts and Artists. Book Club Associates, London, 1980. pp. 98-99.

[Meier96] Meier, Barbara. "Painterly Rendering for Animation," Computer Graphics, SIGGRAPH Annual Conference Proceedings 1996, pp. 477-484.

[Preparata85] Preparata, Franco, Michal Ian Shamos, Computational Geometry, An Introduction, Springer-Veralg, 1985.

[Salisbury94] Salisbury, Michael, Sean Anderson, Ronen Barzel, and David Salesin. "Interactive Pen-and-Ink Illustration", Computer Graphics, SIGGRAPH Annual Conference Proceedings 1994, pp. 101-108.

[Salisbury96] Salisbury, Mike, Corin Anderson, Dani Lischinski, and David Salesin. "Scale-Dependent Reproduction of Pen-and-Ink Illustrations", Computer Graphics, SIGGRAPH Annual Conference Proceedings 1996, pp. 461-468.

[Shewchuk96] ShewChuk, Jonathan. "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator," First Workshop on Applied Computational Geometry, Association for Computing Machinery, May, 1996, pp. 124-133.

[TriangleCode] Code for reference [Shewchuk96] available at http://www.cs.cmu.edu/~quake/triangle.html.

Appendix A. STROKE CLIPPING

The center of the stroke is given by (**cx**,**cy**) and the direction of the stroke is given by (**dirx**,**diry**). This process determines (**x1**,**y1**) and (**x2**,**y2**), the endpoints of the stroke clipped to edges in the image.

The Sobel filtered intensity image is sampled in steps of unit length in order to detect edges. To determine (**x1**,**y1**):

a. set (**x1**,**y1**) to (**cx**,**cy**)

b. bilinearly sample the Sobel filtered intensity image at (**x1**,**y1**), and set **lastSample** to this value

c. set (**tempx**,**tempy**) to (**x1**+**dirx**, **y1**+**diry**), taking a unit step in the orientation direction.

d. if (dist((**x1**,**y1**),(**tempx**,**tempy**)) > (**length** of stroke)/2, then stop

e. bilinearly sample the Sobel image at (**tempx**,**tempy**), and set **newSample** to this value

f. if (**newSample** < **lastSample**) then stop

g. set (**x1**,**y1**) to (**tempx**,**tempy**)

h. set **lastSample** to **newSample**

i. go to step c

At the end of this process, the endpoint (**x1**,**y1**) of the line in one direction has been determined. To find (**x2**,**y2**), the endpoint in the other direction, set (**dirx**,**diry**) to (-**dirx**, -**diry**) and repeat the above process.

Color Plate 9. Another image produced with the technique.