Interactive Painterly Stylization of Images, Videos and 3D Animations

Jingwan Lu [1,2] Pedro V. Sander [1] Adam Finkelstein [2]
[1] Hong Kong UST [2] Princeton University

(a) Image (b) Video (c) 3D Model (d) Hybrid

Figure 1: *Different rendering styles showcasing our real-time system for stylizing (a) an image, (b) a frame of a video, (c) a frame from a rendered 3D animation scene, and (d) a hybrid scene that combines 3D animation of a lizard with a still photograph in the background.*

Abstract

We introduce a real-time system that converts images, video, or 3D animation sequences to artistic renderings in various painterly styles. The algorithm, which is entirely executed on the GPU, can efficiently process 512^2 resolution frames containing 60,000 individual strokes at over 30 fps. In order to exploit the parallel nature of GPUs, our algorithm determines the placement of strokes entirely from local pixel neighborhood information. The strokes are rendered as point sprites with textures. Temporal coherence is achieved by treating the brush strokes as particles and moving them based on optical flow. Our system renders high quality results while allowing the user interactive control over many stylistic parameters such as stroke size, texture and density.

Keywords: Non-photorealistic rendering, painterly rendering, GPU processing, video processing, particle systems

1 Introduction

Artists over hundreds of years have refined a range of painting techniques to convey a scene while injecting their own individual style as a vehicle for abstraction, expressiveness and creativity. The most visually distinctive feature in many painting styles is the clear outline of individual brush marks on the canvas. Researchers working on non-photorealistic rendering (NPR) have introduced a variety of computer graphics techniques for painterly rendering wherein brush strokes emulate many of the effects seen in traditional paintings. The bulk of such research has sought to optimize the brush paths or overall arrangement of the strokes, often at the expense of substantial computation.

This paper presents a system for stylization of images, video, and 3D models. The method supports a broad range of painterly styles based on brush stroke primitives, via a toolbox of parameters that control, for example, stroke size or density. To facilitate the cycle of experimentation and observation, it is crucial to offer the user *interactive* control over such parameters. Moreover, a fully interactive system supports applications where the input data is not known in advance, for example games or streaming video.

In order to achieve interactive frame rates, the algorithms we describe are implemented entirely on the GPU. The challenge is to find algorithms that can exploit the parallel computing power available in this environment. Our key observation is that it is possible to create high-quality painterly renderings by making purely local decisions about the arrangement of strokes. Rather than asking the question "Where should I put the next stroke?" our approach is to ask "Should I place a stroke *here*?" Our strategy for answering this question is based on two components: a rendered buffer that tracks stroke density throughout the image, and stochastic processes for placing new strokes where the density is too low or deleting strokes where density is too high. These purely local processes are suitable for parallelization at the stroke level and can thus be mapped onto the GPU.

This framework easily accommodates *moving* imagery – either video or animated 3D models. In such cases the challenge is to maintain temporal coherence for the strokes, avoiding flickering without falling prey to the "shower door effect" (the illusion that the image is seen through a semi-transparent shower door whose facets are the set of strokes fixed in the image plane). Our approach is to transport the strokes according to optical flow (for video) or exact geometric flow (for 3D models). As a result of stroke advection, their local density changes from frame to frame. Thus strokes are added or deleted to maintain a target density. Except for the choice of optical flow methods, the same simple pipeline works for all three media, even in combination (for example 3D models composited over streaming video).

This paper and the accompanying video demonstrate the algorithm, revealing imagery in a variety of styles and allowing the user to modify parameters and display moving imagery at interactive frame rates (Figure 1). Applications for this work include artistic control in image and video processing applications; painterly rendering for games, virtual worlds or architectural/design tools; and stylistic range in a whimsical variation on video conferencing.

2 Related Work

Stroke-based rendering techniques. Brush strokes are commonly used for simulating various artistic styles. Stroke based approaches, such as [Strassmann 1986; Hertzmann 1998; Kalnins et al. 2002; Park and Yoon 2008] model the brush strokes as spline curves. One advantage is that they can model long, continuous brush strokes with varying size and shape. Model-based approaches such as [Meier 1996; Kaplan et al. 2000; Haller and Sperl 2004; Luft and Deussen 2006] use particles in 3D to model the brush strokes. Usually, they associate 3D particles with the geometry of the model, derive the stroke properties from surface attributes, and render the particles as brush strokes in screen space. These approaches are not appropriate for the image and video domain. First, modeling spline curves on the GPU introduces added complexity with a substantial performance impact. Second, model-based approaches make use of 3D geometry properties that are not available in 2D image domain. In our work, we represent brush strokes as particles in the image and time domain and determine the particle properties based solely on image processing. Thus, our approach has the flexibility to handle image, video, and 3D geometry.

Previous researchers have introduced several stroke-based rendering algorithms for image processing. For example, Shiraishi and Yamaguchi [2000] aimed to create an automatic painterly rendering system with minimal user intervention. They estimate the stroke properties based by approximating the local regions of the source image with rectangular brushes. Gooch et al. [2002] presented a method to use the approximate medial axes of the segmented features of the image to guide the creation of brush strokes. Kovács and Szirányi [2004] proposed a fully automatic rendering method that targets at removing randomness and providing a more natural look by using the image features to guide all of the parameters. All these methods rely on the heavy use of local image features and require expensive computation. Our algorithm makes the decision to place strokes independently at each location, which simplifies the rendering process and still produces high quality rendering results.

Closest to our work in stroke-based rendering are the methods of Hertzmann and Perlin [2000], and Vanderhaeghe et al. [2007]. Hertzmann and Perlin introduce a painterly video rendering system that successively paints over earlier time frames at interactive rates. Their system can also optionally use optical flow to better track scene changes, but it significantly impacts rendering time. Our approach provides a fast real-time fully parallel algorithm pipeline that runs entirely on the GPU and also efficiently handles geometry animation as input through the use of reprojection. The method of Vanderhaeghe et al. can also handle geometry input. Moreover, it achieves high quality results via a temporally coherent blue noise sampling distribution for stroke placement, but at the expense of real-time frame rates. Overall, our system provides a simpler and faster parallel solution when compared to these methods, while still being very generic in handling input composed of video, animated geometry, or both. Furthermore, the user interface provided by our system allows for very intuitive control of different rendering styles.

Temporal coherence in video rendering. Video processing is different from image processing and is usually composed of two sub-problems: (1) rendering individual frames in a specific style, and (2) maintaining temporal coherence across successive frames. Single-image algorithms often cannot be applied directly to individual video frames without inducing poor temporal coherence (usually flickering) into the resulting animation. To address this problem, one solution is to translate strokes from frame to frame using an estimated optical flow vector field [Litwinowicz 1997]. The source video sequence can also be processed as a spatio-temporal voxel volume [Collomosse et al. 2005]. Wang et al. [2004] developed an anisotropic kernel mean shift technique to segment the video data into contiguous volumes. Bousseau et al. [2007] presented a method that employs texture advection along lines of optical flow. Horn and Rhunck [1981] introduced "The Smoothness Constraint" to solve the optical flow vector field. For its simplicity, we adopt the GPU implementation of Warden [2005].

Artistic styles in video rendering. Various styles have been realized in video processing results such as painterly [Litwinowicz 1997; Hays and Essa 2004; Park and Yoon 2008], watercolor [Bousseau et al. 2007], cartoon [Wang et al. 2004; Winnemöller et al. 2006] and abstract [Klein et al. 2002] styles. Litwinowicz [1997] described a technique that transforms ordinary video segments into animations having an impressionist effect. Extending from this work, Hays and Essa [2004] presented a stroke-based painterly video rendering algorithm that constrains the change of stroke properties to guarantee temporal coherence. Park and Yoon [2008] also addressed the painterly rendering of video sequences with focus in using motion maps to maintain temporal coherence. These algorithms produce high quality video rendering results. However, as CPU-based offline algorithms, they are not suitable for real-time applications. Inspired by previous work, we design a GPU-based real-time algorithm that produces rendering results in various styles. We borrow Hays' idea for rendering brush strokes using stroke textures as height maps for per-pixel lighting.

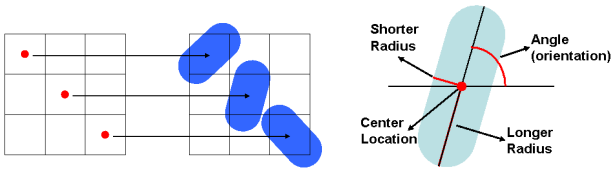
Real-time video stylization. Real-time algorithms have been proposed to stylize video sequences. Klein et al. [2002] introduced an approach that treats the video as a space-time volume of image data (a "video cube"), and extended rendering techniques for video far beyond impressionism to more abstract styles (with interactive controls). Winnemöller et al. [2006] proposed an automatic abstraction framework for images or video. They first reduce contrast in low-contrast regions while enhancing contrast in higher contrast regions and then stylize the imagery using soft color quantization. Hong et al. [2008] later proposed an extension to further improve the processing efficiency. These approaches are GPU-based, but they do not model individual brush strokes, limiting the range of possible styles. Our system operates in real time without precomputation, and supports a broad range of painterly styles.

3 Overview

In this section we provide a detailed overview of our algorithm. We introduce a set of data structures and algorithms that allow all the steps of the stylization process to take advantage of the parallelism provided by the GPU. The entire stroke generation, manipulation and deletion process uses local operations within GPU programmable shaders. In particular, we make heavy use of geometry shaders and their streaming functionality to generate and update the strokes.

3.1 Stroke representation

Stroke texture. Initial stroke properties are stored in a multi-channel 2D texture M. Figure 2(a) demonstrates the relationship between M and the strokes. Each stroke initially corresponds to one texel in the texture. The location of the texel within the texture corresponds to the initial physical location of the center of the stroke within the image. Each texel stores the properties of at most one stroke. This restriction does not impose a practical limitation on the number of strokes since in typical NPR applications the number of pixels far exceeds the number of strokes. Thus, in practice, most of the texels are not associated with any strokes. Figure 2(b) shows the stroke properties that we calculate and maintain in texture M.



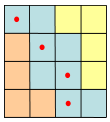
(a) stroke texture (b) stroke properties

Figure 2: *Stroke representation: (a) a 3x3 region of a stroke texture in which three texels (shown with red dots) correspond to strokes, and (b) the stroke properties stored at each texel.*

Layers. For traditional paintings, artists typically begin by painting a coarse representation of the scene using broad brush strokes. Next they paint successive layers of smaller strokes that refine the painting by adding detail, especially in high contrast areas like object silhouettes. Adding fine detail to a region generally draws the viewer's gaze, so it can also be leveraged as a compositional tool.

In order to model this layering process, we use the magnitude of the image gradient to classify each stroke into one of L different layers. In our experiments we have found $L = 3$ to generally provide sufficient expressiveness, and further layers simply add unnecessary computational cost. Figure 3 shows an input source image and the rendering of its three layers. Strokes in different layers have different properties. In the areas where the gradient magnitudes are large, the strokes are smaller, denser and more opaque.

Stochastic stroke placement. In order to allow for completely *localized* stroke processing, we introduce a stroke placement algorithm that determines whether to place a stroke at a given texel position solely based on the result of a stochastic process performed at that texel. For example, if a region ideally should have one large stroke for every ten texels, the probability that the pixel shader generates a stroke at each of those texels is set to 0.1. For pseudo-random number generation, we simply used a texture with random entries and computed the per-pixel texture coordinates as a function of both the screen space coordinates and time.



The framework offers considerable control for determining the coarseness of the strokes in the painting. For $L = 3$ layers of strokes, three different desired probabilities p_c, p_m, p_f for the coarse, medium, and fine layers can be specified in order to indicate the likelihood for a particular type of stroke to be present in that location. The inset figure shows a small example identifying strokes appearing in the medium layer (medium gradient magnitudes shown in cyan, where pixels from the other two layers are denoted orange and yellow). Here the red dots indicate center locations of brush strokes corresponding to half of the pixels in the medium layer, corresponding to $p_m = 0.5$. Section 5 describes in detail the stochastic processes for stroke generation and deletion to achieve this target stroke density.

Note that while the choices for p_c, p_m, and p_f are entirely style-specific, usually, one would want the small number of pixels on edges (higher gradient magnitudes) to be more likely to own strokes in order to more accurately delineate the object boundaries. Therefore, in practice we have found that we usually achieve best results with $0.05 < p_c < 0.3$, $0.4 < p_m < 0.8$, and $0.7 < p_f < 1.0$. However, one should not strictly follow this guideline for all rendering styles. Figure 10 shows examples of different renderings along with the parameters used to create them.

(a) Input photo (b) Painterly result

(c) Coarse layer (d) Medium layer (e) Fine layer

Figure 3: *Layering: given the input photo (a) the output (b) is composited starting from a base layer of coarse strokes (c) up to a top layer (e) containing fine strokes.*

Stroke buffer. In order to efficiently manage the generation and modification of brush strokes, we use a particle system to model the evolution of brush strokes over time. Because of their ability to compact and expand data streams, geometry shaders are suitable for handling the stroke generation and modification operations. The process takes L streams of vertices (strokes), one for each layer. At each frame, the geometry shaders update, generate, and delete strokes from the three buffers independently, based on desired stroke densities p_c, p_m and p_f. Finally, the strokes are rendered as point sprites to the screen, layered from coarse to fine. Refer to Section 7 for rendering results using different painting styles (color and alpha masks).

3.2 Stroke placement algorithm

The proposed algorithm is flexible enough to handle three types of input media: images, videos, and geometry. Next we describe the basic processing pipeline for each type of data. In all cases, the processing is decomposed into three major conceptual steps: image processing, stroke processing, and rendering (Figure 4).

Images. The process of stylizing an input image is as follows (Figure 4-top):

1. *Image processing:* We compute the image gradient at all texels of the low-pass filtered input image (Section 4.1).

2. *Stroke processing:* The stochastic stroke generation process is performed at each texel of the image to generate new strokes and output stroke properties to M. A second rendering pass streams out a vertex buffer of strokes using a geometry shader (Section 5.2).

3. *Rendering:* The geometry shader reads the stroke information from the stroke buffer, generates point sprites, and rasterizes to the frame buffer.

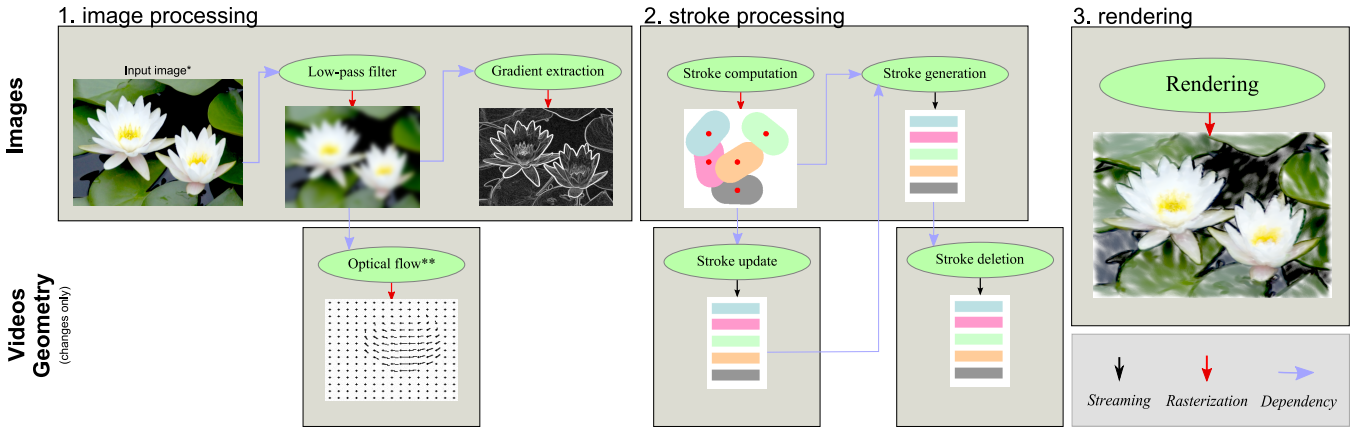


Figure 4: *Main conceptual steps of the algorithm. Note that in the case of geometry, the input image (*) consists of the rendered frame, and the optical flow (**) is computed through forward reprojection.*

Videos. In order to stylize videos, the main challenge is to maintain coherence between consecutive frames. Therefore, in addition to creating strokes, our algorithm also advects existing strokes and deletes strokes that represent regions that are no longer visible or in regions that present excessive stroke overlap. The modifications are as follows (Figure 4-bottom):

1. *Image processing:* The gradient computation is performed for three frames (the current and the two previous frames) and the result is averaged in order to reduce the effect of noise and achieve a gradual transition between strokes in consecutive frames. Additionally, optical flow is computed in order to properly advect the strokes (Section 4.2).
2. *Stroke processing:* Strokes are advected based on the optical flow (Section 5.1), added in empty regions (Section 5.2), and deleted if they no longer represent its underlying position in the input video (Section 5.3).
3. *Rendering:* No changes.

Geometry. In addition to image and video data, the system can also stylize synthetic animated scenes. This is accomplished with only two modifications in the image processing stage of the video pipeline. First, the scene is rendered in order to generate the input image. Second, since we have the underlying geometry, we compute the *exact* motion vectors through forward reprojection (Section 4.3) instead of relying on image-based optical flow.

Note that since the remainder of the pipeline is unchanged, and the rendered image and motion vectors computed from the geometry have the same format as the input image and optical flow computed from videos, the system allows us to use hybrid combinations of images, videos, and geometry (see results in Section 7).

4 Image Processing

Here we describe in detail the image processing steps outlined above; these steps produce the data needed to manage the strokes.

4.1 Gradient Extraction

Artists commonly apply brush strokes on the canvas according to specific rules. Generally, the strokes follow the boundaries between different shading levels in the image in order to effectively convey the shape of the object. Therefore, the most natural orientation for a single stroke at a particular position should be perpendicular to the intensity gradient direction. Specifically, we smooth the image using a 3x3 box filter and then apply the Sobel filter to calculate the gradient magnitude and gradient direction for each pixel. Each step is performed by rendering a full screen quadrilateral with the appropriate shader, and the results are stored in texture G.

In the coarse-layer pixels, the gradient magnitudes are not well-defined or the gradient directions vary significantly and therefore are not a good indication of stroke orientation. In such smooth areas, we simply use the color hue of the image to determine the direction. As a result, coarse strokes that have similar colors all point in similar directions.

4.2 Video: Optical Flow

In order to determine the motion of pixels over time and achieve better temporal coherence, we rely on image-based optical flow. We used the efficient, simplified GPU implementation of a generic optical flow method [Horn and Schunck 1981] due to Warden [2005]. Generally, the algorithm works well for maintaining temporal coherence for video NPR, which does not require absolutely accurate optical flow analysis. Still, temporal coherence is not assured and some flickering remains, particularly in the presence of fast motion. Improving optical flow quality while still maintaining the high framerate remains an area of future work.

4.3 3D Models: Geometry Reprojection

For synthetic 3D models, we have the luxury of knowing the precise 3D coordinates of each surface point on the screen. Thus, instead of relying on an image-based optical flow algorithm, we can accurately compute the motion at each pixel of the screen by tracking the motion of its corresponding surface point. As a result, we can correctly advect each stroke given the motion at its center.

Reprojection. Let the position of a pixel p in frame $f-1$ be denoted by P_{f-1}. We seek a motion flow texture that contains, for each pixel p in $f-1$, the screen-space position of p in frame f, which is given by P_f.

We adapt the reverse reprojection algorithm of Nehab et al. [2007], which operates entirely on the GPU. At a given frame f, Nehab et al. determine the position P_{f-1} of each current pixel p in frame $f-1$. The key idea is to compute, for each vertex v, the

homogeneous projection space coordinates V_{f-1} and V_f for *both* frame $f-1$ and frame f. V_f is computed typically by applying the traditional model-view-projection transformation matrix M_t. V_{f-1} is calculated by using the transformation from previous time frame M_{t-1}. The output vertex position is set to V_f, while V_{f-1} becomes a vertex output attribute that is relayed to the pixel shader through hardware interpolation. The pixel shader then dehomogenizes the interpolated position to retrieve the 3D projection space coordinates P_{f-1} in the earlier frame $f-1$. See Nehab et al. for further details.

In order to compute *forward* motion, we make minor modifications to the above. We first set the output vertex position to V_{f-1}, since we want the forward motion vectors in the screen-space of the previous frame $f-1$, rather than the reverse ones in frame f. We then pass V_f as the attribute to the pixel shader. The following frame's position of this pixel is then given by dehomogenizing the interpolated attribute at each pixel, yielding in P_f. We can now advect each stroke from $f-1$ to f based on the destination screen-space position stored at its center pixel in $f-1$.

The approach above is general enough to allow any form of vertex shader processing, including camera motions and skinned or procedural animations. The only drawbacks are that it requires more constant-store memory for the previous frame's transformation and skinning matrices, and doubles the amount of vertex processing. However, for our application, this overhead is negligible as the bulk of our work lies in the stroke generation and update.

Handling occlusion. So far, our approach does not account for occluded regions. It is possible that a stroke in frame $f-1$ may become occluded in frame f and therefore should be deleted. To account for this, we need to store the post-projection z coordinate for each stroke. When we advect the stroke x and y coordinates through reprojection, we also advect their z coordinates following the exact same procedure. We can then compare this properly advected z value with the z value stored in the current Z-buffer of frame f at the target pixel. Since the post-projection z coordinate was advected based on the transformation matrices, its value should match regardless of the motion undergone at that surface point. The exception is when the surface point is no longer visible because it is occluded by another surface point. In such a case, there is a z mismatch and the stroke is deleted.

We can naturally blend foreground objects with background images and videos in this setting by simply setting the z value of image and video strokes to be equal to the far-plane z. Thus, proper occlusion between the animated model and background is ensured.

5 Stroke Processing

The image processing step generates 2D textures G containing the gradient and V containing optical flow. These textures serve as input to the stroke processing pipeline.

5.1 Stroke Update

The existing strokes used to depict the shape of the objects in one frame often need to change their position, orientation and size in order to match the position and shape of the objects in the next frame. This update is performed entirely by the geometry shader, which streams in the strokes and streams out a buffer with the updated strokes. For a particular stroke, the geometry shader first uses the optical flow vector from V to advect its position then updates the stroke based on color and local density. To reduce temporal artifacts we change the stroke properties only gradually using the following rules:

- **Size and orientation.** A stroke is generated in a specific layer and with a fixed size, and these remain unchanged throughout their lifetime. The orientation is only allowed to change gradually (no more than 1 degree per iteration), except at the fine layer. We allow the fine layer to change quickly, even though this may introduce flickering, because this layer should conform to fast moving boundaries.
- **Color.** Advected strokes take the color of the target location.
- **Opacity.** When a stroke drifts away from their initial layer, it is faded out. Additionally, newly created strokes are faded in gradually in this step.

5.2 Stroke Generation

When rendering video and animated geometry, the stroke advection may expose the canvas. Furthermore, fine-layer brush strokes may drift away from fine areas and get deleted during the stroke update step. For these reasons, we generate new strokes to fill in the blanks and add additional strokes to refine the image features. This step is used to generate all the strokes when rendering a single image, the first frame of a video or a scene where initially there are no strokes.

We consider adding a stroke at each texel and do so using the stochastic process described in Section 3. First we compute properties of potential strokes and output to M using a full screen quadrilateral. On a second pass, we render any existing strokes into an offscreen buffer D using additive blending, in order to measure stroke density everywhere in the image. A third pass determines where to add new strokes, as follows. A buffer that contains one vertex for each canvas pixel is processed by the geometry shader. The shader writes a subset of these vertices as strokes into the output buffer, depending on whether the stochastic process generates a stroke at that position. It uses the position of each input vertex as texture coordinates in D and M. If the value in D is below a threshold, and depending on the result of the stochastic stroke placement algorithm described in Section 3, a geometry shader appends a stroke to the appropriate layer at the current location. Otherwise, the geometry shader does not produce any output. Our system uses a different threshold function for D at the fine layer that tends to generate more strokes, because we find it is more important to track high-gradient edges in moving imagery. Figure 5 shows the generation of strokes from the initial vertex buffer and the updated texture M. It appends additional strokes into the respective buffer according to the texture M and the result of a random process.

The color of the generated stroke is determined by the color of the corresponding pixel in the processed source frame. In order to prevent video noise from rapidly changing a brush stroke color, the color of each stroke is also averaged over three frames. The size, orientation and opacity of the stroke are simply retrieved from the previously prepared textures, M and G. The stroke also has a "status" property, which can be one of the three values "normal", "new" or "old". When a stroke is generated for the first frame of a video or for a single image, the status assumes "normal". When a

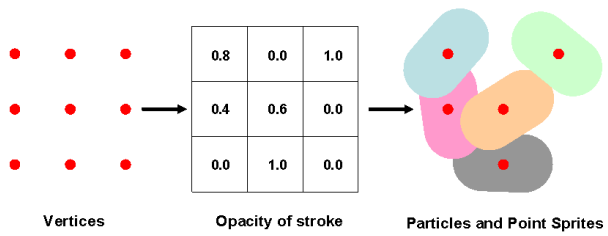


Figure 5: *Stroke generation*

stroke is appended to the existing list of strokes for filling blank or refining features, the status assumes "new" and the stroke is gradually faded in during the update step.

5.3 Stroke Deletion

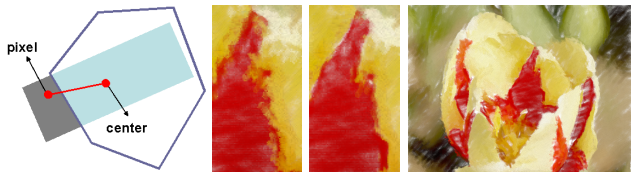
Due to the stochastic nature of the algorithm, multiple strokes overlap in many locations of the rendered image. The optical flow computation updates the stroke positions, which may aggravate the overlap in some areas. Furthermore, adding new strokes at every frame to fill blank regions and refine details may cause the number of brush strokes to increase significantly. To address this problem, we add a subsequent step to delete heavily overlapped brush strokes. We simply render D as in Section 5.2, and then stochastically delete strokes from the geometry stream wherever D exceeds a threshold.

6 Stroke Rendering

Generation. Strokes are rendered using rectangular point sprites. Given the stream of stroke particles (vertices) that represent the brush strokes and their properties, we use the geometry shader to generate two-triangle textured sprites for each stroke.

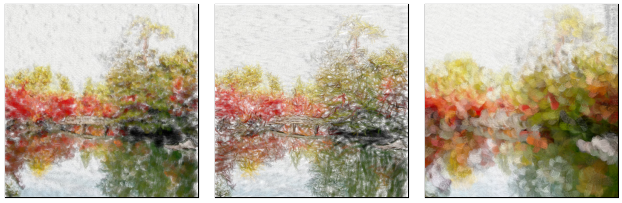
Rendering. To give richer detail to the surface of the brush strokes, a brush texture and alpha mask are applied to the sprites in the pixel shader. For each pixel of the sprite, we construct a simple lighting model from the stroke texture based on [Hays and Essa 2004]. A height map (x, y, z) is constructed from the stroke texture by taking the texture coordinates as (x, y) and the intensity as the height z. A per-pixel normal N is then computed from the height map, and finally the lighting calculation can be performed using a fixed light direction L and the normal N. The final color is the product of the lighting factor $(N \cdot L)$ with the stroke color. The final alpha value is given by the product of the stroke opacity and the mask. Finally, all the transparent brush strokes are alpha composited on the canvas with high frequency strokes rendered over low frequency ones.

Clipping. Since the distribution of the strokes is determined by a stochastic process and the shape of the strokes solely relies on the alpha mask texture, it is possible that large strokes near the object boundaries may "leak" out of the boundaries. We include an additional optional step to reduce this stroke leaking problem efficiently. In the pixel shader of the rendering pass, we linearly scan the pixels from the current location towards the center of the stroke. If we encounter any object boundaries in this process, the current pixel is disabled for rendering. The effect of this step is illustrated in Figure 6(a). While this optional step cannot guarantee preservation of the shape of the stroke, in practice is does not produce objectionable artifacts and avoids the leaking artifacts that can be produced by our efficient greedy algorithm.



(a) stroke leaking (b) before (c) after (d) after: full image

Figure 6: *Sprite clipping reduces leaking. The gray region of the stroke in (a) is clipped to alleviate stroke leaking (b). After clipping the strokes better respect high frequency edges shown in (c) as a closeup of the full image (d).*



(a) small (b) long (c) large

Figure 7: *How stroke size impacts style.*

7 Results

We offer several examples of stylized images generated with our system. Figure 1b-d shows results based on video and animation. The accompanying video demonstrates these examples in motion, as well as showing some of the effects of interactive user control. Figure 7 shows the impact of stroke size on the rendering style of a given image, while Figure 9 shows the effect of texture choice. Figure 10 provides examples of how other stroke parameters affect the rendering style. Next we review the user controls that are available and were employed to produce such results. We then discuss the overall performance.

7.1 Interaction

Here we summarize the stylistic controls available to the user. Of course, the most fundamental decision is the scene content – still image, video, 3D model, or a combination. Having chosen a scene, the user can interactively adjust a variety of parameters. We briefly review previously described parameters here and also present several variations.

Layering. Figure 3 shows how strokes are divided into coarse, medium and fine layers. The separation occurs at two user-controlled gradient thresholds ("Gradient Threshold" in Figure 10). Three other parameters adjust the target stroke density in each layer ("Probability" in Figure 10).

Strokes. Stroke placement and orientation emerges from an automatic process. However, the user may adjust parameters like stroke length, width, alpha and brush textures (Figures 7, 9 and 10). The smooth brush used in Figure 9d results in an "impressionist" style image. In contrast, the brush texture in Figure 9f contains a dotted pattern, resulting in a more "pointillist" style. In general, larger and coarser strokes produce more abstract styles, while smaller and denser strokes give more realistic styles. In Figure 10, the "Relative Size" parameters describe the ratios of length and width in the medium and fine layers, relative to those of the coarse layer.

Stroke lifetime. For moving imagery like video and 3D models, strokes die due to crowding from optical flow or a change in the parameters above. As strokes die they are faded out to avoid flickering, and the duration of the fade can be used to artistic effect. An extreme example is shown to the right where strokes from previous frames of a 3D animated walking man are faded slowly and provide a kind of receding "halo." This effect is reminiscent of various artistic representations of motion, such as Marcel Duchamp's *Nude Descending a Staircase* (No. 2, 1913), or the "speedlines" of Masuch et al. [1999].

(a) Emphasis on left (b) Emphasis on right

Figure 8: *Placing emphasis via controlled stroke density.*

Emphasis. Artists know that by placing more fine detail in an area of an illustration it can draw the viewer's gaze to that area [Guptill 1976] and this effect has been leveraged in various NPR systems, such as that of DeCarlo and Santella [2002]. Our system provides two controls for placing emphasis this way. In the case of hybrid 2D-3D imagery, the user can specify that the gradients in the background be attenuated by a constant factor $0 < f < 1$, which effectively reduces the role of the background in the fine layer and thereby draws attention to the foreground. Alternately, the user can use the mouse to interactively specify the location c and radius r of a central point of interest. Gradients throughout the image are then attenuated by a smooth-step falloff towards f as the distance from c approaches r (Figure 8).

7.2 Performance

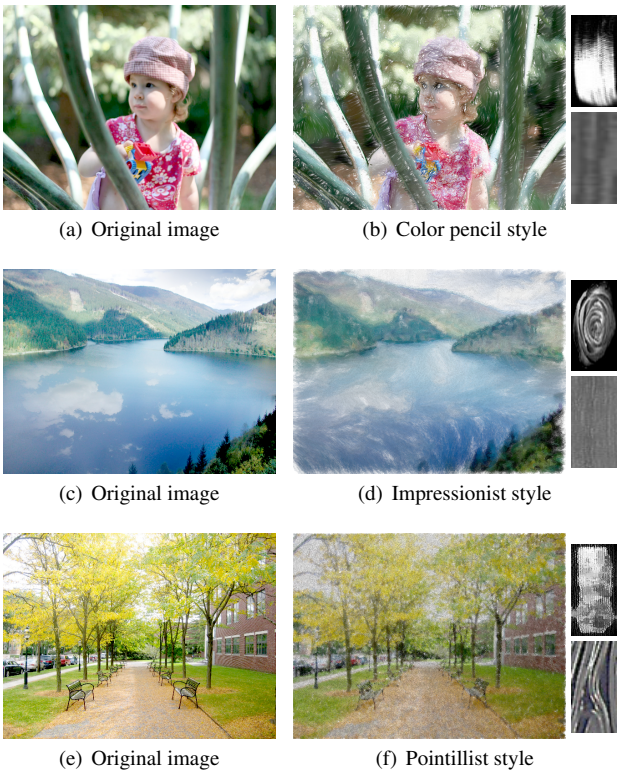
Our system is suitable for real-time applications such as streaming video (e.g., Figure 1(b)) and games (e.g., Figure 1(c)). Please refer to the accompanying video for full examples of stylized videos and animated geometry.

We implemented our system using Microsoft DirectX 10 and benchmarked on an Intel Core2 CPU at 2.39 GHz and 4GB RAM with an NVIDIA Geforce 8800 GTS. All results herein were fully processed and rendered in real-time, allowing immediate user feedback. Frame times ranged from 30 to 50, number of strokes from 10,000 to 60,000, and resolution at 512x512 pixels. The number of strokes, the size of each stroke and the number of pixels in the canvas, all affect the running time of the algorithm linearly, with any of them possibly dominating the cost depending on the parameters set by the user.

8 Conclusion and Future Work

Our GPU-based stroke rendering system provides high quality stylized image, video and 3D animation results in various artistic styles at interactive rates. We design a parallel multi-layer stroke-based stochastic approach to guide the generation and deletion of brush strokes. The movement of brush strokes and the change of stroke properties are modeled based on optical flow and geometry reprojection entirely on the GPU. We apply stroke texture and alpha masks to determine the stroke surface detail and stroke shape which, in turn, determine the feeling and style of the final rendering.

Limitations. The temporal quality of our video processing depends on the effectiveness of the optical flow implementation; we find that failures in optical flow computation lead to distracting artifacts. For example if the optical flow result is very noisy it leads to flickering, or if the result is falsely reported as zero it induces the "shower door effect." Moreover, even when flow can be computed exactly, as with geometric input, there remains a tradeoff between faithful reproduction of input image features, and maintaining temporal smoothness. Nevertheless, our system offers the user some control with regard to this tradeoff. Finally, the stochastic process we describe offers no guarantees about stroke density, and thus strokes can cluster unnecessarily or leave holes in the canvas.



(a) Original image (b) Color pencil style

(c) Original image (d) Impressionist style

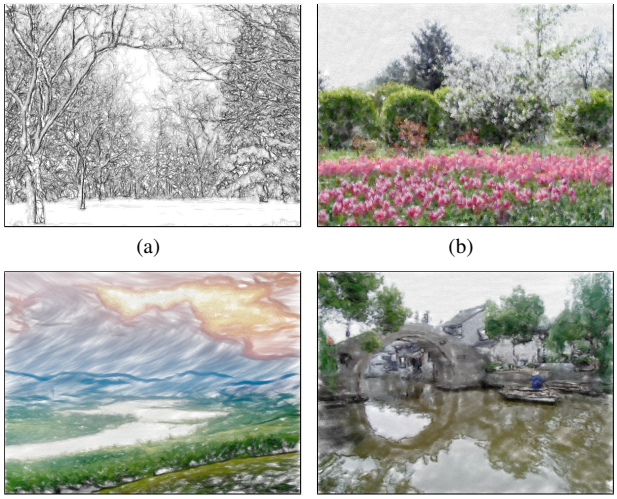
(e) Original image (f) Pointillist style

Figure 9: *Variety of styles based on the brush textures (right).*

Future work. We would like to further investigate the optical flow step in order to further improve the quality of the video sequences, especially under fast motion. This kind of technology could be applied to **video conferencing**, where advantages might include high compression (just send the stroke information), a layer of privacy (fewer concerns about makeup, hair, clothing, etc), and a fun channel for expressiveness. Moreover, a variation in this space would include **virtual meetings** in which parties interact with 3D objects in 3D scenes, all of which can be rendered with consistent style. For such applications it would be ideal to use face tracking methods and specialize the painterly algorithms for rendering faces. Finally, we would like to offer **explicit user control** over the "gradient" field or the "optical flow" field as ways of orchestrating the stroke directions or movement. This would allow for composing a picture like van Gogh's *Starry Night* (1889) or even a version in which the strokes swirl over time.

Acknowledgements

We thank the I3D reviewers for their suggestions for improving the paper. Many of our images are based on pictures by Jianming Lu. We thank Anthony Santella for the photo on which Figure 8 is based, as well as Flickr users Jackie Hodges (Figure 1a) and Randen Pederson (Figure 10a), who released their photos under the Creative Commons. This work was supported in part by Adobe Systems, the NSF grant IIS-0511965, and Hong Kong RGC grant 619207.



(a) (b) (c) (d)

Parameter:	(a)	(b)	(c)	(d)
Stroke Length [pixels]	6.69	5.55	10.67	7.82
Stroke Width [pixels]	1.00	5.47	3.24	3.05
Relative Size (Medium)	0.44	0.47	0.46	0.53
Relative Size (Fine)	0.38	0.23	0.29	0.20
Gradient Threshold 1	0.05	0.09	0.09	0.10
Gradient Threshold 2	0.37	0.36	0.27	0.30
Probability (Coarse)	0.00	0.05	0.06	0.08
Probability (Medium)	0.18	0.18	0.47	0.41
Probability (Fine)	0.78	0.78	0.61	0.79
Alpha (Coarse)	0.01	0.58	0.35	0.36
Alpha (Medium)	0.87	0.81	0.75	0.61
Alpha (Fine)	0.91	0.92	0.87	0.86

Figure 10: *Example stylized images produced using our method, together with many of the parameters used to create them. (See Section 7.1 for an overview.)*

References

BOUSSEAU, A., NEYRET, F., THOLLOT, J., AND SALESIN, D. 2007. Video watercolorization using bidirectional texture advection. In *Proceedings of SIGGRAPH 2007*, 104.

COLLOMOSSE, J. P., ROWNTREE, D., AND HALL, P. M. 2005. Stroke surfaces: Temporally coherent artistic animations from video. *IEEE Transactions on Visualization and Computer Graphics 11*, 5, 540–549.

DECARLO, D., AND SANTELLA, A. 2002. Stylization and abstraction of photographs. *ACM Transactions on Graphics 21*, 3 (July), 769–776.

GOOCH, B., COOMBE, G., AND SHIRLEY, P. 2002. Artistic vision: painterly rendering using computer vision techniques. In *Proceedings of NPAR 2002*, 83–90.

GUPTILL, A. L. 1976. *Rendering in Pen and Ink*. Watson-Guptill Publications, New York.

HALLER, M., AND SPERL, D. 2004. Real-time painterly rendering for mr applications. In *GRAPHITE 2004*, 30–38.

HAYS, J., AND ESSA, I. 2004. Image and video based painterly animation. In *Proceedings of NPAR 2004*, 113–120.

HERTZMANN, A., AND PERLIN, K. 2000. Painterly rendering for video and interaction. In *Proceedings of NPAR 2000*, 7–12.

HERTZMANN, A. 1998. Painterly rendering with curved brush strokes of multiple sizes. In *Proceedings of SIGGRAPH 1998*, ACM, 453–460.

HONG, C., YANG, Z., BU, J., LIU, Y., AND CHEN, C. 2008. Cartoon-like stylization of video for real-time applications. *International Conference on Multimedia & Expo*, 958–988.

HORN, B., AND SCHUNCK, B. 1981. Determining optical flow. *Artificial Intelligence 17*, 185–203.

KALNINS, R. D., MARKOSIAN, L., MEIER, B. J., KOWALSKI, M. A., LEE, J. C., DAVIDSON, P. L., WEBB, M., HUGHES, J. F., AND FINKELSTEIN, A. 2002. WYSIWYG NPR: drawing strokes directly on 3D models. In *Proceedings of SIGGRAPH 2002*, 755–762.

KAPLAN, M., GOOCH, B., AND COHEN, E. 2000. Interactive artistic rendering. In *Proceedings of NPAR 2000*, 67–74.

KLEIN, A. W., SLOAN, P.-P. J., FINKELSTEIN, A., AND COHEN, M. F. 2002. Stylized video cubes. In *SCA '02: Proceedings of the 2002 symposium on Computer animation*, 15–22.

KOVÁCS, L., AND SZIRÁNYI, T. 2004. Painterly rendering controlled by multiscale image features. In *SCCG '04: Proceedings of the 20th spring conference on Computer graphics*, 177–184.

LITWINOWICZ, P. 1997. Processing images and video for an impressionist effect. In *Proceedings of SIGGRAPH 1997*, 407–414.

LUFT, T., AND DEUSSEN, O. 2006. Real-time watercolor illustrations of plants using a blurred depth test. In *Proceedings of NPAR 2006*, 11–20.

MASUCH, M., SCHLECHTWEG, S., AND SCHULZ, R. 1999. Speedlines: depicting motion in motionless pictures. In *ACM SIGGRAPH 99 Conference abstracts and applications*, 277.

MEIER, B. J. 1996. Painterly rendering for animation. In *Proceedings of SIGGRAPH 1996*, 477–484.

NEHAB, D., SANDER, P. V., LAWRENCE, J., TATARCHUK, N., AND ISIDORO, J. R. 2007. Accelerating real-time shading with reverse reprojection caching. In *GH '07: Proceedings of Symposium on Graphics Hardware*, 25–35.

PARK, Y., AND YOON, K. 2008. Painterly animation using motion maps. *Graphical Models 70*, 1-2, 1–15.

SHIRAISHI, M., AND YAMAGUCHI, Y. 2000. An algorithm for automatic painterly rendering based on local source image approximation. In *Proceedings of NPAR 2000*, 53–58.

STRASSMANN, S. 1986. Hairy brushes. *Proceedings of SIGGRAPH 1986 20*, 4, 225–232.

VANDERHAEGHE, D., BARLA, P., THOLLOT, J., AND SILLION, F. 2007. Dynamic point distribution for stroke-based rendering. In *Rendering Techniques 2007 (Proceedings of the Eurographics Symposium on Rendering)*, 139–146.

WANG, J., XU, Y., SHUM, H.-Y., AND COHEN, M. F. 2004. Video tooning. In *Proceedings of SIGGRAPH 2004*, 574–583.

WARDEN, P., 2005. Gpu optical flow. http://www.petewarden.com/notes/archives/2005/05/gpu_optical_flo.html.

WINNEMÖLLER, H., OLSEN, S. C., AND GOOCH, B. 2006. Real-time video abstraction. *ACM Trans. Graph. 25*, 3, 1221–1226.