Painting with Polygons: A Procedural Watercolor Engine

Stephen DiVerdi, *Member*, *IEEE*, Aravind Krishnaswamy, *Member*, *IEEE*, Radomír Měch, *Member*, *IEEE*, and Daichi Ito

Abstract—Existing natural media painting simulations have produced high-quality results, but have required powerful compute hardware and have been limited to screen resolutions. Digital artists would like to be able to use watercolor-like painting tools, but at print resolutions and on lower end hardware such as laptops or even slates. We present a procedural algorithm for generating watercolor-like dynamic paint behaviors in a lightweight manner. Our goal is not to exactly duplicate watercolor painting, but to create a range of dynamic behaviors that allow users to achieve a similar style of process and result, while at the same time having a unique character of its own. Our stroke representation is vector based, allowing for rendering at arbitrary resolutions, and our procedural pigment advection algorithm is fast enough to support painting on slate devices. We demonstrate our technique in a commercially available slate application used by professional artists. Finally, we present a detailed analysis of the different vector-rendering technologies available.

Index Terms—Natural media, watercolor painting, vector graphics, real time

1 Introduction

Despite recent advances in digital painting, there remains a large contingent of traditional media artists. Natural media such as watercolors, oil paints, bristle brushes, and charcoals remain relevant to modern artists because of their expressive and serendipitous workflows. The wide range of complex textures and qualities that can be created using, e.g., watercolor paints, is difficult to match in today's digital media.

Natural media simulation, therefore, is an active area of research in the computer graphics community. Toward this end, interactive systems have been developed [1], [2] to allow artists to create compositions more in the style of their traditional tools. Unfortunately, to date none of these research prototypes have been able to make their way into commercial systems. Either their compute requirements have been too high, or they have been unable to support the document resolutions necessary for professional work.

Rather than attempt yet another exact replication of watercolor painting behaviors, we pursue another direction. Our goal instead is to create a painting application with a wide range of dynamic behaviors that are inspired by core watercolor effects such as color blending, feathering, and edge darkening. With a similar fundamental style, our painting application can achieve the same wide variety of complex and serendipitous results as watercolor, but with a unique character that is specific to our algorithm.

We present a novel formulation of an interactive watercolor-like paint algorithm that departs from existing research in many significant ways. First, our formulation uses a particle-based model of pigment flow, rather than grid based. Second, the particle representation is vector instead of raster, allowing for rendering at arbitrary resolutions. Third, the particle update step is a physically inspired procedural algorithm that is very fast to calculate. The result of these differences in approach means that unlike previous work, our algorithm can be used to generate high-resolution output on low-powered devices, while still recreating a useful subset of interactive watercolor paint behaviors. We are able to achieve common watercolor effects such as edge darkening, nonuniform pigment density, granulation, and backruns. Furthermore, by varying model parameters, we can simulate a variety of different types of brush and pigment types.

To demonstrate the performance and real-world utility of our formulation, we shipped a reduced form of the algorithm for the Apple iPad as Adobe Eazel, in conjunction with the release of Adobe Photoshop CS 5.5. Artists are able to paint on their iPads and then transfer the composition to a desktop machine where Photoshop can rerender it at a higher resolution. Due to the novelty of our algorithm, professional artists have been able to create unique paintings with the quality of watercolors that rival their work with other painting tools, but on devices with much less computational power, confirming the success of our approach.

This is an extension of our previous publication [3]. We have added additional results and an in-depth analysis of the polygon rendering options and their performance characteristics.

2 Related Work

The creation of watercolor painting-like images has been extensively studied. Much of the work has focused on the

- *S. DiVerdi is with Adobe Systems, Inc., 428 Alice St., Apt. 522, Oakland, CA 94607. E-mail: steved@adobe.com.*
- *A. Krishnaswamy is with Google, Inc., and Adobe Systems, Inc., 4102 Ashbrook Cir., San Jose, CA 95124. E-mail: ieee@aravind.ca, aravind@adobe.com.*
- *R. Měch and D. Ito are with Adobe Systems, Inc., 345 Park Ave, San Jose, CA 95110. E-mail: {rmech, dito}@adobe.com.*

Manuscript received 24 May 2012; revised 16 Aug. 2012; accepted 17 Sept. 2012; published online 26 Sept. 2012.
Recommended for acceptance by M. Garland and R. Wang.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCGSI-2012-05-0089.
Digital Object Identifier no. 10.1109/TVCG.2012.295.

automatic conversion of photographs into paintings [4], by analyzing the image contents and placing a series of brush strokes. These techniques have been extended to work on videos and in real-time [5], [6]. Bousseau et al. [7] took a different approach, by combining many different image filters applied to photographs or 3D renderings, to create compelling watercolor depictions. While these results are pleasingly artistic, the lack of interactive, artist-driven control limits their utility for painting applications.

Some commercially available applications provide interactive watercolor tools, such as Corel Painter [8] and Ambient Design ArtRage [9]. In Painter's case, the artist paints a brush stroke and then waits for a processing routine to compute the diffusion effect before another stroke can be placed, thus interrupting the painting workflow. ArtRage's watercolor brush allows many brush strokes in rapid succession, but does not compute on-canvas motion of the pigment once it has been deposited. Ultimately, there is not yet a commercially available painting application that captures the interactive and dynamic nature of real watercolor painting.

To create more faithfully realistic reproductions of watercolor paintings, Curtis et al. model the underlying processes of water and pigment advection in their system [10]. They achieved impressive simulations of real watercolor behaviors, but due to the complexity of the algorithm were unable to achieve a fully interactive system. More recently, Chu and Tai [1] and separately, Van Laerhoven and Van Reeth [2] demonstrated similar levels of realism in interactive watercolor painting systems that heavily utilized GPUs to compute their effects in real time. Because of the high compute and resource requirements of these algorithms, they have yet to find their way into commercially shipping products, largely because they have not been demonstrated to be able to handle the resolutions necessary for professional work on commonly available hardware.

In a similar vein, interactive oil painting has been simulated as well, including thick paint application [11] and complex brush paint deposition and dirtying [12]. However, oil paint simulation is fundamentally different from watercolor, in that once deposited on the canvas, oil paints do not tend to flow arbitrarily across the canvas, while this is a main effect of watercolors. Therefore, the tradeoffs made in handling performance, resolution, and quality are fundamentally different in our work.

Because of the complex textures found in watercolor painting, vector output is much less common than raster. Applications like Adobe Illustrator [13] can be used to create watercolor-like vector compositions by using static vector artwork of scanned watercolor brush strokes, but it lacks all of the dynamic aspects of watercolor painting. The Ardeco system [14] turns arbitrary images into art deco-style vector representations consisting of linear gradients, while diffusion curves [15] enable the creation of more complex color gradients, but neither technique is amenable to be controlled by a brush stroke input interface. Finally, the work of Ando and Tsuruno [16] uses an adaptive vector representation to track the front of a pigment fluid mixing in water. However, because of the lack of diffusion support, over time the front's complexity becomes unbounded, limiting the viable duration of the simulation.

3 Algorithm

The intuition behind our model is that pixel grid-based simulations are dense, whereas watercolor stroke effects are generally sparse, and that the specific path pigment particles take through canvas media is not as important as the generally complex dynamic behavior. In fact, each particle in real watercolor paint can be thought of as taking a random walk, and the aggregate behavior is that of watercolor paint. Based on this intuition, we adopt a sparse representation for our paint pigment, and we use a random walk algorithm to update its position each time step. By carefully selecting the model and update equation, we can recreate a variety of interesting behaviors.

At the core of our model, we represent paint pigment as a collection of dynamic "splat" particles, where each splat is a complex polygon of n vertices. Many such splats are placed along the trajectory of a stroke based on rules controlled by the current brush type, and then the vertices of these splats are advected according to another set of rules at each time step. Each splat has its own opacity value, and splats are rendered back to front with standard transparency blending ("over" compositing). With this approach, we are able to reproduce a variety of dynamic watercolor paint behaviors, including edge darkening, granulation, backruns, and different types of brush strokes.

3.1 Paint Initialization

During stroke input, stamps are placed in uniform path length increments along the stroke path, so that slow and fast strokes covering the same distance will result in the same number of stamps placed. Each stamp is a set of one or more splats (initially circles), arranged according to the current brush type. We use the "wet-on-dry" brush as an example, which consists of seven splats of $n = 25$ vertices arranged with one centered on the stamp position and six placed around its perimeter (see Fig. 2). Each splat stores motion vectors including a motion bias vector $\mathbf{b}$, and a per-vertex velocity vector $\mathbf{v}$. Additionally, a splat stores its age a (in steps), as well as parameters of the brush type: roughness r (in pixels) and flow f (percentage). These parameters are similar in effect to those used in MoXi [1], but we use them differently in our algorithm.

At each stamp, an amount of water is also added to the canvas. Water does not move on canvas and is stored separately from the list of splats, as a rasterized "wet map" consisting of a 2D grid of cells. We set the wet map resolution to match that of the display being painted on, so on an iPad 2, it is $1{,}024 \times 768$ cells. The brush type specifies the size and shape of the region that is wetted by rasterizing a constant value into the wet map. This value represents the remaining amount of time each cell will stay wet. If the cell is already wet (i.e., has nonzero value) from an earlier stroke, its value is overwritten.

The user selects a brush stroke size in pixels, which is then used to scale the splats and wet region based on roughly how large the final resulting stroke will be, to achieve the specified size.

3.2 Pigment Advection

At each time step, the set of live splats is iterated across, updating each vertex position $\mathbf{x_t}$ to $\mathbf{x_{t+1}}$ based on the equations

$$\mathbf{d} = (1-\alpha)\mathbf{b} + \alpha\frac{1}{\mathbf{U}(1,1+r)}\mathbf{v} \quad (1)$$

$$\mathbf{x}^* = \mathbf{x_t} + f\mathbf{d} + \mathbf{g} + \mathbf{U}(-r,r) \quad (2)$$

$$\mathbf{x_{t+1}} = \begin{cases} \mathbf{x}^* & \text{if } w(\mathbf{x}^*) > 0 \\ \mathbf{x_t} & \text{otherwise,} \end{cases} \quad (3)$$

where $\mathbf{g}$ is a global gravity vector, α is a tuning parameter that blends the splat motion bias with per-vertex velocity, which we set to 0.33, $\mathbf{U}(a,b)$ is a uniform random variable between a and b, and $w(\mathbf{x})$ is the wet map value at position $\mathbf{x}$. Based on splat parameters and random numbers, a new candidate position $\mathbf{x}^*$ is computed, and then if the canvas is wet at that position, the vertex is updated; otherwise, the vertex does not move. See Fig. 3 for an illustration. Once a splat's vertices have all been updated, its opacity is recomputed by conserving the "amount of pigment," defined as the opacity times the area.

Conceptually, these equations embody the biased random walk behavior depicted by particles in real watercolor paint flow. Each vertex's motion is first a combination of its own velocity and the splat's bias velocity—this allows the splat to continue to expand in area, while also effectively moving according to some water velocity. The vertex velocity is varied by the roughness to reproduce semipermeability of the canvas media. The overall motion vector $\mathbf{d}$ can be restricted by the flow pigment property which corresponds to the inverse of paint viscosity. Gravity further biases all splats to provide a preferred direction for paint to flow in. The final random parameter, determined by roughness, is the source of the random walk behavior that encompasses all the branching and roughening aspects of watercolor paint.

After all splats have been updated, the wet map is updated as well by decrementing all the nonzero values stored in it. This effectively reduces the remaining wetness at each cell, and when a cell reaches zero, pigment will no longer be able to flow into that portion of the canvas.

3.3 Sampling Management

Over time, the different advection directions of each vertex in a splat can result in splats that have very nonuniformly sampled boundaries, which creates unrealistically straight hard edges due to local undersampling. Over many splats, this artifact can create a faceted, polygonal appearance in strokes, especially as they turn corners or have fine water features that cause neighboring splats to have very different wetness values. We have two strategies for dealing with this artifact.

The first is to constrain the motion of vertices within a splat to not be too far from their neighbors. For a splat, each vertex i's new position is computed, $\mathbf{x}^{\mathbf{i}}_{\mathbf{t+1}}$, and then compared to its neighbors' old positions, $\mathbf{x}^{\mathbf{i-1}}_{\mathbf{t}}$ and $\mathbf{x}^{\mathbf{i+1}}_{\mathbf{t}}$, and if the distances between either pair is above some threshold, vertex i's position is not updated (that is, $\mathbf{x}^{\mathbf{i}}_{\mathbf{t+1}} = \mathbf{x}^{\mathbf{i}}_{\mathbf{t}}$).

The second strategy is to periodically resample each splat's boundary. To do this, the splat's total perimeter is computed, and divided by the number of vertices to compute the arc length per vertex. Then, starting at an arbitrary location, the splat's vertices are moved to uniform arc length increments along the boundary.

Both strategies also have the effect of somewhat smoothing the splat boundary, while limiting the polygonal artifacts that may appear in the final stroke appearance. The constrained motion is fast to perform and has the desired effect, but also ultimately limits the propagation of the paint somewhat, which can alter a stroke's appearance in other ways. Boundary resampling is slower by comparison, but achieves higher quality final output so this is what we use. Fig. 3 shows the effect of boundary resampling on splat shape.

3.4 Lifetime Management

A splat has three stages of life: flowing, fixed, and dried. When first added to the canvas, a splat is flowing. After some number of steps (the age parameter of the brush type, different from the canvas wetness), the splat stops being advected and is in the fixed state. While fixed, the splat can potentially be rewetted to resume advection. After a period of time without moving, the pigment is determined to have permanently stained the canvas and the splat is considered dried. Once dried, the splat is rasterized into a dry pigment buffer and removed from the simulation. This keeps the rendering cost for the canvas capped as the artist continues to make many strokes in a painting.

The addition of water can rewet fixed splats to achieve effects like back runs and feathered edges. When water is applied to the canvas, fixed splats' vertices are checked to see if they overlap with the new water, and if they do, some of those vertices are unfixed and the splat's age is reset. To determine if a rewetted vertex becomes unfixed, a random number is tested against a threshold based on how close the splat is to being totally dry and how strong the water's unfixing property is set.

Once a splat has become fixed, its advection under its initial momentum is complete and so the splat's per-vertex velocity vectors are set to zero, along with the splat's bias vector. When the splat is rewetted, the direction of the vertex motion is determined by the water placement, mimicking the effect of the flow of real water through an absorptive canvas. When the wet regions are rasterized, they also write a radially outward pointing per-pixel "water velocity" vector to the wet map. Then, when the rewetted vertices are updated via (1), the splat vertex's $\mathbf{v}$ vector is replaced with the vector sampled from the water velocity at the vertex's location, $\mathbf{v_w}(\mathbf{x_t})$.

In addition to controlling splat advection, the age of the splat also determines the impact of the granulation texture. As a splat dries, granulation has a larger impact on its appearance, so a texture (generated from scanned, real watercolor brush strokes) is used to apply a small delta to the splat's opacity, with a weight based on how close to dry the splat is. The granulation texture is a single image that covers the entire canvas, and each splat uses its canvas coordinates plus a small random jitter to sample from it. Granulation effects can be seen in Fig. 4.

3.5 Brush Types

We implemented five brush types in our model that reproduce a variety of characteristic watercolor strokes, though many more variations are possible. The brush types are simple, wet-on-dry, wet-on-wet, blobby, and crunchy. Each uses a different arrangement of splats per stamp (see

Fig. 2), and has different settings for the brush parameters. Each stroke in Fig. 2 has a target width $w = 45$ pixels. The wet map cells are set to 255 when wetted, and the simulation is run at 30 Hz, so they take 8.5 seconds to dry. Unless otherwise specified, splats have life $l = 30$ (so 1 second), roughness of $r = 1$ pixel, and flow of $f = 100\%$. The gravity vector is $\langle 0, 0 \rangle$.

The simple brush uses a single splat per stamp, and demonstrates nonuniform pigment distribution because of the random nature of the splat evolution, and blending between strokes due to the advection into surrounding wet portions of the canvas. The splat's diameter $d = w$ and its bias vector is $\mathbf{b} = \langle 0, 0 \rangle$.

Wet-on-dry places seven splats per stamp, with six arranged around the perimeter of the seventh. The center splat has $\mathbf{b} = \langle 0, 0 \rangle$, while the six perimeter splats at $\theta = \{0, \frac{\pi}{6}, \ldots, \frac{5\pi}{6}\}$ have $\mathbf{b} = \langle \frac{d}{2}\cos\theta, \frac{d}{2}\sin\theta \rangle$, where $d = \frac{w}{2}$. This causes the perimeter splats to collect around the edges of the stroke, adding additional pigment for an edge darkening appearance.

Wet-on-wet places a small splat ($d = \frac{w}{2}$) inside a larger splat ($d = \frac{3w}{2}$) for each stamp. This results in a progressively darker region toward the center of the stroke for a feathered appearance characteristic of applying pigment to a wet canvas. Both splats have $r = 5$ pixels, $l = 15$ steps and $\mathbf{b} = \langle 0, 0 \rangle$.

The blobby brush places four randomly sized splats ($d \in [\frac{w}{3}, w]$) in a cross pattern per stamp, with $l = 15$ steps and $\mathbf{b} = \langle 0, 0 \rangle$. Along with adding some noise to the splats' color, this creates a more heavily nonuniform stroke shape and pigment density, similar to watercolor paint applied to a rougher canvas or with more granular pigment.

Finally, the crunchy brush places one splat per stamp ($d = w$), but sets $r = 5$ pixels, $f = 25\%$, and $l = 15$ steps to reduce how far the vertices propagate while increasing the influence of the random walk component of the advection. The result is a stroke appearance that exhibits more significant branching and broken stroke texture, with very rough edges.

See Fig. 4 for a comparison between real watercolor paint and the types of effects our algorithm is able to generate.

4 Application

In order to create a usable painting application based on our algorithm, a number of considerations are necessary. We have made both desktop and iPad applications implementing our algorithm for real-time dynamic watercolor painting.

4.1 Interactive Rendering

Our interactive painting application uses OpenGL to render the splats and dried pigment buffer. Since splats can be concave or even self-intersecting polygons, standard rendering approaches are not appropriate. A typical solution is to tessellate the polygon, but since our polygons are changing shape every frame, the overhead of tessellation is too high. Instead, we use a two pass stencil buffer approach per-splat [20]. For the first pass, the color buffer is masked out and the stencil operation is set to invert, and the splat vertices are rendered as a triangle fan. This results in the stencil buffer storing a value of one everywhere the polygon is filled and zero elsewhere. The second pass writes to the color buffer and sets the stencil test to equal to one and the stencil operation to zero, and renders a quad that covers only the bounding box of the single splat. This colors the pixels interior to the polygon with proper winding rules, and also resets the stencil buffer to all zeroes for the next splat, avoiding the need for additional passes.

OpenGL creates aliased polygons that appear jagged on screen, but full screen antialiasing can be expensive and may not be supported on slate devices. To smooth the results in our interactive painting application, we use a fast post-processing filter that performs an adaptive per-pixel directional blur based on a set of heuristics [17]. For offline rendering at arbitrary resolutions, we use a software polygon rasterizer that includes high quality, analytical antialiasing.

In our interactive system, feedback is provided about the current state of the wet map by darkening the document in wet regions. This allows the user to know when their strokes will be affected by preexisting canvas wetness. Further feedback about splat life stage is given by additional darkening of splats that are currently flowing. In this way, the effect of rewetting can be predicted, as dried splats will not respond to additional water, whereas merely fixed splats will.

4.2 User Interface

The basic form of our desktop application presents the canvas alongside a GUI for controlling the paint parameters, such as selecting brush type and size, or changing the canvas texture. Users can paint with a mouse in the standard case, or they can use a more sophisticated input device such as a Wacom tablet. The stylus pressure can be mapped to the brush size, so that users can dynamically control the size of the mark made by varying the stroke pressure. Similarly, stylus tilt or rotation can be mapped to influence the stroke splats' bias vectors to create strokes with a directional preference for advection. This can be useful for applying paint that makes a hard edge on one side and a smooth falloff on the opposite side.

Conversely, our iPad application has different affordances and a correspondingly different interface. Only finger-based input is available, so no pressure or tilt sensitivity is possible. On the other hand, the iPad's accelerometer can be used to control the global gravity vector, to make a physical mapping between the orientation of the iPad and the pigment's flow. Furthermore, the iPad's small screen encourages maximizing screen real estate for painting, so we do not present any on-screen interface and let the canvas occupy the entire space. Settings are called up using a specific gesture on the canvas, which allows the user to control a subset of the features of the full algorithm.

4.3 Vector Output

When a canvas is saved, two output files are generated: a bitmap of the rendered image as a PNG, and a vector representation of all the splats (including dried) as an SVG [18]. See Fig. 5 for an example of the SVG vector data directly embedded in this document. Fig. 1 shows progressive zooms into a painting, maintaining polygon detail even under extreme magnification.

While the SVG specification includes support for advanced features such as image filter effects and blend

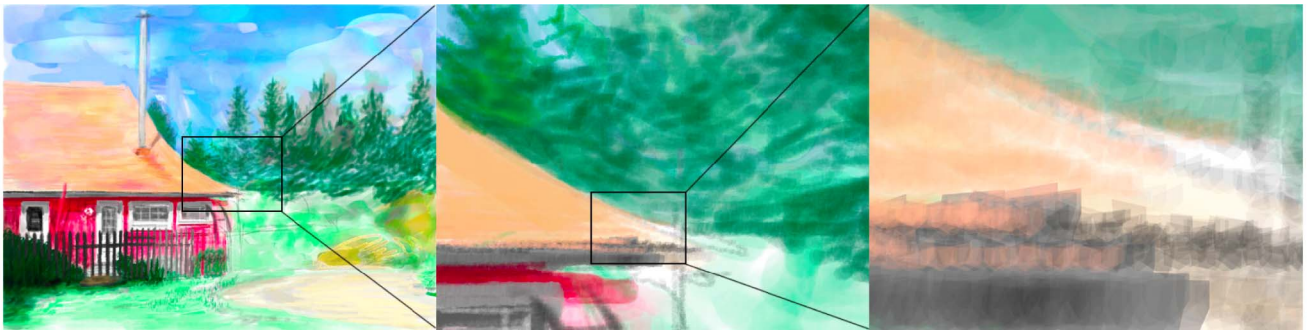

Fig. 1. A vector watercolor painting made in our interactive iPad painting application, displaying complex texture and color blending. Insets zoom in to show stroke detail and resolution independence.

modes that could duplicate our granulation texture shading math, the unfortunate state of affairs is that no commonly available SVG viewer supports those features. In fact, the complexity of the SVGs we generate, even without texture but just with many overlapping transparent paths, is such that available viewers may struggle to display them efficiently. Conversely, our interactive application is able to display the same data significantly faster, due to its use of GPU acceleration for rendering. Therefore, for now, we limit our SVG output to store only transparent filled paths, and leave the granulation effects for a time when more capable SVG viewers are available.

5 Results

The use of different brush types to create different painting effects can be seen in Fig. 6. Fig. 7 shows a real watercolor painting by one of the authors, side-by-side with the same composition created in our prototype. While specific details differ between the two, the digital version reproduces many desirable qualities of the original.

Because of the low computational requirements of our algorithm, we were able to release a version of it as the iPad application called Adobe Eazel for Photoshop CS5 [19]. This version includes a basic feature set (only wet-on-dry brush, no granulation, no rewetting). It runs at the iPad's native resolution ($1{,}024 \times 768$) and has the ability to send the document to a desktop version of Photoshop to rerender the document at a higher resolution with high-quality antialiasing.

As a result of this public release, our algorithm was used by many artists of many different experience levels and aesthetic styles, which has resulted in a wide variety of different compositions (see Fig. 11).

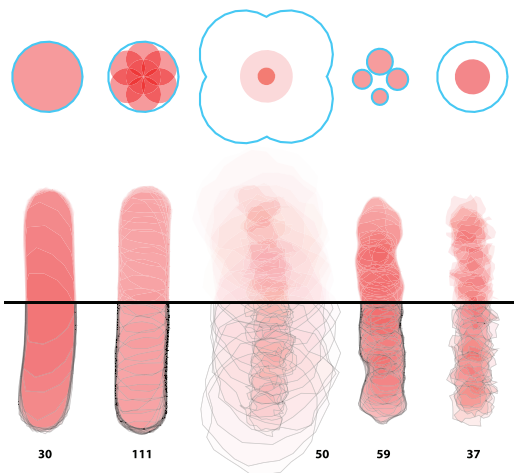

Fig. 2. Initial splat configurations and resulting stroke for each brush type. *Left to right:* simple, wet-on-dry, wet-on-wet, blobby, crunchy. Cyan outlines indicate the water region per-stamp. Black outlines on the bottom half of each stroke indicate final splat shapes. The total number of splats in each stroke is also indicated.

5.1 Performance

The nature of our formulation means that performance in terms of resolution is not particularly important, because the interactive rasterization doesn't impact the final output. Instead, we are limited by the number of active splats being animated and rendered simultaneously. Specifically, rendering is the most costly step, especially on low-powered devices like the iPad.

We were able to improve the rendering performance by only drawing a subset of the flowing splats, at an adjusted opacity to create a similar bulk stroke appearance. Once splats dry, all splats are rendered into the dried buffer at the correct opacity, creating a smoother, higher quality final stroke.

We tested our performance on an iPad 2 with a 1 GHz Apple A5 processor and PowerVR SGX 543MP2 GPU, and on a MacBook Pro with a 2.2 GHz Intel Core i7 processor and an AMD Radeon HD 6750M graphics card, to see how many splats could be rendered simultaneously while maintaining interactive frame rates. The resulting data is in Table 1. To put these numbers in context, the painting from Fig. 1 is a typical example made on a desktop computer, which took 383,145 simulation steps and totals 685,898 splats over 18,192 strokes, with an average simultaneously active splat count of 134 and maximum of 6,447.

The release of the new iPad (aka iPad 3) introduces a much higher resolution target, at $2{,}048 \times 1{,}536$ or four times

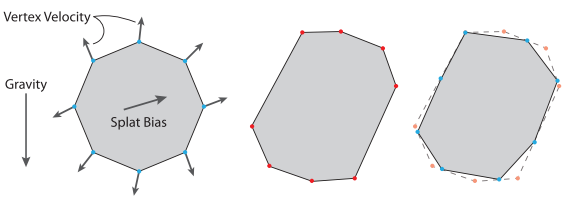

Fig. 3. Each splats' motion is dictated by a per-vertex velocity, a splat bias, and a global gravity. *From left to right:* A splat's initial configuration, after advection, and after boundary resampling.

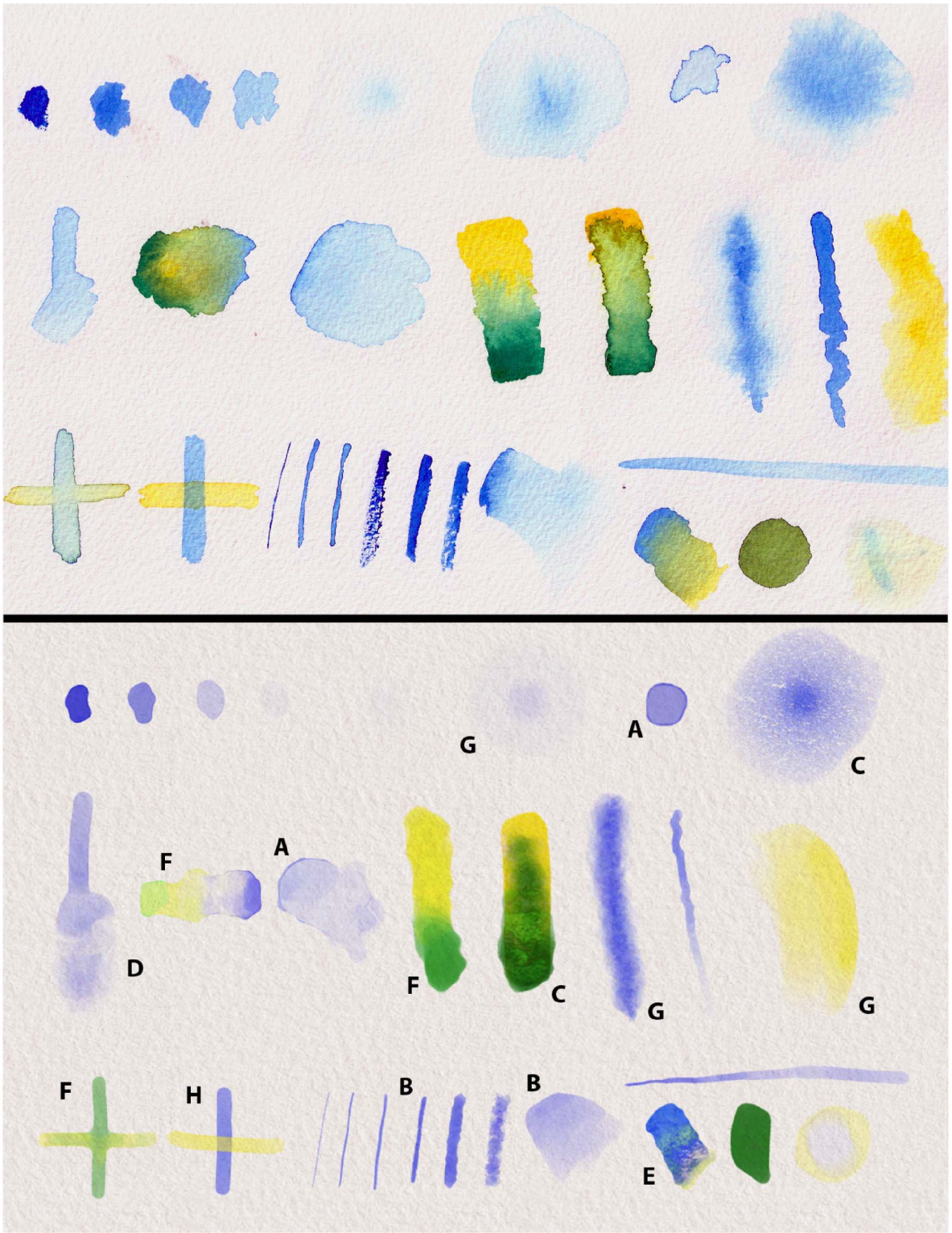

Fig. 4. A comparison of similar strokes made with real watercolor paint (top) versus our algorithm (bottom). Paper texture is added to our results for comparison purposes. The strokes are chosen to showcase a variety of characteristic watercolor behaviors, including edge darkening (A), nonuniform pigment density (B), granulation (C), rewetting (D), back runs (E), color blending (F), feathering (G), and glazing (H). Strokes exemplifying particular effects have been labeled with the corresponding letter. While our algorithm does not make identical strokes, it exhibits the same range and depth of expressiveness as traditional watercolor.

as many pixels, combined with the 1 GHz Apple A5X processor and PowerVR SGX 543MP4 GPU. As our algorithm is resolution-agnostic, it is straightforward to adapt it to the new screen (keeping the same wet map resolution of $1{,}024 \times 768$), and the new GPU provides enough throughput that there is a limited performance degradation, as shown in Table 1.

The fact that the number of active splats dictates performance implies that the way to manage runtime performance of the application is to limit that number.

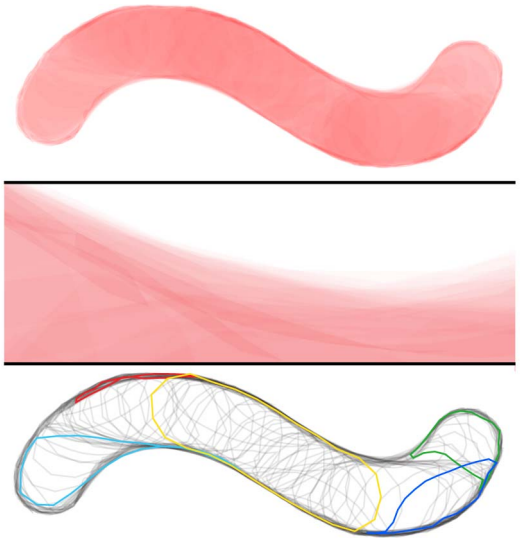

Fig. 5. Top : The vector output of a brush stroke made from left to right. Zoom in to see resolution independence. $Middle$: Rasterized version, zoomed in to 800 percent to show detail. $Bottom$: All 195 splats outlined instead of filled, with a few splats highlighted to show shape variations. Vertices on the edge of the wet region have a tendency to get "stuck" as their motion is restricted, which can result in teardrop shapes.

Toward that end, we set the max lifetime of the different brush types such that only a target number of splats are flowing at the same time for average painting speeds.

Intuitively, thicker strokes made by larger brushes should be more costly to render because of their higher pixel count (especially for limited pixel fill-rate devices like the iPad), but they are actually faster because fewer splats are placed per path length of stroke. This is because stamp spacing is set as a percentage of stamp size, so thicker strokes have more distance between consecutive stamps and result in fewer active splats while painting. Very small brush strokes result in many more simultaneous active splats and therefore are much slower to render. To combat this effect, we reduce the lifetime of splats in small strokes, to keep the target number of active splats more manageable.

Fig. 6. A painting made with our desktop application, which shows how the different brush types in Fig. 2 can be applied. The brushes used in each region are (A) simple, (B) wet-on-dry, (C) wet-on-wet, (D) blobby, (E) crunchy, and (F) with rough canvas texture.

6 Discussion

As a result of our public release, we have been able to collect a large amount of feedback from users on the painting application, and also to see how a wide variety of users are successful with, or struggle with, creating their compositions.

6.1 User Feedback

In general, user reactions have been mixed. A significant amount of the iPad application feedback has focused on the interface and on other painting features such as undo and layers. A common lament was, "I wish I had some sort of pen or brush input device." Specifically regarding the actual dynamic paint behavior, while many users have been impressed by the appearance, there is difficulty in learning

Fig. 7. *Left:* An original watercolor painting by one of the authors. *Right:* A digital reproduction by the same artist in our desktop application. While not identical, it achieves a very similar style and feel, and reproduces many of the salient features. Insets on each side zoom into marked portions to show fine detail.

TABLE 1
Approximate Number of Splats that Can Be Simulated and Rendered Simultaneously at Different Frame Rates and on Different Hardware

ms/frame	iPad 2	iPad 3	MacBook Pro
33	400	300	4000
100	1000	700	10000

to control it—one user said, "I definitely need to work with it some more and I have a feeling I'll start catching on to how to better work with it." Another later said, "The interaction feels very natural to me now." This is quite similar to traditional watercolor paints, which are some of the most difficult artistic media to master. The artists who persevered and gained greater control over the app (see Fig. 11, for examples) were able to use the blending behaviors to achieve complex color gradients and textures that would have been very difficult to make with other digital paint tools. Some of their comments on the behavior: "I like the 'flow' in the paint, works well when painting using opacity," "Its like trying to paint with superwet watercolor," and "Great water-colour-ish style here." It is difficult to objectively state whether the comments are "good" or "bad"—as with any other artistic medium, personal style of the artist is a dominant influence, and where some artists will find confusion and frustration, others find success and serendipity, just as with real watercolors.

6.2 Limitations

While having many advantages, our novel formulation also has disadvantages over other paint simulation methods. The sparse, vector representation of paint is good for reducing computational complexity, but it does raise the cost for dense detail, which means vector representations of granulation texture, etc., are infeasible with purely vector data. Heavy branching, intricate flow, and complex textures are all difficult to represent with pure vector formats, which is why we rely on a raster technique for pigment granulation.

Furthermore, because of the procedural approach to the pigment evolution, simulating the full range of real paint behaviors becomes increasingly difficult and the algorithmic complexity does not scale well. Conversely, a physical simulation can add the physical parameters into a unified framework in a straightforward manner to reproduce other types of paint such as acrylics or oil paints, or to modify real physical properties of pigment such as viscosity or particle size. The fidelity with which real paint can be faithfully reproduced with a physical simulation is higher than with our approach. Our goal was to achieve a balance between interactivity and quality, so we focus on a reproducing a similar range of dynamic behaviors, but with a distinct expressive experience.

7 Analysis

Given that our algorithm relies so heavily on the rasterization of complex polygons for its performance, it is worth investigating the optimal approach among available technologies. Graphics cards are carefully tuned to rasterize triangles as fast as possible, but complex polygons have many features that interfere with the standard graphics pipeline. First, complex polygons have more than three vertices (in our case, 25), which makes the potential variation in shape much greater. Second, those shape variations include concave polygons which violate some fundamental assumptions of triangle rasterization, such as the half-space test for determining triangle interior. Third, the polygon can actually intersect itself and can even create internal holes by turning back in on itself, which require proper handling of winding rules for correct rasterization (see Figs. 8 and 9, for examples).

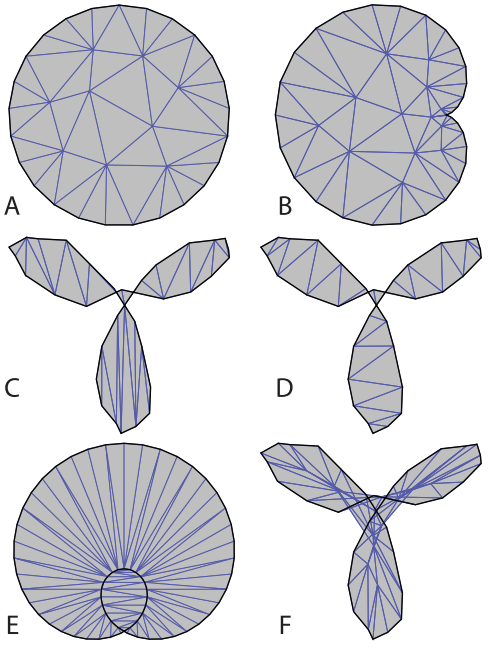

Fig. 8. Examples of output for the triangle-based techniques. (a) `mvc` pretriangulation of circle. (b) `mvc` output positions for concave boundary. (c) `glutess` triangulation of self-intersection polygon. (d) `triangle` on the same. (e) `triangle` fails to respect winding rules, erroneously filling the inner loop. (f) `mvc` fails to keep triangulation inside self-intersecting boundary.

Since correct rasterization of complex polygons is so onerous, conventional wisdom is to triangulate the polygon first and then use the graphics cards to rasterize the output set of triangles. The other common suggestion is to use the stencil buffer and a two-pass approach, but that the overhead of OpenGL state changes per-polygon will be

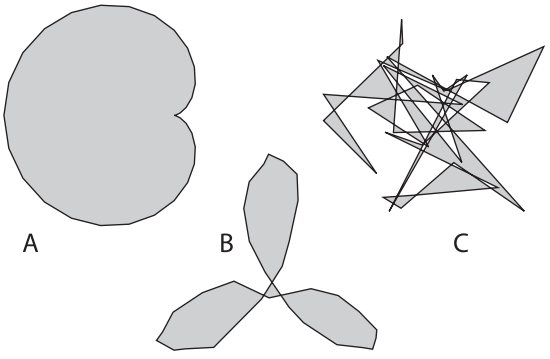

Fig. 9. Three example complex polygons used for performance analysis. (a) A concave cardioid. (b) A self-intersecting rose curve with per-vertex noise added. (c) A random polygon exhibiting significant self-intersection and self-overlap.

prohibitive. Furthermore, the implications of each of these strategies differ between cases where static polygons will be drawn many times (perhaps under affine transform) versus dynamic polygons that change shape frame to frame. Finally, many commercial applications that draw complex polygons (such as Adobe Illustrator [13]) rely on software rasterization.

Our application's requirements are somewhat unique. While we need to render many complex (self-intersecting and self-overlapping) polygons every frame, they are dynamic so triangulations cannot be reused frame to frame. On the other hand, since our results depend on the bulk appearance of many polygons, we are tolerant of rendering errors per-polygon, such as incorrect winding rules. This grants us flexibility to consider approaches that may not work in other contexts. To understand these tradeoffs, we experimented with six different techniques.

Two-Pass Stencil Buffer (`stencil`). As described in Section 4.1, the OpenGL stencil buffer can be used in a two-pass algorithm to render complex polygons, detailed in the OpenGL Programming Guide [20]. The first pass writes to the stencil buffer to set up a coverage mask, and the second pass uses the mask to restrict writing to the color buffer with a fill color. The downsides to this approach are that it requires multiple OpenGL state changes per polygon, and requires a significant amount of overdraw, which can impact its performance for small or self-overlapping polygons.

NV Path Rendering (`nvpr`). NVIDIA recently introduced the NV_path_rendering extension to OpenGL [21], which provides first-class support for filling and stroking complex polygons (including quadratic and cubic beziers) to better enable hardware acceleration for vector rendering applications. Ultimately, it provides a formalized and more complete way of performing the same two-pass stencil buffer approach, but with expanded support for curve specification, OpenGL 3D integration, and complex shading. For our limited use case, it does not present significant differences from `stencil`, but its performance may benefit from first-class support.

GLU Tessellation (`glutess`). The OpenGL Utility Library (GLU) accompanies OpenGL and has included a polygon tessellation facility since its inception [20]. Its goal is to provide robustness in the case of complex polygons involving concavity, self-intersection, self-overlap, and multiple contours (i.e., holes), and so it correctly implements winding rules as well. The output of the tessellator is a set of nonoverlapping triangles that can be rendered normally (see Fig. 8c). Unfortunately, different platforms have implementations of varying quality and we found that tessellation quietly failed on some polygons (e.g., a polygon with multiple self-intersections at the same point) and simply produced no output.

Triangle Tessellation (`triangle`). A more reliable triangulation alternative is the Triangle library by Shewchuk [22]. The goal of the library is to create constrained Delaunay triangulations for mesh computations, but it includes a very carefully engineered triangulation engine that is extremely robust. On the other hand, correct rasterization of complex polygons are not its target, and so it does not include support for proper winding rules. Its output is also a set of nonoverlapping polygons which we then render normally with OpenGL. See Figs. 8d and 8e, for examples.

Mean Value Coordinates (`mvc`). An alternative to retriangulating a polygon for each new configuration of vertices is to triangulate it once when it is initialized (as a circle, see Fig. 8a) and then update the positions of the internal vertices as the perimeter vertices are moved. This approach uses mean value coordinates [23], [24] which are a generalization of barycentric coordinates to arbitrary polygons, and provide a smooth function that interpolates among n vertices. Given a triangulated set of vertices $\mathbb{W} = \{\mathbb{B}, \mathbb{U}\}$ and edges $\mathbb{E}$, where $\mathbb{B}$ are the polygon boundary vertices and $\mathbb{U}$ are the interior vertices. For each $v_i \in \mathbb{W}$, we compute its mean value coordinates $\lambda_{i,j}$ to each other vertex, where $\{v_i, v_j\} \notin \mathbb{E} \Rightarrow \lambda_{i,j} = 0$. The property of mean value coordinates is such that

$$v_i = \sum \lambda_{i,j} v_j \tag{4}$$

$$u_i = \sum \lambda_{i,j} u_j + \sum \lambda_{i,k} b_k \tag{5}$$

$$u_i - \sum \lambda_{i,j} u_j = \sum \lambda_{i,k} b_k \tag{6}$$

$$\mathrm{Ax} = \mathrm{By}, \tag{7}$$

where $u_i \in \mathbb{U}$ and $b_i \in \mathbb{B}$. Equation (7) is the system of equations we solve, where x is an unknown vector of interior vertex positions, y is the given vector of boundary vertex positions, and A and B are matrices constructed from (6). We precompute an LU decomposition of A, and then for a new set of boundary positions, compute By and find x by forward backward substitution. See Fig. 8b for an example output.

The advantage of this approach is that it has a constant cost, regardless of the polygon shape's complexity, whereas triangulation can take dramatically longer for difficult polygons. The downside, however, is that it only produces correct triangulations for (most) concave polygons, and self-intersecting or self-overlapping polygons may have internal vertices outside the polygon boundary (see Fig. 8f), and winding rules will not be respected. The other potential pitfall is that performance is $O(n^2)$, whereas triangulation should be possible in $O(n \log n)$, but at what value of n one outperforms the other is difficult to determine as triangulation performance depends heavily on the complexity of the polygon.

Software Rasterization (`agm`). The standard approach for rasterizing complex polygons is software rasterization using a scanline algorithm. The process can be made more efficient by computing a trapezoidal decomposition first [25]. Vector rendering programs such as Adobe Illustrator that have existed since before GPU computing was common rely on basic algorithms like this for their rendering engines. For our experiment, we used the rasterizer from Illustrator (called AGM) which has been carefully tuned over many years for good performance, but is a single-threaded CPU implementation. AGM does not produce identical results to an OpenGL method because its rasterization implements "overscan" (conservative rasterization), whereas OpenGL is "centerscan." This means that AGM turns on any pixel touched by the polygon, instead of any

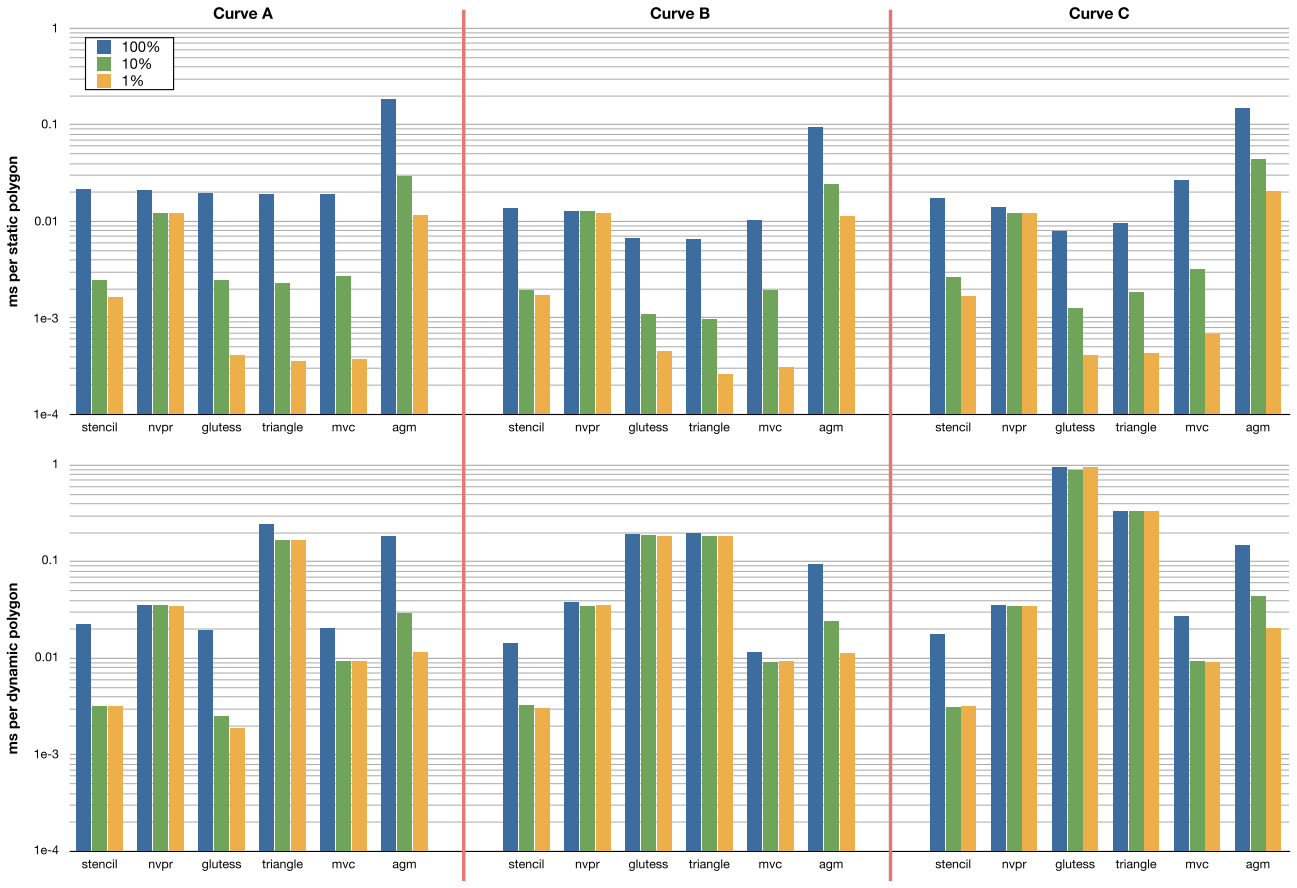

Fig. 10. Comparison of rasterization performance for three example curves (see Fig. 9) at three different zoom levels, for six different rasterization techniques, in static and dynamic polygon cases. Times are in milliseconds, and are averaged over 1,000 iterations. The different color bars show each curve drawn at one of three scales.

pixel whose center is inside the polygon, but it is a small difference in most cases.

7.1 Experiments

We measured the rendering performance of each of the six techniques on three different example polygons (see Fig. 9) at three different zoom levels, in both static and dynamic cases. All measurements were made on a Mac Pro with two 2.27 GHz quad core Intel Xeon E5520 processors and an NVIDIA Quadro 4000 graphics card. In the dynamic case, the polygon's shape is different each time it is drawn, and so setup (such as triangulation) must be done every time. The static case is when a polygon is drawn many times with the same shape (modulo affine transforms), so the setup cost is omitted.

Since a large portion of the cost of drawing triangles is the streaming of vertex data to the GPU, for the static polygon case we also experimented with using display lists and vertex buffers (VBOs). Ultimately, we found that display lists yield the best performance when the setup cost is not considered, so that is what we report in our results. The VBO setup cost was actually less than the display list creation, so the preferred method depends on the specific ratio of setups to draws and the relative cost of each for a target application.

The example polygons we measured test different aspects of each of the rasterization methods. Each polygon has 32 vertices. Fig. 9a is a concave cardioid, but otherwise normal, Fig. 9b is a rose curve with simple self-intersection (with noise added to avoid triangulation degeneracies), and Fig. 9c is random, with heavy self-intersection and complex self-overlap. At full size on a 512×512 canvas, they are 164,480, 53,315, and 7,512 pixels, respectively. For each polygon, we tested at three different scales, so the number of pixels rasterized were 100, 10, and 1 percent of the full sizes.

7.2 Results

The results of our experiments are presented in Fig. 10. There are a number of interesting observations that can be made about the data.

Considering the dynamic case (our use case), across all polygons and all scales, `stencil` is either the fastest or tied for fastest, despite the cost of frequent OpenGL state changes. Therefore, we used it in our application. It is also clear that the cost of triangulation is prohibitive if the polygon is only to be drawn once, easily an order of magnitude slower for complex polygons, and `agm` can outperform `glutess` and `triangle` in some cases. Finally, `mvc` and `nvpr` are both limited by overhead costs. It is worth noting that `agm` is the only algorithm to scale linearly with number of pixels in this case—all the others are overridden by other costs for small polygons.

The static polygon case presents a somewhat different picture. First, `agm` is included for comparison, though its implementation and performance are unchanged from the

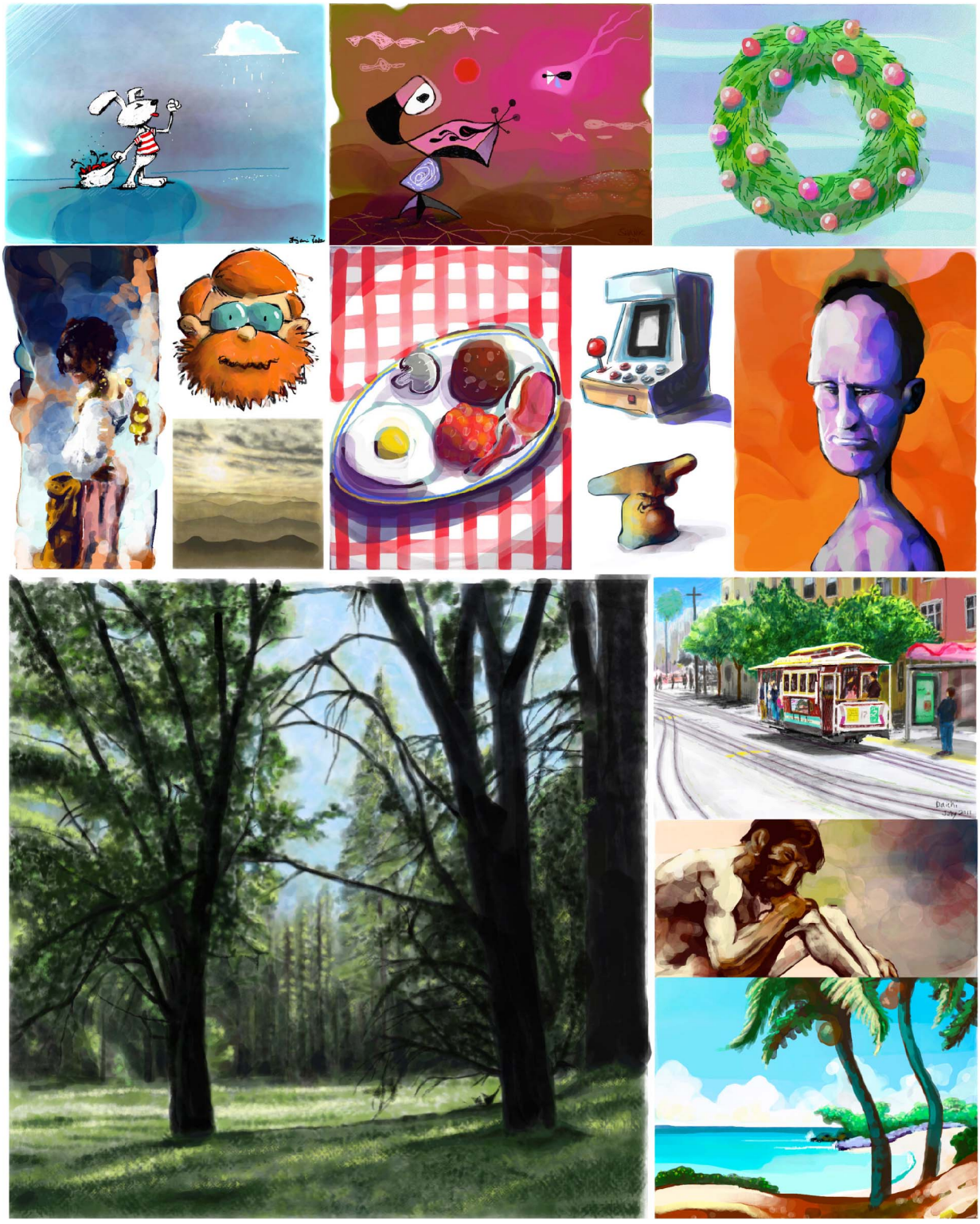

Fig. 11. Examples of paintings by professional artists exhibiting different styles characteristic of watercolor painting, using our desktop and iPad software. Note the range of achievable appearances of our algorithm. Used with permission.

dynamic case. It is heartening to see that without the overhead costs for `glutess`, `triangle`, and `mvc`, they scale linearly with number of pixels. The OpenGL state change overhead of `stencil` is more apparent, comparatively. Also, even for large polygons, the performance of `stencil` is worse than triangulation approaches, because it requires overdraw while the triangulation methods only render each output pixel once. Overdraw is also the reason why `mvc` performs worse, as it is unable to create holes in the output and so rasterizes more pixels than it should. Finally, `nvpr` is similar to `stencil` for large polygons, but the overhead costs still dominate for smaller sizes.

It is interesting to reexamine the conventional wisdom about rasterization of complex polygons in light of this data.

Perhaps most important is that if a polygon is only going to be drawn once (or maybe a few times), triangulation is clearly the worst solution and should only be used if its particular output is needed, but even then, robustness is a concern. Any of the alternatives are faster, even software rasterization in many cases. On the other hand, when rasterizing the same polygon many times, the overhead of the two-pass stencil buffer approach is not so high as to rule it out. While tessellation is faster, particularly for small polygons, robustness concerns may again be a factor in making the final decision.

8 Conclusion

We have presented a novel formulation of a dynamic and interactive paint engine that reproduces many of the important behaviors of watercolor paint. The splat model is lightweight enough to allow low-powered devices such as iPads to easily run the painting application. The vector representation makes rendering at any resolution possible with high-quality antialiasing for final output. We demonstrated that through exposed parameters of the model, a wide variety of different types of brushes and effects can be achieved, allowing a full range of expressive and creative possibilities.

We explored a number of algorithms available for rendering complex polygons to understand their relative performance characteristics and limitations and determine the option that best matched our use case. The result that two-pass OpenGL stencil rendering is best is somewhat surprising and opposes the conventional wisdom in this regard.

To evaluate the utility of our algorithm, we commercially released an iPad application that implements a subset of the full algorithm's functionality. It was downloaded over 4,500 times the first day and achieved a position of seventh best selling paid application in the store.

While watercolor paint simulation was our inspiration, the difficulty of achieving an experience that matches artists' carefully trained expectations about paint behavior is extremely high, and unmet expectations can greatly sour the experience (a sort of uncanny valley effect). Therefore, the goal of the algorithm was not to duplicate the experience of watercolor painting on a computer, but to create a range of dynamic behaviors that allow users to achieve a similar style of process and result, while at the same time having a unique character of its own. With time and further development, the particular feeling of our algorithm may attain its own aesthetic that users strive for, in a way that other media have their adherents. In this respect, we have created something new in the spectrum of traditional to digital media.

Future work is to continue to explore the range of effects that are possible with this type of algorithm, and to see if other paint media can be emulated, or other useful artistic tools. The vector splat formulation may have potential in other simulation-inspired domains as well, where compute power is not available for full-fledged physical simulations. From the application development perspective, we hope to continue to iterate on our interface, including common painting features such as layers, an interface to let users define new brush types, and improving the ability for users to more easily work from photos or other source material.

Acknowledgments

The authors would like to thank all of the beta testers who provided invaluable feedback. Special thanks to our contributing artists: John Derry, Paul Kercal, Mike Miller, Benjamin Rabe, Don Shank, and Mike Shaw.

References

[1] N. Chu and C.-L. Tai, "MoXi: Real-Time Ink Dispersion in Absorbent Paper," *Proc. ACM SIGGRAPH,* pp. 504-511, 2005.
[2] T. Van Laerhoven and F. Van Reeth, "Real-Time Simulation of Watery Paint," *Computer Animation and Virtual Worlds,* vol. 16, pp. 429-439, Jul. 2005.
[3] S. DiVerdi, A. Krishnaswamy, R. Měch, and D. Ito, "A Lightweight, Procedural, Vector Watercolor Painting Engine," *Proc. ACM SIGGRAPH Symp. Interactive 3D Graphics and Games,* pp. 63-70, Mar. 2012.
[4] A. Hertzmann, "A Survey of Stroke-Based Rendering," *IEEE Computer Graphics and Applications,* vol. 23, no. 4, pp. 70-81, July 2003.
[5] A. Hertzmann and K. Perlin, "Painterly Rendering for Video and Interaction," *Proc. Int'l Symp. Nonphotorealistic Animation and Rendering,* pp. 7-12, 2000.
[6] J. Lu, P.V. Sander, and A. Finkelstein, "Interactive Painterly Stylization of Images, Videos and 3D Animations," *Proc. Interactive 3D Graphics and Games,* pp. 127-134, 2010.
[7] A. Bousseau, M. Kaplan, J. Thollot, and F.X. Sillion, "Interactive Watercolor Rendering with Temporal Coherence and Abstraction," *Proc. Int'l Symp. Nonphotorealistic Animation and Rendering,* pp. 141-149, 2006.
[8] Corel, "Painter 12," http://www.corel.com/, 2011.
[9] Ambient Design, "ArtRage 3.5," http://www.ambientdesign.com/, 2011.
[10] C.J. Curtis, S.E. Anderson, J.E. Seims, K.W. Fleischer, and D.H. Salesin, "Computer-Generated Watercolor," *Proc. ACM SIGGRAPH,* 1997.
[11] W. Baxter, J. Wendt, and M.C. Lin, "IMPaSTo: A Realistic, Interactive Model for Paint," *Proc. Int'l Symp. Nonphotorealistic Animation and Rendering,* pp. 45-148, 2004.
[12] N. Chu, W. Baxter, L.-Y. Wei, and N. Govindaraju, "Detail-Preserving Paint Modeling for 3D Brushes," *Proc. Int'l Symp. Nonphotorealistic Animation and Rendering,* pp. 27-34, 2010.
[13] Adobe Systems, Inc., "Illustrator CS5," http://www.adobe.com/, 2010.
[14] G. Lecot and B. Lévy, "ARDECO: Automatic Region DEtection and COnversion," *Proc. Eurographics Symp. Rendering,* 2006.
[15] A. Orzan, A. Bousseau, H. Winnemöller, P. Barla, J. Thollot, and D. Salesin, "Diffusion curves: A Vector Representation for Smooth-Shaded Images," *Proc. ACM SIGGRAPH,* pp. 92:1-92:8, 2008.
[16] R. Ando and R. Tsuruno, "Vector Fluid: A Vector Graphics Depiction of Surface Flow," *Proc. Int'l Symp. Nonphotorealistic Animation and Rendering,* pp. 129-135, 2010.
[17] T. Lottes, "FXAA," NVIDIA, http://developer.nvidia.com/sites/default/files/akamai/gamedev/files/sdk/11/FXAA_WhitePaper.pdf, 2011.
[18] E. Dahlström, P. Dengler, A. Grasso, C. Lilley, C. McCormack, D. Schepers, and J. Watt, "Scalable Vector Graphics (SVG) 1.1 (second ed.)." W3C, http://www.w3.org/TR/2011/REC-SVG11-20110816/, 2011.
[19] Adobe Systems, Inc., "Adobe Eazel for Photoshop CS5," http://itunes.apple.com/us/app/adobe-eazel-for-photoshop/id421302663, 2011.
[20] M. Woo, J. Neider, T. Davis, and D. Shreiner, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2,* third ed. Addison-Wesley Longman Publishing Co., Inc., 1999.
[21] M. Kilgard, "NV_Path_Rendering," http://www.opengl.org/registry/specs/NV/path_rendering.txt, June 2011.
[22] J.R. Shewchuk, "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator," *Applied Computational Geometry: Towards Geometric Engineering,* M. C. Lin and D. Manocha, eds., pp. 203-222, Springer-Verlag, May 1996.
[23] K. Hormann and M. Floater, "Mean Value Coordinates for Arbitrary Planar Polygons," *ACM Trans. Graphics,* vol. 25, no. 4, pp. 1424-1441, Oct. 2006.

[24] T. Ju, S. Schaefer, and J. Warren, "Mean Value Coordinates for Closed Triangular Meshes," *Proc. ACM SIGGRAPH*, pp. 561-566, 2005.
[25] R. Seidel, "A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons," *Computational Geometry: Theory and Applications*, vol. 1, no. 1, pp. 51-64, July 1991.

Stephen DiVerdi received the BS degree from Harvey Mudd College in 2002 and the PhD degree from the University of California, Santa Barbara in 2007. He is a senior research scientist in the Advanced Technology Labs at Adobe Systems, Inc. His research explores how computer graphics and computer vision can enable new types of interactions and experiences. He is a member of the IEEE.

Aravind Krishnaswamy received the BMath and MMath degrees in computer science from the University of Waterloo. He is currently a senior technologist with the Visual Computing Lab at Adobe Systems, Inc. During his time there, he has been involved in the research and development of real-time photo realistic image synthesis, digital painting, and novel experiences on mobile devices. He is a member of the IEEE.

Radomír Měch received the MSc degree from Charles University, Prague in 1993 and the PhD degree from the University of Calgary in 1997. He is a researcher and manager in the Advanced Technology Labs at Adobe Systems, Inc. His group focuses on casual 3D modeling, procedural modeling, rendering and graphics design. His areas of research are procedural modeling, with a particular focus on interaction with procedural models, rendering, and 3D printing. He is a member of the IEEE.

Daichi Ito received the BFA degree in animation/illustration and the MA degree in art from San Jose State University. He is a technical artist in the Advanced Technology Labs at Adobe Systems, Inc. His research focuses on understanding artists' needs and filling the gap between artists and technologies by introducing new ideas and computer programming algorithms.