Example-Based Brushes for Coherent Stylized Renderings

Ming Zheng
ETH Zurich
Disney Research Zurich
mima.zheng@gmail.com

Antoine Milliez
ETH Zurich
Disney Research Zurich
antoine.milliez@disneyresearch.com

Markus Gross
ETH Zurich
Disney Research Zurich
markus.gross@disneyresearch.com

Robert W. Sumner
ETH Zurich
Disney Research Zurich
bob.sumner@disneyresearch.com

Figure 1: These 3D paintings are rendered in screen space using our method with calligraphy and watercolor styles. The paint stroke rendering is temporally coherent as the characters and camera are animated.

ABSTRACT

Painterly stylization is the cornerstone of non-photorealistic rendering. Inspired by the versatility of paint as a physical medium, existing methods target intuitive interfaces that mimic physical brushes, providing artists the ability to intuitively place paint strokes in a digital scene. Other work focuses on physical simulation of the interaction between paint and paper or realistic rendering of wet and dry paint. In our work, we leverage the versatility of example-based methods that can generate paint strokes of arbitrary shape and style based on a collection of images acquired from physical media. Such ideas have gained popularity since they do not require cumbersome physical simulation and achieve high fidelity without the need of a specific model or rule set. However, existing methods are limited to the generation of static 2D paintings and cannot be applied in the context of 3D painting and animation where paint strokes change shape and length as the camera viewport moves. Our method targets this shortcoming by generating temporally-coherent example-based paint strokes that accommodate to such length and shape changes. We demonstrate the robustness of our method with a 2D painting application that provides immediate feedback to the user and show how our brush model can be applied to the screen-space rendering of 3D paintings on a variety of examples.

CCS CONCEPTS

• **Computing methodologies** → **Non-photorealistic rendering**;

KEYWORDS

Stylization, Painterly Rendering, Temporal Coherence

ACM Reference format:
Ming Zheng, Antoine Milliez, Markus Gross, and Robert W. Sumner. 2017. Example-Based Brushes for Coherent Stylized Renderings. In *Proceedings of NPAR'17, Los Angeles, CA, USA, July 28-29, 2017,* 10 pages.
DOI: 10.1145/3092919.3092929

NPAR'17, Los Angeles, CA, USA

978-1-4503-5081-5/17/07…$15.00
DOI: 10.1145/3092919.3092929

1 INTRODUCTION

State of the art tools for digital 2D painting have reached a level of maturity that enables expert users to create artistic masterpieces

using digital painting software. The most popular software packages use a splat based metaphor as the core brush model. In this model, brushes are imitated by repeatedly stamping small brush images along a stroke's centerline. While flexible, splat based models cannot easily reproduce many real-world brush appearances. Thus, more sophisticated methods explore data driven techniques that use scans of real brush strokes created with media such as oil paint or calligraphic ink. These methods reproduce the look of real-world brush strokes with a greater degree of fidelity.

Expressive digital painting is not restricted to 2D. Recent research has elevated digital painting to 3D in the context of computer-generated animation. Using 2D gestures on a tablet, users can place paint strokes on and around 3D characters and environments and even animate the strokes using keyframing tools. In order to preserve the hand-crafted painterly aesthetic while providing the flexibility of traditional digital painting, rendering methods project the 3D strokes into the 2D viewplane and apply standard 2D brush models to generate the rendered image. The result is a dimensional painting that can be viewed from any perspective or even brought to life using animation.

While these 3D paintings offer an entirely new aesthetic for computer generated animation, they employ splat based brush models that are limited in their expressivity and ability to mimic real-world brushes. The core challenge in this domain is supporting coherent brush shape and length changes that are inherent in the context of 3D animated painting. When a stroke is animated or if the virtual camera moves, the projected 2D stroke can exhibit dramatic deformations in screen space: its shape and curvature can vary over time, and its length can arbitrarily evolve. Currently, only splat based brush models can accommodate these deformations in a temporally coherent way. As a consequence, painterly animation has not benefited from advances in more expressive brush models based on real-world capture. Thus, 3D animated paintings lack the richness of 2D digital painting since example-based stroke styles are off limits in 3D.

Our work addresses this shortcoming with a new method for generating continuously deforming paint strokes. We formalize this problem in the discrete sense: starting from a paint stroke of fixed length, and a library of example strokes painted with the same medium, we precompute shortened and elongated versions of the strokes, respectively by removing stroke sections and adding sections found in the stroke library. At run time, if a stroke is shrunk or stretched, we use a 2D deformation algorithm along with Poisson blending to continuously add or remove pixels from the stroke, while guaranteeing temporal continuity.

Our contributions reside in a method for matching stroke parts to shorten and elongate strokes in a discrete way, and a fast and robust algorithm for continuous deformation of the strokes. To the best of our knowledge, the work we present in this paper is the first method for rendering 3D paintings using a modular, example-based stroke rendering. Since our algorithm is fast, it can also enhance 2D painting interfaces by displaying real-time feedback to the user when drawing strokes, which previous methods could not provide without popping artifacts.

2 RELATED WORK

Being one of the main pillars of non-photorealistic rendering, painterly stylization has been subject to many research publications. While we can refer the reader to [Hertzmann, 2003] or [Hegde et al., 2013] for an extensive study of existing techniques, it is worth noting the differences between different lines of work to position our research. In particular, one can distinguish between automatic and interactive methods. On the one hand, automatic methods like [Litwinowicz, 1997] stylize images and videos in screen space, and research in the linage of [Meier, 1996] aims at stylizing 3D scenes by automatically placing paint strokes on objects and characters. On the other hand, methods presenting interactive applications give more control to artists. Notable works include paint gestures to place multiple strokes for texturing 3D models [Salisbury et al., 1994], or to place individual strokes, mimicking existing 2D painting tools while using 3D objects as canvases, like the seminal WYSIWYG NPR [Kalnins et al., 2002], Deep Canvas [Katanics and Lappas, 2003], or OverCoat[Schmid et al., 2011].

When reviewing the state of the art in painterly rendering, it is worth aiming one's attention towards the paint stroke models that are commonly used. A vast majority turn out to be splat based or to use simple procedural or geometry-based models. Many lightweight brush models were developed for 2D applications, with a focus on computational efficiency, and the frequency of such methods in painterly rendering is no surprise. Notable procedural methods include the popular work by DiVerdi and colleagues for mimicking watercolor paintings [DiVerdi et al., 2012], or more advanced methods that treat the 2D canvas as a mesh and represent paint strokes using triangles [Benjamin et al., 2014].

While such techniques make sense for their specific applications, they mostly produce uniform strokes that exhibit symmetry. As a consequence, painterly rendering techniques that rely on them lack variety. For example, splat models are used in [Meier, 1996], WYSIWYG NPR [Kalnins et al., 2002], Deep Canvas [Katanics and Lappas, 2003], and OverCoat [Bassett et al., 2013, Schmid et al., 2011]. The method of [Litwinowicz, 1997] uses simple 2D capsule geometry around generated paint strokes.

In the context of digital 2D paintings, attempts at more realistic renderings have made use of physical simulation to produce realistic paint strokes. Papers that target elementary tasks like simulating liquid dispersion in paper [Chu and Tai, 2005, Van Laerhoven and Van Reeth, 2005] to those that provide complete frameworks for bristle simulation like Wetbrush [Chen et al., 2015], all make use of a particle simulation to represent paint transfer from the brush to the support as well as an Euler fluid simulation to animate the paint settling on the support. Intermediate methods like IMPaSTo [Baxter et al., 2004] put an emphasis on the final appearance of the paintings, while [Shi and Zhou, 2014] makes use of the GPU to provide artists with real-time feedback when simulating paint.

In our current work, we leverage the power of example-based methods. While such techniques do not make use of complex physical simulations, they can produce a wide range of painting styles. Two research papers were a direct inspiration for our work: [Ando and Tsuruno, 2010] and [Kim and Shin, 2010]. Their strategy is to acquire strokes from physical media and then segment them in pieces. Once a user draws a query stroke, a rendered stroke can be

synthesized by stitching stroke pieces together. A line of followup work has emerged from these advances. In particular, the numerous publications by Lu and colleagues target realistic-looking paintings using example-based methods. In HelpingHand [Lu et al., 2012], they present a method for shape matching between query strokes and acquired examples. Their further developments in RealBrush [Lu et al., 2013] use examples of smeared and smudged paint strokes to broaden the range of styles that their method can produce. Finally, in DecoBrush [Lu et al., 2014a], they present an extension of their example-based framework to pattern strokes. However, by relying heavily on the shape of the query strokes, they have made example-based stroke animation very challenging.

It is also important to note that this line of work is closely related to the area of user-guided texture synthesis where the painting metaphor was first introduced by [Ashikhmin, 2001]. Recent advances by [Lukáč et al., 2015] are able to synthesize single strokes with similar quality to RealBrush. Due to their combined formulation of edge and directional awareness they are able to reproduce the boundary effects typical for natural media. However, all these methods share the same drawback that prevents their direct application to the 3D world: Running their algorithms on the same stroke at different animation keyframes yields very different appearances which cannot be simply mitigated by interpolation.

Hence, continuously animating complex paint strokes is a challenge often mentioned as an open problem in papers cited above. Of the few proposed solutions, the most significant in our opinion is SLAM textures [Bénard et al., 2010], where self-similar textures of various lengths can be synthesized along with continuous animations blending between them. However, SLAM textures need, by definition, to be self similar and tileable, which makes them inappropriate for paint strokes that carry semantic information, such as variations in shape and texture due to the painting direction or pen pressure. Inspired by SLAM textures and taking a fresh start from the methods presented in 2010 [Ando and Tsuruno, 2010, Kim and Shin, 2010], we synthesize example-based strokes using stroke example pieces of known size. We show how to generate a discrete set of strokes of various lengths and how to seamlessly animate between them. We believe to have opened a new direction for stroke animation on which advances similar to those of Lu and colleagues can be built.

3 STROKE ACQUISITION AND PRE-PROCESSING

Our method takes libraries of strokes painted with different physical media as input. The examples shown in this paper were acquired with a consumer flatbed scanner. As is common with existing example-based methods the painted strokes have to be isolated from the support color. However, a fully automatic method is not desirable since speckles of color around a paint stroke can be ambiguous. On the one hand, they can be caused by noise in the acquisition process or grain in the support material, in which case they should be removed. On the other hand, they can also be a side-effect of using a rough brush and should be preserved. We therefore involve the user in this process and ask him/her to perform the clean-up manually with the help of common tools such as the Photoshop Magic Wand. After this step no further user interaction is required.

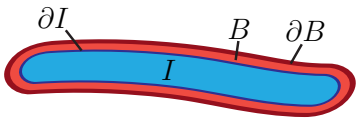
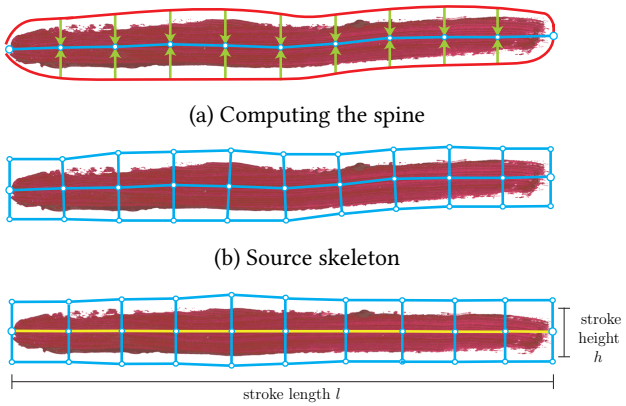


Figure 2: We first distinguish between the interior area I (blue) and boundary area B (red). ∂I and ∂B refer to the boundaries of the respective regions and are 1-pixel wide 8-connected lines in the discrete case.

Kim [Kim and Shin, 2010] or Lu [Lu et al., 2013] use shape information for stroke synthesis and their stroke libraries include examples with various curvatures to improve their results. However, we rely on geometric stroke deformation to handle shape changes continuously over time and cannot benefit from using curvy examples. Therefore it is sufficient for our system to only collect relatively straight example strokes. Additionally, we assume the painting gesture to be from left to right to define the stroke's start and end. This assumption can be enforced by manually rotating the acquired stroke images if needed.



(a) Computing the spine

(b) Source skeleton

(c) Target skeleton

Figure 3: We trace ∂B (red line) from the top and bottom respectively at a regular distance to find points on the stroke spine (blue line)(a). We then cast lines orthogonal to the spine at each spine point and intersect it with ∂B to form a quad mesh around the stroke (b). Finally we map the spine segments and ribs from (b) to a straight spine (yellow line)(c). Performing texture mapping from the source to target skeleton flattens the example stroke.

Since the examples may still contain small curvy parts or be askew, we perform an additional stroke *flattening* step (Figure 3). We start by computing a deformation *skeleton* that covers the stroke. Using the same technique as described in [Jamriška et al., 2015], we first extract the stroke boundary region B and interior region I by applying Gaussian blur to the alpha mask of each stroke image and by thresholding the result. We refer to the boundary of each region as ∂B and ∂I respectively (Figure 2). Choosing a large kernel size (31 pixels in our experiment) results in a smoother contour and adds an

offset around the stroke so that the full stroke is covered. The alpha threshold differs from library to library depending on how vivid the boundary effects are e.g. oil is a medium with highly distinct features around the boundary and requires a higher threshold.

Unlike Kim [Kim and Shin, 2010] or Lu [Lu et al., 2013] we do not require the user to trace the stroke center line (also referred to as *spine*). Instead we automatically extract it by examining the stroke boundary across a vertical scanline from left to right at a fixed interval. We connect the top and bottom pixel of ∂B at the current scanline and define the midpoint as a point on the spine. For each such point, we cast a line orthogonal to the spine and intersect it with ∂B to obtain the *ribs* of the skeleton. Connecting the rib ends and spine points defines a sequence of quads that ensures full coverage of the stroke. Finally, to flatten the stroke we map the spine to a straight polyline of equal length and equal segment lengths. Ribs are mapped in a similar fashion onto the flattened spine and connected to form the *target* skeleton. The deformation is simply a quad-based texture-mapping procedure from the source to the target skeleton. Since we assume relatively straight input strokes, this process does not introduce significant distortions, and our stretching and shrinking problem formulations benefit from working along a straight spine.

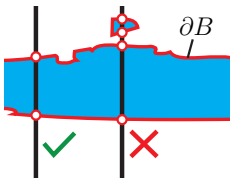
The resulting textures are the *base strokes* of our system and texture hierarchy as described in Section 4. Each base stroke has a length l and height h that are defined by the first stroke pixel encountered from the left and right respectively top and bottom (see Figure 3c). Additionally, we recompute the boundary-interior mask for each base stroke and define ∂B of the mask as the boundary of the base stroke. We have built an interface for the user to choose which base stroke is used to render a certain input stroke as shown in the accompanying video.

4 STROKE CUTS

4.1 Overview

Our method begins by synthesizing a texture for a given query length l_q. We refer to stroke lengthening when $l_q > l$ and to stroke shortening when $l_q < l$. These operations need to be continuous to avoid popping effects during an animation. A naive approach that satisfies this requirement is to scale the base stroke texture to match l_q. However, this method destroys the nature of the texture by stretching its features. To circumvent this issue, we draw inspiration from the artmap methodology, in particular from the self-similar artmaps by [Bénard et al., 2010], and create a continuous hierarchy of textures. Our hierarchy is constructed as following: For stroke stretching we *cut* the base stroke at appropriate places and pull the pieces apart. For each cut we find a suitable image patch of width w_{max} from the stroke library to insert. Our hierarchy consists of textures of length $l + w \cdot n_c$ where $w \in [0, w_{max}]$ and n_c denotes the number of cuts. At each level in the hierarchy we reveal the first w pixel columns of the matching patch at every cut. The inserted partial image patch is processed to ensure continuity at the contour and at the seams (Section 4.4). Note that this algorithm can be applied recursively to synthesize arbitrarily long textures. Stroke shrinking works in a similar fashion but instead of adding patches we gradually remove image patches (Section 4.5).

4.2 Stroke Segmentation



We regard a cut as a vertical line that separates two neighboring columns of pixels. For each example stroke we first compute suitable cut locations. Small speckles or streaks located around the stroke boundary are salient features that contribute to the stroke's character. To avoid cutting through them, we first identify such features using the interior-boundary segmentation technique described in Section 3. By using a small kernel size (3 pixels) we obtain a thin boundary line around every stroke feature. If a column of pixels contains more than 2 boundary pixels it is classified as non-valid for placing a cut next to it (see inset). We also add a horizontal margin (2 pixels in our experiments) to ensure some distance to non-valid cut locations. Lastly, we enforce the minimum distance between two cuts to be w_{max}. Placing cuts too close to each other leads to distortions in the synthesized texture. This heuristic is not needed for very grainy material such as crayon (Figure 4). For such cases we simply cut the stroke at regular intervals.

4.3 Patch Matching

The next step given the cut locations is to determine the image patch to insert. We may assume that strokes within a library already bear a certain level of similarity and calculate a set of candidate patches based on the library. For each example stroke in the library we move a window of width w_{max} from left to right. If both the left and right end of the window are valid cuts (see Section 4.2) we include the image bounded by the window as a candidate and move the window by $w_{max}/2$ to the right. If not we move the window by one pixel to the right until a valid window is found. This ensures that patch candidates do not have separated features.

For each cut, we iterate over all patch candidates to find the most suitable patch to insert. We deliberately use an ambiguous term such as *suitable* as it is difficult to formulate what makes a patch an ideal fit for a certain cut. Different strokes from the same library can exhibit slight differences in color. A color dependent similarity measure will therefore often discard potentially fitting patches because of their difference in luminance or hue. Remaining good matches for a certain cut are often found close to that cut, resulting in obvious repetitions within the same stroke. We decided to broaden the range of good candidates by defining a hue and luminance invariant similarity measure and rely on Poisson blending in the following step to even out color differences (see Section 4.4). The remaining criteria is that the inserted patch must have similar grain to its potential left and right neighborhood. We therefore use the histogram of oriented gradients (HoG) as the similarity metric. First, we deform each candidate as in the patch insertion step (Section 4.4). We then compare the left half of the candidate (c_l) with the right-most $w_{max}/2$ pixels of the left neighborhood (n_l). Respectively, we compare the right half of the candidate (c_r) with the left-most $w_{max}/2$ pixels from the right neighborhood (n_r). The height of the compared region is defined by the maximum spanning rectangle around the left and right neighborhood as well as the candidate as shown in Figure 5. This prevents background pixels

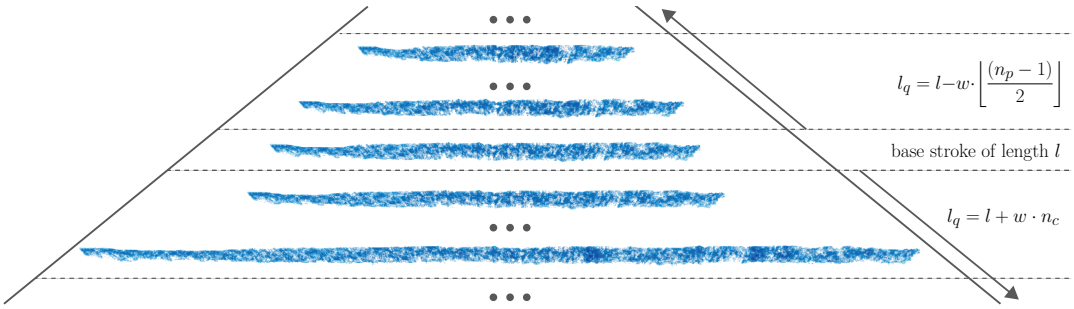


Figure 4: By iteratively removing and inserting columns of pixels to a base paint stroke, we can synthesize a pyramid of arbitrarily long stroke textures. w denotes the width of the patches to be removed (upper half of the pyramid) respectively inserted (lower half of the pyramid). In case the of longer strokes we segment the base stroke with n_c cuts and insert patches at the cuts (Section 4.4). In the case of shorter strokes we segment the base stroke into n_p patches and remove every second one (Section 4.5).

from biasing the score. The final score s of a candidate is calculated as following:

$$s = (\|\text{HoG}(n_l) - \text{HoG}(c_l)\|^2 + \|\text{HoG}(n_r) - \text{HoG}(c_r)\|^2) \cdot \frac{1}{N}$$

The score is normalized by the number of pixels N in the HoG region since the maximum spanning rectangle differs from candidate to candidate. We define the candidate with the lowest score as the insertion patch at the cut in question.

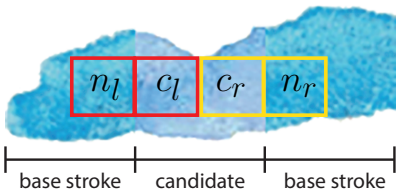


Figure 5: The maximum spanning rectangle in red and yellow is used to compare the histograms of gradient describing the stroke texture.

4.4 Patch Insertion

Given the cuts of a base stroke, their matching patches and the stroke's boundary ∂B we can synthesize a stroke texture of length $l_q > l$ as following: At each cut we wish to insert $w = \lfloor \frac{(l_q - 1)}{n_c} \rfloor$ columns of pixels. We initially take the w first columns of the matching patch as the image patch P to insert. To ensure that the stroke contour remains continuous we deform P to match ∂B at the cut. P itself is enclosed by a quad defined by the boundary of its source base stroke. The deformation is essentially mapping this quad to the rectangle defined by ∂B as shown in Figure 7.

Since we use a color invariant similarity metric, the direct insertion of the deformed P may produce visible seams. We use Poisson blending as first presented in [Pérez et al., 2003] to normalize color while preserving the local variations of the patch. Poisson blending normalizes color intensities from the boundary to the interior of

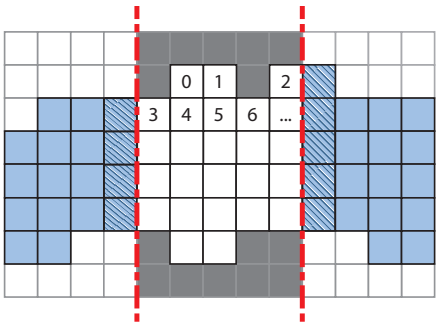


Figure 6: For any inserted patch (white), the hatched pixels are used as boundary conditions in our Poisson color blending.

an image without altering the image gradients by solving the Poisson equation with Dirichlet boundary conditions. In our work, we impose the color intensities of the pixels adjacent to the cut and preserve the gradients inside the inserted patch. We encode neighbor relations in a bidirectional map between pixel positions and mask indices. For example, in Figure 6, neighbors of pixel 1 are pixels 0 and 5. This implementation allows us solve the Poisson equation directly as a linear system, and the factorization of the system can be reused for the three color channels. Our implementation only requires a few milliseconds of computation per patch.

If l_q gradually increases, so does w. As a result, our algorithm gradually inserts columns of pixels at each cut (see Figure 8). This achieves a similar effect as one would by applying Seam Carving [Avidan and Shamir, 2007] to widen an image. However, our method has two major advantages over Seam Carving: first, we simultaneously insert several lines of pixels at consistent locations which is less salient than having one single line added at successively different locations. Second, stretching an image with seam carving is very close to scaling the image and hence shares the drawback of stretching stroke features, as is shown in the accompanying video. Our method, however, preserves the fine textural details of the painting medium.

Our method is able to synthesize coherent textures for arbitrary values of w. If $w = w_{max}$ we insert the complete matching patch in every cut. In the case where $w > w_{max}$ we re-apply the steps from Section 4.2 and Section 4.3 to the texture obtained after inserting the full patches. Our algorithm can therefore be applied iteratively to synthesize textures of arbitrary length (see Figure 4). When iterating our algorithm on stretched strokes, keeping the stroke cuts at the same locations is very likely to insert the same patches at each iteration, which could result in obviously repetitive patterns. We therefore add an offset to the search window defined in Section 4.2 at each iteration, ensuring that the cuts will lie at different locations.

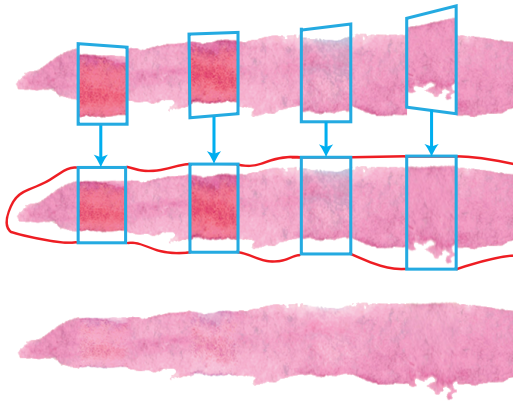


Figure 7: In the top stroke, patches are rigidly inserted. We deform the patches (middle stroke) and apply Poisson blending (bottom stroke) to obtain a consistent stroke shape and color.

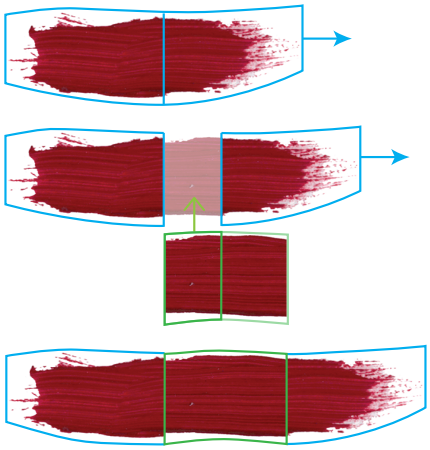


Figure 8: We continuously insert image columns as the base stroke is pulled apart at the cut until we have inserted the full matching patch.

Figure 9: Minimal stroke (right) obtained by recursively removing patches from the original (left).

4.5 Stroke shrinking

At the beginning of a drawing gesture and often in the context of 3D paint rendering, very short strokes have to be rendered. Starting from a base stroke, we therefore extend our hierarchy of paint strokes with textures of length $l_q < l$ by removing patches from the original stroke.

A perceptually motivated approach would be to select patches to remove based on their low saliency or on the similarity to their neighbors. However, playing back such a non-uniform patch removal creates a non-uniform stretching animation, which is not pleasant and incoherent with the stretching animations we designed so far.

Instead, we segment the stroke into regular-sized patches of width w_{max} and simply remove every other patch in the stroke. Say we have n_p patches, this allows us to synthesize textures of length $l - w \cdot \lfloor (n_p - 1) \cdot \frac{1}{2} \rfloor$ where $w \in (0, w_{max}]$ are the first w pixels of the patch to be removed. In other words, we simultaneously remove $\lfloor (n_p - 1) \cdot \frac{1}{2} \rfloor$ columns of pixels per level in the texture hierarchy (see Figure 4). We keep the start and end patch fixed and apply the removal recursively until we reach a minimal length. In our implementation, we stop iterating at 4 patches. Figure 9 shows an example of such a minimal stroke. Similarily to stroke stretching, we deform the patch being removed and apply Poisson blending to maintain shape and color coherence.

5 RENDERING

We have applied our core framework to various use cases: a 2D painting application providing direct visual feedback, an interface to stylize and render 3D paintings, and an application generating replay animations of static 2D paintings. In every case, users can choose the example paint stroke which is used to build a hierarchy of strokes as previously described. To render a 2D painting, or a frame from a 2D or 3D painting animation, we generate a stroke of the required length using the techniques described in Section 4.

Since we perform the stroke synthesis on a straight stroke, the synthesized stroke should be deformed to match the target shape. We embed the straight stroke into a quad strip using quads of fixed length (we use 10 pixels in our results), and resample the target stroke shape to the same resolution. By computing *ribs* on the target stroke, we can deform the synthesized stroke into the scene by texture mapping the quads, as shown in Figure 10.

Finally in the case of 3D paintings, we use the depth information of each stroke point, and use Mixed-Order Compositing [Baran et al., 2011] to coherently handle the depth sorting of rendered stroke fragments. Fragments from strokes at similar depths are sorted in stroke-painting order, while strokes at significantly different depths are sorted in depth order.

To improve performance, we pre-compute the hierarchy of strokes with fully inserted patches, and perform the partial patch insertion as well as the deformation at run time. Generally speaking, the

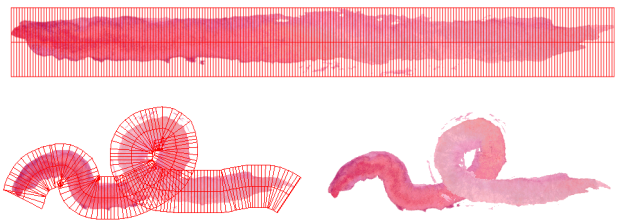


Figure 10: Stroke embedding (top), and deformed into target shape (bottom). Our simple stroke parameterization is also capable of handling self-intersections.

computation of the hierarchy is a system design decision. In our system, the hierarchy is computed on demand, whenever a stroke texture of a certain length needs to be rendered. The bottleneck is the patch matching algorithm which results in a wait time of several seconds depending on the library size. Textures can then be generated at interactive rates (including the post-processing steps) and are cached to speed up rendering of animation frames. It is also possible to pre-compute a number of textures and provide them as input to the system (similar to SLAM textures).

6 RESULTS

We implemented our continuous stroke synthesis system as a library and used it to develop both 2D and 3D painting applications. The accompanying video shows an experienced artist using our 2D application to paint the fish shown in Figure 11 with large gray pencil strokes as well as black calligraphy strokes for the outlines and details. Our implementation provides real-time visual feedback and artists can see each stroke unravel along their painting gesture. More 2D paintings authored with our software are displayed in Figure 12, using various examples of physical media.

Our contributions also target 3D paintings in which paint strokes are embedded around 3D characters and rendered in screen space. Figure 14 shows a 3D animated fish painting rendered using our method in a minimal style, taking inspiration from Japanese *sumi e* paintings. As the fish wiggles, the 3D camera rotates around it, and the paint strokes that constitute the painting dramatically change in length and shape on the screen. The synthesized strokes, however, remain temporally coherent thanks to our algorithm. We used example strokes painted with a marker, oil paint, and watercolor. In a similar fashion, the jellyfish shown in Figure 13 exhibits dramatic screen-space deformations as it is animated and the camera is moving. The depth ordering of the strokes varies during the animation, and our algorithm handles these complex effects coherently.

The rooster in Figure 1 shows a more complex 3D painting rendered using our method. We use the same library of strokes to stylize the whole character and show the diversity of stroke shapes that are natively supported by our method.

7 CONCLUSION AND FUTURE WORK

In this paper, we have extended the range of visual styles accessible to 3D artists by designing a brush model that benefits from the adaptability of example-based stroke synthesis. Our method can

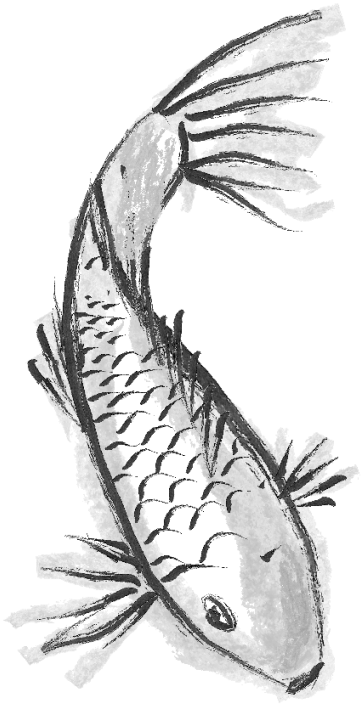


Figure 11: 2D fish painted using crayon and calligraphy strokes.

synthesize a hierarchy of strokes of various lengths and seamlessly transition between different levels of this stroke hierarchy. To the best of our knowledge, this core brush model is the first that produces continuous animations of deforming paint strokes based on scanned examples of physical painting media. Our efficient algorithm supports interactive rates for real-time stroke synthesis in a 2D digital painting application. We demonstrated these contributions in different scenarios by showcasing 2D paintings created by a professional artist using our application and by rendering 3D paintings using novel visual styles.

While these advances already extend the range of possible visual styles in 3D animation, we believe that several directions for future work deserve to be explored. Departing from existing example-based methods for paint stroke synthesis, we have not looked into the animation of more complex paint effects. In particular, no specific attention has been given to realistic color blending as in [Lu et al., 2014b]. Animated color smearing, taking inspiration from [Lu et al., 2013], is another exciting direction of research. For example if a red stroke crosses the path of a blue one during an animation, a continuous smearing of the colors would create the impression that paint is moving in front of the viewer's eyes. One other target would be to incorporate pen pressure in the stroke synthesis. Indeed, professional digital artists are used to mapping the pressure of their pen on their tablet to various stroke parameters, and the artists that used our 2D painting application were excited about the idea of incorporating pressure into the application. Simply scaling

a stroke along its ribs will deform its inner texture and grain, and modulating the width of a synthesized stroke along its length is therefore a challenging topic. Finally, stroked painted with real media exhibit subtle variations based on stroke curvature. Incorporating these variations into our method in a data-driven fashion is another avenue of future work that could further increase the realism and fidelity of our system.

REFERENCES

Ryoichi Ando and Reiji Tsuruno. 2010. Segmental brush synthesis with stroke images. *Proceedings of Eurographics–Short papers* (2010), 89–92.

Michael Ashikhmin. 2001. Synthesizing natural textures. In *Proceedings of the 2001 symposium on Interactive 3D graphics*. ACM, 217–226.

Shai Avidan and Ariel Shamir. 2007. Seam carving for content-aware image resizing. In *ACM Transactions on graphics (TOG)*, Vol. 26. ACM, 10.

Ilya Baran, Johannes Schmid, Thomas Siegrist, Markus Gross, and Robert W. Sumner. 2011. Mixed-order Compositing for 3D Paintings. *ACM Trans. Graph.* 30, 6, Article 132 (Dec. 2011), 6 pages. DOI: https://doi.org/10.1145/2070781.2024166

Katie Bassett, Ilya Baran, Johannes Schmid, Markus Gross, and Robert W. Sumner. 2013. Authoring and Animating Painterly Characters. *ACM Trans. Graph.* 32, 5, Article 156 (Oct. 2013), 12 pages. DOI: https://doi.org/10.1145/2484238

William V. Baxter, Jeremy Wendt, and Ming C. Lin. 2004. IMPaSTo: A realistic, interactive model for paint. In *Proceedings of the International Symposium on Non-Photorealistic Animation and Rendering, NPAR*, Stephen N. Spencer (Ed.). ACM, New York, NY, 45–56. DOI: https://doi.org/10.1145/987657.987665

Pierre Bénard, Forrester Cole, Aleksey Golovinskiy, and Adam Finkelstein. 2010. Self-Similar Texture for Coherent Line Stylization. In *NPAR 2010: Proceedings of the 8th International Symposium on Non-photorealistic Animation and Rendering*.

Mark D. Benjamin, Stephen DiVerdi, and Adam Finkelstein. 2014. Painting with Triangles. In *NPAR 2014, Proceedings of the 12th International Symposium on Non-photorealistic Animation and Rendering*.

Zhili Chen, Byungmoon Kim, Daichi Ito, and Huamin Wang. 2015. Wetbrush: GPU-based 3D painting simulation at the bristle level. *ACM Transactions on Graphics* 34, 6 (2015), 200.

Nelson S-H Chu and Chiew-Lan Tai. 2005. MoXi: real-time ink dispersion in absorbent paper. *ACM Transactions on Graphics (TOG)* 24, 3 (2005), 504–511.

Stephen DiVerdi, Aravind Krishnaswamy, Radomir Mech, and Daichi Ito. 2012. A Lightweight, Procedural, Vector Watercolor Painting Engine. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '12)*. ACM, New York, NY, USA, 63–70. DOI: https://doi.org/10.1145/2159616.2159627

Siddharth Hegde, Christos Gatzidis, and Feng Tian. 2013. Painterly rendering techniques: a state-of-the-art review of current approaches. *Computer Animation and Virtual Worlds* 24, 1 (2013), 43–64. DOI: https://doi.org/10.1002/cav.1435

Aaron Hertzmann. 2003. Tutorial: A Survey of Stroke-Based Rendering. *IEEE Comput. Graph. Appl.* 23, 4 (July 2003), 70–81. DOI: https://doi.org/10.1109/MCG.2003.1210867

Ondřej Jamriška, Jakub Fišer, Paul Asente, Jingwan Lu, Eli Shechtman, and Daniel Sỳkora. 2015. LazyFluids: appearance transfer for fluid animations. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 92.

Robert D. Kalnins, Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Joseph C. Lee, Philip L. Davidson, Matthew Webb, John F. Hughes, and Adam Finkelstein. 2002. WYSIWYG NPR: Drawing Strokes Directly on 3D Models. *ACM Transactions on Graphics (Proc. SIGGRAPH)* 21, 3 (July 2002), 755–762.

G Katanics and T Lappas. 2003. Deep Canvas: integrating 3D painting and painterly rendering. *Theory and Practice of Non-Photorealistic Graphics: Algorithms, Methods, and Production Systems* (2003).

Mikyung Kim and Hyun Joon Shin. 2010. An Example-based Approach to Synthesize Artistic Strokes using Graphs. In *Computer Graphics Forum*, Vol. 29. Wiley Online Library, 2145–2152.

Peter Litwinowicz. 1997. Processing Images and Video for an Impressionist Effect. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 407–414. DOI: https://doi.org/10.1145/258734.258893

Jingwan Lu, Connelly Barnes, Stephen DiVerdi, and Adam Finkelstein. 2013. RealBrush: painting with examples of physical media. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 117.

Jingwan Lu, Connelly Barnes, Connie Wan, Paul Asente, Radomir Mech, and Adam Finkelstein. 2014a. DecoBrush: Drawing Structured Decorative Patterns by Example. In *ACM Transactions on Graphics (Proc. SIGGRAPH)*.

Jingwan Lu, Stephen DiVerdi, Willa Chen, Connelly Barnes, and Adam Finkelstein. 2014b. RealPigment: Paint Compositing by Example. *NPAR 2014, Proceedings of the 12th International Symposium on Non-photorealistic Animation and Rendering* (June 2014).

Jingwan Lu, Fisher Yu, Adam Finkelstein, and Stephen DiVerdi. 2012. HelpingHand: Example-based Stroke Stylization. In *ACM Transactions on Graphics (Proc. SIGGRAPH)*, Vol. 31. 46:1–46:10.

M Lukáč, J Fišer, Paul Asente, Jingwan Lu, Eli Shechtman, and Daniel Sỳkora. 2015. Brushables: Example-based Edge-aware Directional Texture Painting. In *Computer Graphics Forum*, Vol. 34. Wiley Online Library, 257–267.

Barbara J. Meier. 1996. Painterly Rendering for Animation. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*. ACM, New York, NY, USA, 477–484. DOI: https://doi.org/10.1145/237170.237288

Patrick Pérez, Michel Gangnet, and Andrew Blake. 2003. Poisson image editing. In *ACM Transactions on Graphics (TOG)*, Vol. 22. ACM, 313–318.

Michael P Salisbury, Sean E Anderson, Ronen Barzel, and David H Salesin. 1994. Interactive pen-and-ink illustration. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. ACM, 101–108.

Johannes Schmid, Martin Sebastian Senn, Markus Gross, and Robert W. Sumner. 2011. OverCoat: an implicit canvas for 3D painting. *ACM Trans. Graph.* 30, Article 28 (August 2011), 10 pages. Issue 4. DOI: https://doi.org/10.1145/2010324.1964923

Liqiang Shi and Shizhe Zhou. 2014. Real-time dynamic and pressure-sensitive brush rendering. (2014), 1093–1098.

Tom Van Laerhoven and Frank Van Reeth. 2005. Real-time simulation of watery paint. *Computer Animation and Virtual Worlds* 16, 3/4 (2005), 429.

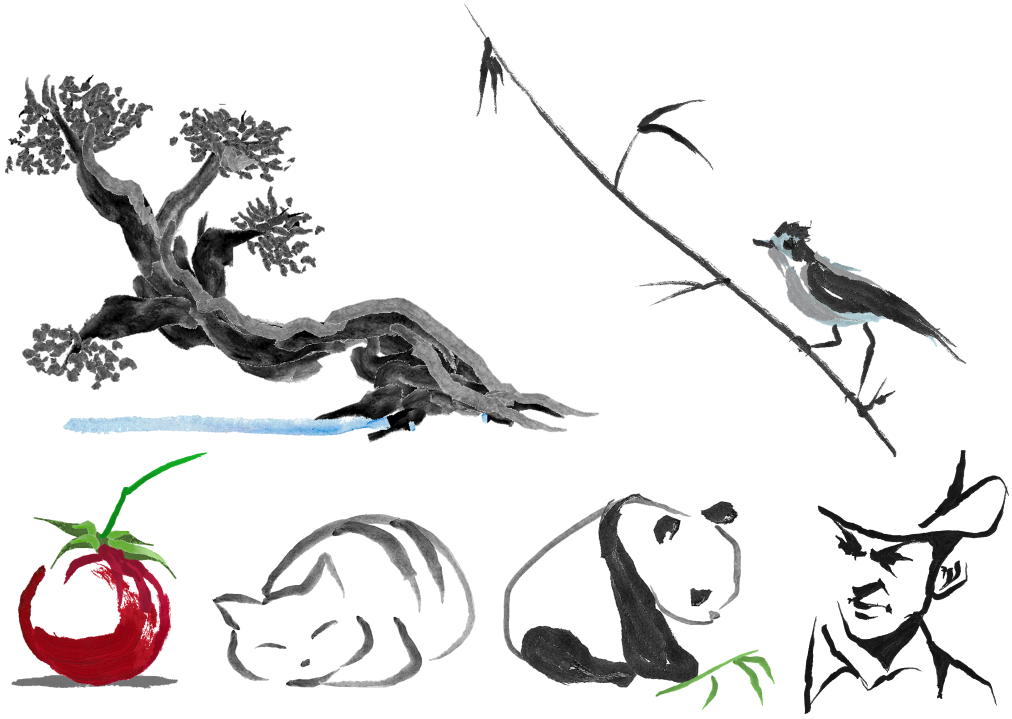


Figure 12: Digital paintings created with our 2D painting application.

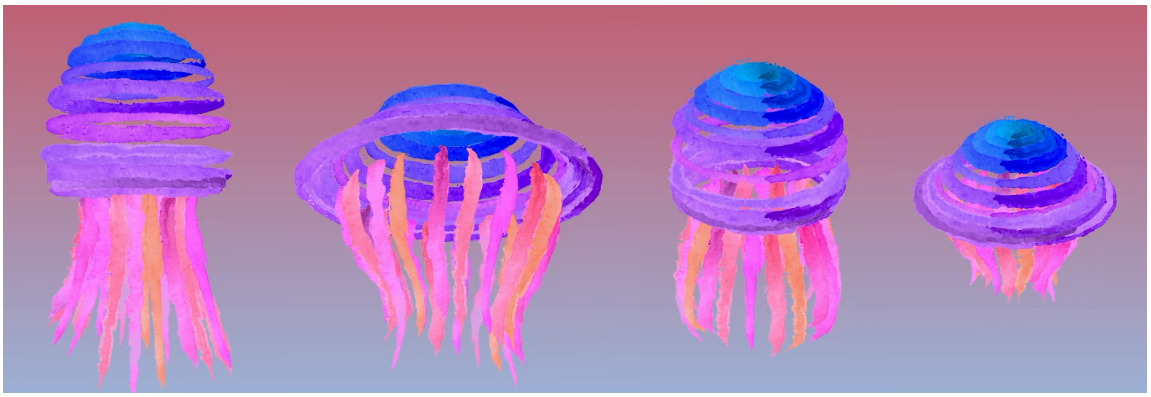


Figure 13: Jellyfish animation rendered using watercolor strokes on rough paper. During animation the stroke order of the jellyfish rings will differ since they are close in depth. We use mixed-order compositing [Baran et al., 2011] to guarantee smooth stroke order transitions.

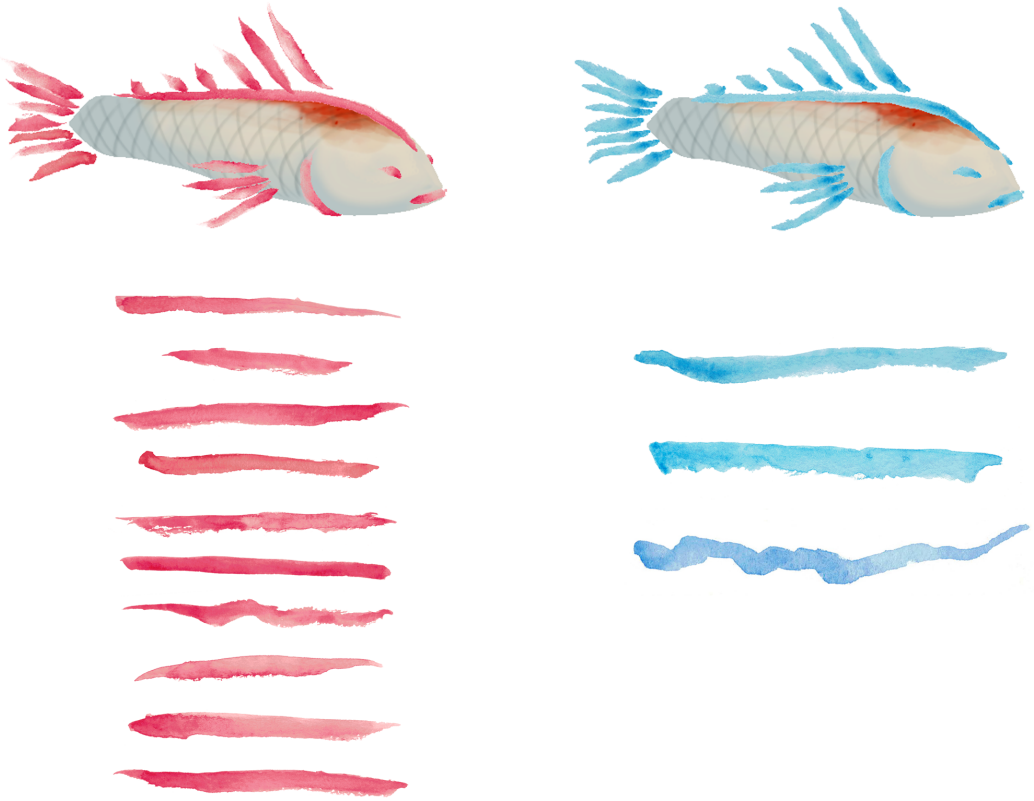


Figure 14: Animated 3D fish (top) stylized using two different watercolor libraries (bottom).