Fluid Simulation as a Tool for Painterly Animation

Sven C. Olsen* Swarthmore College

Bruce A. Maxwell† Swarthmore College

Abstract

In this paper we combine recent work from two different subfields of computer graphics: semi-Lagrangian fluid simulation and painterly rendering. We have implemented a painterly rendering system in which users can select regions of an image in which a fluid field will be used to inform the placement and rendering of brush strokes, and thus generate renderings which include long curved strokes that inherit many of the aesthetically pleasing properties associated with fluid fields. The attractive properties of fluid fields are most apparent when those fields are in motion, and our stroke based renderer builds on recent techniques for creating painterly animations. We have combined fluid based stroke motion with other painterly animation techniques in order to create animations based on video clips. We have also developed an interesting technique in which an image is warped according to a fluid field and a painterly animation generated based on the resulting distortion.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms; I.6.3 [Simulation and Modeling]: Applications

Keywords: Non-photorealistic rendering, Fluid Simulation

1 Introduction

In 1963, Ivan Sutherland published his famous dissertation *Sketchpad: A man-machine graphical communication system*. And thus was born the field of computer graphics. Since then it has been widely recognized that computers have enormous potential as artistic tools. This paper, like many others in the discipline of computer graphics, concerns itself with one attempt to realize that potential.

We follow in the tradition of stroke-based non-photorealistic rendering which began with Haelberi [1990]. Haelberi's system allowed users to generate impressionist images by creating a number of brush strokes colored according to an input image. Unlike more recent papers such as [Curtis et al. 1997], Haelberi did not try to make a system that precisely simulated the characteristics of actual paint, rather his goal was to allow users to quickly create images which possessed many of the visual merits associated with paintings.

Haelberi's stroke based system inspired a number of impressionistic image filters which have now become commonly used artistic tools. Indeed some descendent of Haelberi's original application is included in almost every modern image editor.

Figure 1: Detail of *The Starry Night*, Vincent Van Gogh, 1889.

The inspiration for this paper comes from two famous descendents of Haelberi's system. Litwinowicz and Hertzmann both produce images in which large, homogenous areas of the source images may be rendered using long flowing brush strokes [Litwinowicz 1997; Hertzmann 1998]. Hertzmann explicitly sets out to create long curved brush strokes, while in the case of Litwinowicz a more modest flowing effect arises as a result of his use of thin plate splines for the purpose of color gradient interpolation. In both systems the resulting effects can be reminiscent of the brush stroke patterns seen in some expressionistic paintings, strokes fluidly curling and turning around each other. However, both systems align their brush strokes normal to the color gradient of the source image, and therefore the sense of turbulence which might otherwise be conveyed by the stroke patterns is always muted, never approaching the wild whirlpools and vortices which are so striking in a painting like *The Starry Night*.

Our goal is to create painterly video clips *a la* [Litwinowicz 1997] in which the brush strokes possess a turbulent character similar to that suggested by some of Van Gogh's paintings. To this end we introduce a novel, non-gradient based stroke placement method. When rendering and controlling our brush strokes, we make use of a semi-lagrangian fluid simulator, very similar to the one introduced to the graphics community by Stam [1999]. Using simulated fluid fields in the context of a stroke based non-photorealistic rendering system allows us to produce stroke behaviors which are visually appealing, and which offer expressive potential beyond that of gradient based techniques.

2 Related Work

Litwinowicz introduced a system that expanded on Haelberi's stroke based technique to allow for video based non-photorealistic animations[Litwinowicz 1997]. Like many other non-photo realistic systems that output video [Meier 1996; Klein et al. 2000;

*e-mail:sven@sccs.swarthmore.edu

†e-mail:maxwell@swarthmore.edu

Kalnins et al. 2003], Litwinowicz had to find a way to deal with the problem of timewise coherency. Litwinowicz dealt with the issue of coherency between stroke placements in successive frames by cleverly combining a quality controlled Delaunay triangulation and an optical flow algorithm. Like Litwinowicz's, our own system acts on sequences of video, and we have adopted many features of his coherency scheme. Litwinowicz also introduced a strategy for edge based brush clipping- another feature that we use in our own work.

The goal of [Hertzmann 1998] was to create a system in which users could enter a description of the style of painting that they wanted to use, and the system would then paint a version of the input image using that style. While his methods and goals differ considerably from our own, Hertzmann's is the only other research paper that we are aware of which explicitly sets out to render images using long, curved brushes.

We are not the first paper to coopt fluid flow simulations for the purposes of non-photorealistic rendering. [Witting 1999] explains how DreamWorks has used fluid simulations in the creation of their animated feature *The Prince of Egypt*. However, as far as we know, we are the first to use fluid simulation in the context of a stroke based system. We are also the first to expand [Litwinowicz 1997] to incorporate a non-gradient based stroke alignment strategy and to substitute alternate vector fields in regions of homogenous color.

Several commercial programs for creating stroke based animations from video exist [Synthetik Software, Inc. 2002; RE: Vision Effects 2003], though neither program is designed to create animations using long, curved brush strokes.

In this paper we combine recent work from two different subfields of computer graphics: semi-Lagrangian fluid simulation and painterly rendering. We have implemented a painterly rendering system in which users can select regions of an image in which a fluid field will be used to inform the placement and rendering of brush strokes, and thus generate renderings which include long curved strokes that inherit many of the aesthetically pleasing properties associated with fluid fields. The attractive properties of fluid fields are most apparent when those fields are in motion, and our stroke based renderer builds on recent techniques for creating painterly animations. We have combined fluid based stroke motion with other painterly animation techniques in order to create animations based on video clips. We have also developed an interesting technique in which an image is warped according to a fluid field and a painterly animation generated based on the resulting distortion.

3 The Process

3.1 Creating regions

Our goal is to be able to render certain regions of the video, such as sky or other parts of the background, using strokes which are aligned and advected by a fluid field. However, it is generally not desirable to use such fluid field based rendering methods in areas of the video containing more detailed features, such as human faces. Therefore we need to divide each frame of the video into two regions. In the non-fluid region, our rendering technique is very nearly a pure reimplementation of Litwinowicz's system. In the fluid region there are a number of changes which have been made to Litwinowicz's basic framework in order take advantage of the information provided by the simulated fluid fields.

The regions of video which we generally want to render using fluid fields tend be large areas of homogenous color. One could use video analysis techniques to identify and track large homogeneous regions to be rendered using the fluid flow fields. However, a fully automated region identification and tracking system is not necessarily desirable. While large homogenous regions are the natural candidates for fluid flow based rendering, it is not necessarily the case that we would want those regions and no others to be rendered using fluid fields. We feel that the decision to render a given region using fluid fields is a choice best left to the artistic sensibilities of the user, and therefore we prefer an interface which allows users to identify and label regions of the video as fluid or nonfluid as they see fit.

When drawing our brush strokes, we clip them to the edges of the objects in the image as in [Litwinowicz 1997]. Object edges are defined using Sobel values- the magnitudes of an approximate color gradient (though as we explain later, our Sobel values have been modified in order to minimize noise). Currently we use a region growing technique to find and track the boundaries of the fluid regions in video sequences. The seed pixel for each fluid region is selected by the user, and should be a pixel that will always be within the fluid region. Then, using a standard region growing algorithm, the system grows the seed pixel until the region is bounded by pixels with gradient magnitudes larger than a threshold G. For this implementation we used G equal to 10. This method works well for images with large homogeneous regions such as interior walls or sky, which are the kinds of regions where the curving brush strokes and fluid vortices create much more interest than typical painterly rendering algorithms.

It is worth noting that the manner in which we solve the region tracking problem is completely independent from the rest of our system - indeed in implementing our application we have considered both the source video and a set of boolean maps dividing each video frame into fluid and non-fluid regions to be inputs to the rest of the algorithm. Source videos more complicated than those which we have used would most likely require more sophisticated region tracking algorithms.

3.2 Stroke Motion

For each frame of the video, we create a painterly rendering using a set of brush strokes. After each frame has been rendered, we advect the brushstrokes using the fluid field if the stroke lies in the fluid region, or using an optical flow field if it is in the non-fluid region. This allows us to achieve a measure of visual continuity between frames since each stroke represents a slightly displaced stroke from an earlier frame.

The problem facing any system in which strokes are advected between frames is that as the strokes move their density tends to become far too high in some areas, and far too low in others. One existing solution to this problem involves first finding a Delaunay triangulation for the point set containing all stroke positions and the corners of the image [Litwinowicz 1997]. To prevent the strokes from becoming too sparse, a quality control constraint is used as in [Shewchuk 1996], forcing new strokes to be inserted as is needed to prevent any triangle in the triangulation from growing beyond a given area. To prevent the strokes from becoming too dense, if any pair of strokes is closer than a given distance, the lower-rendered of the two strokes is deleted. We have adopted Litwinowicz's triangulation based density control method, though we use a simplified version of Shewchuk's area based quality control logic.[1] It is important to note that while different fields advect the strokes, the density control algorithms do not distinguish between fluid and non-fluid regions.

In the fluid regions, the fluid simulator provides the motion vectors for a video sequence. In the non-fluid regions, however, we follow the current practice of using optical flow analysis to generate the vector field for frame-to-frame coherence of brush strokes [Litwinowicz 1997]. While Litwinowicz used Bergen's hierarchical algorithm to calculate the optical flow fields, we found that in the

[1]The details of our quality control algorithm are explained in section 3.7.1

case of our own videos, Bergen's algorithm tended to produce troublesomely inaccurate fields, which led to large, inaccurate stroke placements. Instead, we found that Horn and Schunck's optical flow algorithm creates much nicer, smoother fields for our video sequences [Horn and Schunck 1981]. This observation is in keeping with a comparative analysis that suggests that, despite its venerable age, Horn and Schunck is one of the best optical flow algorithms available [Galvin et al. 1998].

3.3 Stroke Rendering

Our renderings of each scene are created using texture mapped brush strokes. Information about each stroke is stored in a vector, and the strokes themselves are drawn as texture mapped polygons using OpenGL.

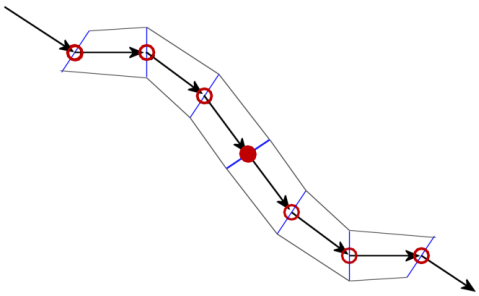

Figure 2: **Brush Polygon:** Starting from the stroke position point (solid red), two sets of center points (red outline) are created by moving in small, fixed length steps forward and backward through the fluid field. Field vectors are drawn in black. Polygon edges are drawn in gray.

When rendering strokes in the fluid region we want the strokes to curve according to the underlying fluid field. To generate the curved stroke we use an algorithm very similar to the particle tracing method which we will later use in the fluid simulator. The first step is to calculate a set of stroke center points. Our first center point is the position of the stroke itself. To get the rest of the points, we first use bilinear interpolation to find the fluid field velocity underneath the position of the stroke, and then move some small distance in the direction of the velocity vector (typically half a grid square length); that gives us our second center point. We then recalculate the velocity, and continue the pattern of moving and recalculating until the stroke is either clipped or has reached a desired length. We then use the same process to get a second set of stroke center points, this time moving backwards through the fluid field. Once the center point calculation is complete we calculate the points in the polygon, creating two polygon points for each center point by moving out by half the desired stroke width from the center of the stroke, in a direction perpendicular to the velocity vector calculated at the center point.

Stroke rendering in the non-fluid region is much simpler. Our polygons all rectangles, drawn with a base orientation of 45 degrees plus or minus some random permutation value.

3.4 Stroke Clipping

Stroke clipping methods can go a long way in helping to define the object in a scene. When brush strokes overlap object boarders, the result is that the objects in the scene seem to blur into each other. Clipping brush strokes to object edges can eliminate this blurring effect.

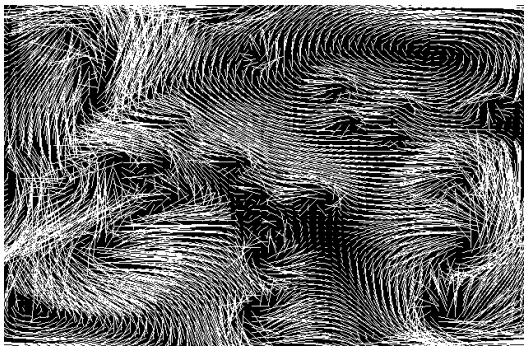

Figure 3: The velocity field generated by our fluid simulation

Figure 4: The source frame to be rendered

Figure 5: The region definition (white is fluid, black is nonfluid)

Figure 6: The rendered frame

In [Litwinowicz 1997] Sobel filters are used to determine the edges of objects in the scene. We adopt Litwinowicz's terminology, and define the "Sobel value" to be the magnitude of the color gradient as calculated using Sobel convolution matrices [Fisher et al. 1994]. Thus:

$$SobelValue(x,y) = \sqrt{Sx(x,y)^2 + Sy(x,y)^2}. \quad (1)$$

Once the Sobel values have been calculated, a stroke is clipped in the event that it ever passes over a pixel in which the Sobel value is lower than that in the pixel before it. In order to minimize the number of false edges, it can be wise to avoid calculating Sobel values directly from the underlying image. Litwinowicz generated his Sobel values by first blurring the image using a Gaussian kernel, and then running the Sobel convolution matrices over the blurred image.

We have found that even given the initial blur, the Sobel values produced by our images still tend to be a little noisy, and therefore we go a step farther in order to clean up the edge information. We run a second Gaussian blur over our Sobel values, and then get rid of what remains of the noise by setting all Sobel values less than 6 to 0. Because most of the noise in the Sobel values takes the form of thin lines, a small kernel is sufficient to eliminate it. We have been able to consistently produce nice Sobel values by using a kernel of radius 8 in the first blur, and a kernel of radius 4 in the second blur. The additional cleanup of the Sobel values is especially useful given that our region tracking methods also benefit from it.

3.4.1 Stroke Coloring

Stroke coloring presents an interesting challenge. For the strokes in the nonfluid region, we know that whatever optical flow algorithm we use is going to be imperfect. Therefore keeping stroke color constant for the life of the stroke will lead to a steadily growing number of obviously misplaced brush strokes, destroying the quality of the resulting video. On the other hand, using the color in the underlying image to determine stroke color means that the color of each stroke changes from frame to frame, which necessarily implies less timewise coherency. One proven stroke coloring strategy is to store intensity and RGB color permutation values for each stroke [Litwinowicz 1997]. The base brush color is reset every frame to match the underlying image, but by keeping the permutation values constant for the life of the stroke we increase the degree of continuity between strokes in successive frames. The random color permutations also serve to give the rendering a more handcrafted appearance.

In our own system we have used a variation on Litwinowicz's coloring technique. For strokes inside of the fluid region, we expect that the underlying color will be more or less constant, and are more interested in frame to frame coherency than in preserving the details of color in the underlying image. Thus we only recolor the stroke if its color exceeds a given square sum difference (SSD) with the underlying image color. This has the effect of making it very likely that a stroke will keep the same color as it moves from frame to frame. For strokes in the nonfluid region, we use a smaller SSD threshold, reflecting the increased extent to which we are willing to sacrifice coherency for accurate stroke coloring.

3.5 Stroke Data

We store a number of values for each brush stroke. Our stroke motion algorithms require that we store the position of each stroke. In order to render each stroke, we need to know it's width and color. We also the store delta values indicating how much to perturb stroke color and intensity. When a stroke is in the nonfluid region, we may wish to perturb its orientation, and therefore that delta value also needs to be stored. Finally, we store a boolean value indicating whether the stroke was last rendered as part of the fluid or nonfluid region. This allows us to reinitialize the characteristic values of a stroke if it moves from one region to the other. Such reinitialization is primarily useful because it allows us to vary the ranges used to initialize stroke width and delta values from region to region. In our experience it is preferable to have smaller strokes in the nonfluid region in order to preserve the higher level of detail.

We initialize our brush stroke positions using a 2 dimensional Halton sequence as in [Keller 1997]. Given that the stroke density control algorithms quickly take care of all distribution issues, any pseudorandom stroke initialization method would serve to generate the initial stroke placements. But as computer graphics researchers we have a great affection for Halton sequences, and their low discrepancy properties saves us a couple processor cycles during the first iteration of our density control algorithms.

3.6 Fluid Simulation

Our fluid fields are simulated using a semi-Lagrangian fluid solver of the type which has recently become popular in computer graphics. Semi-Lagrangian solvers sacrifice some amount of physical accuracy in return for speed and stability. This makes them well suited to many graphics applications, as in graphics we are usually content to create something that viewers will perceive as a fluid, rather than something that precisely models fluid properties. In our own case we take this attitude a step further: There is no need for the fields which we generate to approximate actual fluid behavior, our only goal is to create fields which are visually pleasing. The brush alignments in the Van Gogh paintings which provide the main inspiration for our work are quite certainly not derived from any physically accurate fluid simulation. Thus our aims imply that we can freely interfere with the mathematics behind our simulation in hopes of creating more visually satisfying fields.

3.6.1 Voricity Confinement

Voricity confinement is a technique used to remedy one of the inherent inaccuracies of semi-Lagrangian fluid solvers. The technique was invented by Steinhoff in the early 90s, and it was introduced to the graphics community by [Fedkiw et al. 2001]. Semi-Lagrangian solvers tend to create fields in which whirlpools die out more quickly than they should; the technique of voricity confinement addresses this problem by adding in an artificial force which acts to magnify the curl of the field and thus reintroduce the turbulent effects which would otherwise be lost.

The confinement force is given by

$$f_{conf} = \varepsilon h(\mathbf{N} \times \omega) \quad (2)$$

Where h is a parameter that indicates the density of the simulation grid, ω is the curl of the velocity field and $\mathbf{N}$ is the normalized gradient of $|\omega|$.

When this technique is used in the context of a fluid modeling application, the value of ε that is used must be carefully calculated to be consistent with physical reality. But in our case, we are free to reintroduces as much turbulence into the field as we find appealing. Sometimes this leads us to chose an ε much greater than would be appropriate in a physical simulation.

3.6.2 Particle Tracing

The "Lagrangian" component of semi-Lagrangian solvers is the particle tracer used in the advection step of the solver. A fast and simple solver can be built by simply using linear tracing [Stam 2003]. In [Fedkiw et al. 2001], a much more sophisticated monotonic cubic interpolation method is used. We take a rather different

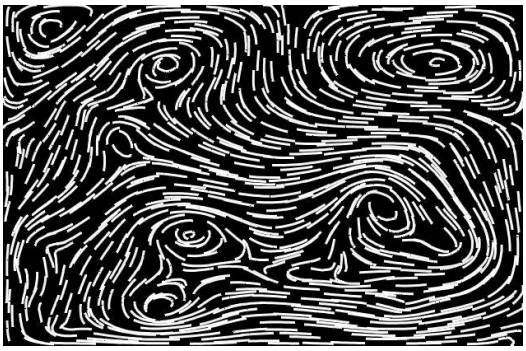

Figure 7: Strokes rendered using a simulation with a physically plausible $\varepsilon = 1$

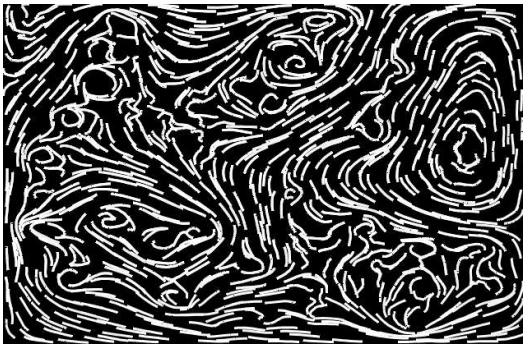

Figure 8: Strokes rendered using a simulation with $\varepsilon = 8$, greatly increasing the number of small vortices

route, and perform our particle tracing using the same strategy of successive small steps which we use when drawing curved brush strokes. In the brush drawing case we needed to advect a particle for a given distance, however in this situation we need the particle advection to continue for a given Δt. But the transitioning from distance to time is easy enough, as we can use the speed of the particle to calculate a time cost for every step in the advection.

3.6.3 Boundaries

Depending on the effect we want to create, it can be useful to impose boundaries on the fluid flow. If we have the fluid field consider the nonfluid region as a boundary to flow, then the brush strokes will flow around the other objects in the scene. The other option is to impose no boundaries on the flow, which creates the impression that the strokes are flowing behind the strokes in the nonfluid region.

Again, because we have little interest in physical accuracy, we have settled for a very simple implementation of fluid bounds. As suggested in [Stam 2003], we simply force the flow to be zero at all grid points inside of the boundary, and in combination with our particle tracing method, that is sufficient to create a plausible simulated boundary to the flow. We have used a Jacobian solver to solve the sparse linear systems present in the projection and diffusion steps of the solver (see [Stam 1999]). Properly the boundary points ought to impact the matrices fed into the linear solver, however the character of the Jacobian solver is such that we can ignore the effects that the boundaries ought to have on the linear systems, sacrificing a bit of accuracy in return for a simpler boundary implementation.[2]

[2]Our experience has been that the Gauss-Seidel solver used in [Stam 2003] becomes problematic if used with our simple boundary implementation, this is why we switched to the Jacobian solver.

Semi-Lagrangian solvers completely separate the evolution of the velocity field from that of the material to be advected. Thus, the solver in [Stam 1999] was actually two independent solvers, one which calculated the evolution of the velocity field, and one which calculated the evolution of the smoke density field given the current and past velocity fields. In visual simulation of smoke, two density fields were advected by the velocity field, one representing smoke density, and one representing temperature. (The temperature values were used to calculate forces to be applied to the velocity field.) In theory, any number of scalar fields can be advected by the velocity fields. In our own case, we are only interested in the evolution of the fluid velocity field, and thus we are properly only using half of the solver presented in [Stam 1999].

3.7 Still Image Based Rendering

Thus far we have focused on video based renderings, but interesting effects can be achieved by modifying the algorithms to work on still images. Still images created by simply rendering a give source image as if it were a single frame from a video can be quite attractive in their own right. Animations can be created from a single image by using the same source image for all the frames of an animation (in this case the strokes in the fluid region will move, while strokes in the rest of the image will stay still).

3.7.1 Fluid Warping Animations

We can also use the fluid field to warp a given image over a series of frames. The resulting animations show the progressive distortion of the image under the effects of the fluid field. Many image editors are able to create animations by warping images, but our combination of fluid fields and stroke based rendering allows us to create warping animations with some unique qualities.

In the warped image case we eliminate nonfluid regions altogether. We initialize the strokes based on our source image, and advect all brushes using the fluid field. After the first frame of animation, the stroke colors no longer match up with the original image, which presents a problem when new strokes need to be added, as querying the new color from the source image quickly destroys any semblances of shapes or objects.

Because we are using a simplified version of Shewchuk's quality control algorithms, there is a simple coloring option available to us. In our quality control phase we walk through all the triangles in the triangulation and create a new stroke in the case that the area of the triangles exceeds a given maximum value. The new stroke is placed at the circomcenter of the overly large triangle, and forced back inside of the bounds of the image in the case that it lies outside of them. Thus we can determine the color for any new stroke using the colors of the three strokes which formed the overly large triangle. A scanline interpolation method would be the preferred way to calculated the new stroke's color, but scanline interpolation is undesirable in the occasional cases in which the circomcenter lies outside of the image boundaries. Simply averaging the strokes' colors is a more robust method, and given sufficiently high stroke density, the results of averaging are not perceptibly inferior to those of scanline interpolation.

The stroke clipping used in video based rendering allow the objects in the scene to retain their definition, despite the blurring effects which normally result from stroke based rendering. While the objects in a still image based rendering are going to be distorted, we don't necessarily want to blur the objects' boundaries. It is possible to adapt the clipping methods of video based rendering to work in the context of warped images. We use the source image to create Sobel values as per the video based case. Each time we advect the brushes, we use the fluid field to warp the Sobel image - thus we can continue to clip the brush strokes to object boundaries, even as

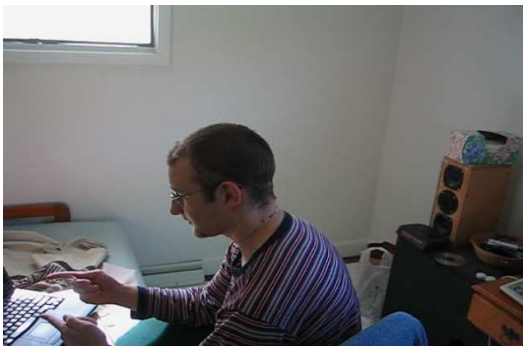

Figure 9: A still image used to create a warping animation

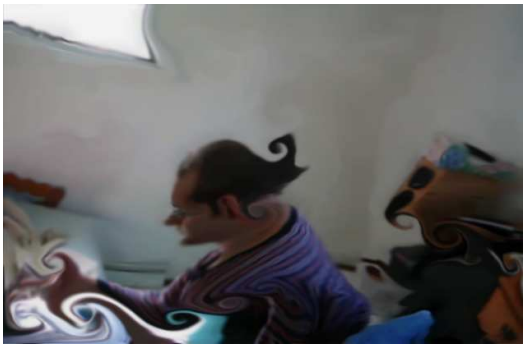

Figure 10: The warped source image

Figure 11: The warped Sobel values

Figure 12: A frame from the animation

those boundaries are distorted. For each frame of the animation we create a new Sobel image, the value at each pixel is calculated by starting at pixel's position and tracing back through the fluid field. The particle tracing algorithm is identical to the one used in the advection step of the solver. We perform a bilinear interpolation on the Sobel values from the previous frame to find the Sobel value under the advected pixel position.

Warping the Sobel values in this way adds a notable computational cost to the animation process. However, if we are already calculating advected positions for each pixel in the image for the purpose of Sobel warping, we can warp the source image at very little additional cost. This gives us an alternative technique for calculating the color of new strokes, we can simply query the warped source image. Warping the source image also gives us the option of recolor the brushes if they depart from the color in the underlying image, which can result in more coherent animations.

The warping case seems to create situations in which edge clipping is more effectively done by clipping any stroke which encounters a sufficiently high Sobel value (rather clipping if the Sobel value decreases as in [Litwinowicz 1997]).

4 Results

We have written our program using C++ under Microsoft Visual Studio 6, using a plethora of freely available libraries and source code. In keeping with the tradition of Litwinowicz's original paper, we use Jonathan Shewchuk's *Triangle* for the Delaunay triangulation [Shewchuk 2002]. Our fluid simulator is an extended version of the minimalistic implementation provided with [Stam 2003]. We have used David Minnen's publicly available implementation of Bergen's optical flow algorithms [Minnen 2002], while all other optical flow algorithms use Intel's OpenCV computer vision package. We use wxWindows for our user interface, OpenGL to handle the actual stroke rendering, and SGI's Image Format Library (IFL) for writing and reading all image files.

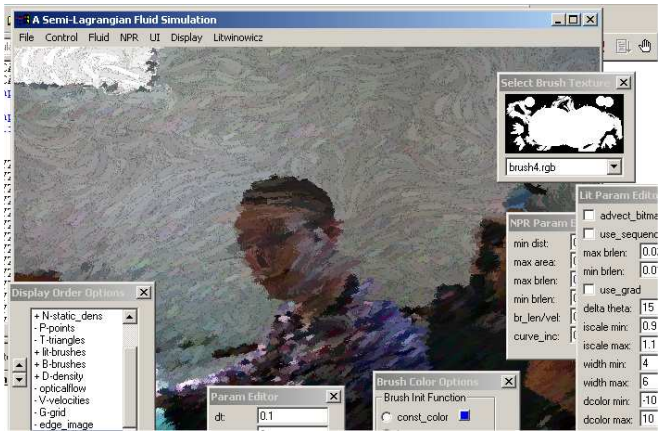

Figure 13: Our Application

Our application consists of a window that can interact with the fluid simulation and render the animations. There are a number of dialogs for editing the parameters governing the fluid simulation and stroke initialization / display. Our application is designed and built purely to serve our needs as graphics researchers; which means that in these dialogs every imaginable parameter is immediately exposed to the user. The bits of the program involving Sobel values and region tracking have been implemented using a collection of small command line programs and patched together using perl scripts. Were we targeting our program at casual users, we

would almost certainly want to obscure the parameters behind a couple of default configurations.

Our program has been developed and tested using an Athlon 1Ghz desktop computer. The main costs of rendering a frame are Sobel value generation and the density control logic. Generating the Sobel values for a 640x480 images takes 10 seconds. The time required for the density control logic depends on the desired brush density; density control in our demo animations typically takes around 20 seconds per frame. But for sufficiently sparse stroke distributions the system can run in close to real time. Using the fluid field to warp a 640x480 image typically takes about 2 seconds.

Along with the traditional mouse based velocity insertion, our fluid simulator includes some simple field editing tools which allow users to freeze the velocity values in parts of the simulation - this can be used either to create a final field with a desired overall pattern (freezing parts of the field when they exhibit the desired properties, and then unfreezing the entire field before rendering the video), or to stop portions of the field from evolving while the video is rendered. Similarly the application allows users to halt all of the density control and stroke advection logic while still running the fluid simulation, which allows us to see the effects different fields on the given strokes. In this way interesting movies can quickly be created from still images even without any stroke advection - and without the cost of density control logic to slow us down such videos can be generated in real time.

Our fluid simulator uses a 64x64 grid. Using a grid of this size allows the fluid simulation to run in real time, which in turn makes it easy to interact with the fluid fields. Using a larger grid would allow for more detailed fields, but once the simulation slows to around an update per second, interacting with the fluid field becomes much more cumbersome.

When creating our renderings we have often found it useful to display the source image behind the brush strokes which we have generated. Even given very high stroke densities, vortices in the fluid field will tend to have a small spot at their center which is never covered by any brush strokes; displaying the source image behind the strokes serves to cover up what would otherwise be a number distracting black spots at the center of the vortices. Attractive effects can also be achieved by displaying the source image behind a set of strokes which do not completely cover the underlying time. This strategy can result in renderings which give a paint-like impression while still persevering many of the details of the original underlying image.

5 Discussion and Future Work

We have implemented a novel painterly rendering method informed by fluid simulations. Fluid fields have demonstrated themselves to be an artistically effective tool for informing stroke alignment in areas of homogenous color, and our use of fluid velocity fields to advect strokes between frames has produced painterly animations in which stroke motion is pleasingly smooth and visually coherent. We have presented stroke rendering techniques specialized for use with fluid fields, and fluid simulation algorithms specialized for use with painterly rendering.

One natural avenue for future work would be give the present system the ability to derive fluid fields from the source video, thus potentially reducing the amount of user input necessary to generate animations. For example, the color values in the images might be used to infer forces to be applied to the fluid field. In this way animations could be created without the user ever needing to directly edit a fluid field.

How exactly one might use color information or to infer a set of forces is an interesting question. One strategy would be to follow in the path of [Witting 1999], and use the colors in the image to infer a temperature field, which could the in turn be used to apply forces to the fluid field. However, the question of how to best infer forces depends heavily on the kind of effect that the user wishes to achieve. An ideal system would provide a number of potential force inference methods, each tailored to create fields of a specific character.

In a similar vein, it could be interesting to explore means in which high level information provided by the user might be used to infer forces to be applied to the fields. It is tempting to consider adapting the keyframing techniques from [Treuille et al. 2003] to function of fluid velocities, rather than smoke density. In this way one might generate a set of forces such that the velocity field (and thus the implied stroke orientations) would reach certain states at specific frames of the animation. If one could get such a keyframing system working, it would be possible to seamlessly transition between fluid and gradient based stroke orientation methods.

In this paper we have demonstrated how recent mathematical models developed for the task of fluid simulation can be used to good artistic effect in the realm of painterly rendering. It is our hope that our work will provoke further exploration of new methods for orienting and controlling the brush stokes, and open the way for new artistic endeavors.

References

BERGEN, J., ANANDA, P., HANNA, K., AND HINGORANI, R. 1992. Hierarchical model-based motion estimation. In *Proceedings of the 2nd European Conference on Computer Vision*, Springer-Verlag, 237–252.

CURTIS, C. J., ANDERSON, S. E., SEIMS, J. E., FLEISCHER, K. W., AND SALESIN, D. H. 1997. Computer-generated watercolor. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 421–430.

FEDKIW, R., STAM, J., AND JENSEN, H. W. 2001. Visual simulation of smoke. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, 15–22.

FISHER, B., PERKINS, S., WALKER, A., AND WOLFART, E., 1994. Hipr - hypermedia image processing reference. http://www.cee.hw.ac.uk/hipr/html/.

GALVIN, B., MCCANE, B., NOVINS, K., MASON, D., AND MILLS, S. 1998. Recovering motion fields: An evaluation of eight optical flow algorithms. In *Proceedings of the British Machine Vision Converence (BMVC)*.

HAEBERLI, P. 1990. Paint by numbers: abstract image representations. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, ACM Press, 207–214.

HERTZMANN, A. 1998. Painterly rendering with curved brush strokes of multiple sizes. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM Press, 453–460.

HORN, B. K. P., AND SCHUNCK, B. G. 1981. Determining optical flow. *Artificial Intelligence 17*, 1-3 (Aug.), 185–203.

KALNINS, R. D., DAVIDSON, P. L., MARKOSIAN, L., AND FINKELSTEIN, A. 2003. Coherent stylized silhouettes. 856–861.

KELLER, A. 1997. Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 49–56.

KLEIN, A. W., LI, W., KAZHDAN, M. M., CORRA, W. T., FINKELSTEIN, A., AND FUNKHOUSER, T. A. 2000. Non-photorealistic virtual environments. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 527–534.

LITWINOWICZ, P. 1997. Processing images and video for an impressionist effect. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 407–414.

MEIER, B. J. 1996. Painterly rendering for animation. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM Press, 477–484.

MINNEN, D., 2002. http://www.cc.gatech.edu/ccg/people/david/miniproj.html.

RE: VISION EFFECTS, 2003. Video Gogh 2.7.

SHEWCHUK, J. R. 1996. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry: Towards Geometric Engineering*, M. C. Lin and D. Manocha, Eds., vol. 1148 of *Lecture Notes in Computer Science*. Springer-Verlag, May, 203–222. From the First ACM Workshop on Applied Computational Geometry.

SHEWCHUK, J. R., 2002. Triangle: A two-dimensional quality mesh generator, version 1.4. http://www-2.cs.cmu.edu/ quake/triangle.html.

STAM, J. 1999. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 121–128.

STAM, J. 2003. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*.

SUTHERLAND, I. E. 2003. Sketchpad: A man-machine graphical communication system. Tech. Rep. UCAM-CL-TR-574, University of Cambridge, Computer Laboratory, Sept.

SYNTHETIK SOFTWARE, INC., 2002. Studio Artist 2.0.

TREUILLE, A., MCNAMARA, A., POPOVIC, Z., AND STAM, J. 2003. Keyframe control of smoke simulations. *ACM Transactions on Graphics (TOG) 22*, 3, 716–723.

WITTING, P. 1999. Computational fluid dynamics in a traditional animation environment. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 129–136.

Figure 14: In this rendering the nonfluid regions are not completely covered by brush strokes, allowing for some of the fine details in the underlying image to show through.

Figure 15: In this rendering we have bounded the fluid simulation to the fluid rendered region of the image.