Ciallo: GPU-Accelerated Rendering of Vector Brush Strokes

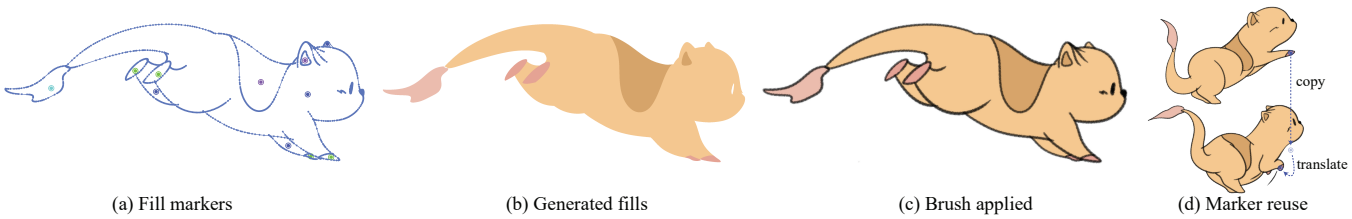
Shen Ciao
Zhongyue Guan
Qianxi Liu
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China

Li-Yi Wei
Adobe Research
San Jose, USA

Zeyu Wang
The Hong Kong University of Science and Technology (Guangzhou)
Guangzhou, China
The Hong Kong University of Science and Technology
Hong Kong, China

(a) Raster brush (b) Vector curves (c) Our integration (d) Editable strokes

Figure 1: Our integration of raster and vector brush strokes. We take a brush that was previously only available for producing raster drawings (a), render it on a vector curve (b) with GPU acceleration, and obtain the final integrated drawing (c). (d) Artists can still freely edit the stroke shape as if it were a regular vector stroke, or use another raster brush to replace the current one.



(a) Fill markers (b) Generated fills (c) Brush applied (d) Marker reuse

Figure 2: Painting system with vector fill. We develop a prototype painting system with a vector coloring process for drawn frames in an animation. The dotted circles ⊙ in our vector drawing (a) are called fill markers that record where to fill color. Our method generates filled regions (b) accordingly and fuses them with outline strokes (c). Users can copy markers between frames and manipulate them based on new outlines (d). The fill regions are updated automatically.

ABSTRACT

This paper introduces novel GPU-based rendering techniques for digital painting and animation to bridge the gap between raster and vector stroke representations. We propose efficient rendering methods for vanilla, stamp, and airbrush strokes that integrate the expressiveness of raster-based textures with the ease of real-time editing. Based on our stroke representation, we implement an open-source prototype drawing system with a vector fill feature, demonstrating that our techniques can enhance the expressiveness, efficiency, and edibility of digital drawing. Our work can serve as a foundation for future research on vector-based and GPU-accelerated rendering techniques in industrial-level brush engines.

SIGGRAPH Conference Papers '24, July 27-August 1, 2024, Denver, CO, USA

ACM ISBN 979-8-4007-0525-0/24/07…$15.00
https://doi.org/10.1145/3641519.3657418

CCS CONCEPTS

• **Computing methodologies** → **Non-photorealistic rendering**; • **Human-centered computing** → **Interactive systems and tools**.

KEYWORDS

digital painting, vector graphics, brush stroke rendering

ACM Reference Format:
Shen Ciao, Zhongyue Guan, Qianxi Liu, Li-Yi Wei, and Zeyu Wang. 2024. Ciallo: GPU-Accelerated Rendering of Vector Brush Strokes. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Papers '24 (SIGGRAPH Conference Papers '24), July 27-August 1, 2024, Denver, CO, USA*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3641519.3657418

1 INTRODUCTION

Digital painting technology has been approaching the quality and realism that traditional painting can achieve while providing more flexibility and convenience, such as layers and compositing, undo and redo [Myers et al. 2015], novel material and mixing properties [Shugrina et al. 2022; Sochorová and Jamriška 2021], and automation and assistance [Jacobs et al. 2018; Xing et al. 2014]. Brush stroke rendering is fundamental in digital painting and is critical for supporting the desired painting process and result.

Contemporary digital painting applications represent drawings using either vector or raster graphics. Each representation offers complementary advantages and disadvantages. For instance, vector graphics provide easier manipulation, whereas raster graphics provide higher expressiveness. However, artists desire the benefits from both representations—manipulating strokes and filled regions in real time while employing the raster-based brush textures.

Many painting programs have attempted to integrate these benefits. For example, the vector graphics program Adobe Illustrator provides versatile brushes, and the raster graphics program Clip Studio Paint develops vector layers that record brush strokes with vector curves. However, their integration is unsatisfying due to technical limitations. Adobe users have long complained about the inability to use Photoshop's brushes in Illustrator [Adobe Community 2009]. Clip Studio Paint supports neither real-time rendering of strokes nor filling colors on vector layers.

To address these challenges, we develop vector stroke rendering techniques that can facilitate high artistic expressiveness at a GPU-accelerated rendering speed. Inspired by the polyline representation in Blender Grease Pencil, our techniques enjoy the benefits of both vector and raster representations, including the ease of manipulating stroke geometry and the expressiveness of raster-based brush texture. Moreover, we develop a prototype painting system called Ciallo with a *Vector Fill* feature to facilitate vector-based color filling with the planar map technique [Asente et al. 2007]. When artists fill colors in their drawings, we allow them to freely manipulate the position or color to fill a region enclosed by our brush strokes.

We have open-sourced our prototype digital painting system at https://github.com/ShenCiao/Ciallo. Our stroke representation and rendering techniques are easy to implement and can be integrated into existing digital painting software and potentially other applications such as game engines and visualizations. Compared to existing brush renderers without GPU acceleration, our work enables expressive and efficient vector graphics for digital drawing and various applications.

In summary, our paper makes the following contributions:

- Expressive brush stroke rendering techniques for vanilla, stamp-based brushes, and airbrushes using a hybrid vector-raster representation and GPU acceleration.
- An open-source prototype painting system incorporating our brush stroke rendering techniques, enabling expressive stylization and interactive editing.

2 RELATED WORK

Stroke rendering and region filling are two of the cornerstones in digital painting programs [Warnock and Wyatt 1982]. Below we review the relevant features and techniques in contemporary digital painting programs and research papers.

2.1 Stroke Rendering

We select representative programs and group them into four categories based on stroke rendering techniques, as shown in Table 1.

Table 1: Stroke rendering techniques used in representative digital drawing programs. Vector: whether a program renders vector strokes. GPU: whether a program uses GPU-accelerated stroke rendering. Raster brush: whether a program renders raster brush strokes. If a program supports raster brushes, the Vector column will not be checked if it has a vector representation that cannot be used together with raster brushes, e.g., Krita and Adobe Fresco.

Sec.	Program	Vector	GPU	Raster brush	Open-source
2.1.1	Inkscape	✓	–[1]	–	✓
	Synfig Studio	✓	–	–	✓
	Adobe Illustrator	✓	✓	–	–
	Affinity Designer	✓	✓	–	–
2.1.2	Adobe Photoshop	–	–	✓	–
	Adobe Fresco	–	–	✓	–
	Krita	–	–	✓	✓
	Clip Studio Paint	✓	–	✓	–
	OpenToonz	✓	–	✓	✓
2.1.3	Corel Painter	–	✓	✓	–
	BlackInk	–	✓	✓	–
2.1.4	Disney Meander	✓	✓	✓	–
	Ours	✓	✓	✓	✓

[1] Inkscape has an experimental OpenGL renderer, which is not officially released.

2.1.1 Standard Vector Strokes. Many programs, such as Adobe Illustrator, Inkscape, Affinity Designer, and CorelDRAW, focus on vector graphics using the SVG standard [W3C 2001]. We refer to strokes defined in SVG as "standard vector strokes." Previous research has focused on accelerating SVG rendering [Dokter et al. 2019; Kilgard 2020; Kilgard and Bolz 2012]. However, SVG does not support variable width and texture overlap, which are crucial properties in digital painting.

Figure 3 shows a drawing rendered with and without variable width. The one with variable width has a significantly expressive appearance. All the aforementioned programs support variable width but they have to convert strokes to filled outlines in SVG format.

Although there was a proposal to support variable width [W3C 2014], it was not accepted. The research on GPU-accelerated vector stroke rendering [Kilgard 2020; Kilgard and Bolz 2012] did not consider the variable width either.

Figure 3: Expressive strokes with variable width. Variable width on a single stroke is critical for drawing professional illustrations. The butterfly drawing with variable width has a significantly improved appearance.

When mapping textures onto a stroke, standard vector strokes commonly use the "skeletal stroke" model [Hsu and Lee 1994; Hsu et al. 1993]. Its modern GPU implementations [Kilgard 2020; Kilgard and Bolz 2012] tessellate a stroke and map deformed textures onto individual segments. However, the textures do not overlap and deform significantly along a stroke, which artists do not prefer. Our strokes prevent texture deformation and effectively manage texture overlap and blending.

2.1.2 Stamp Strokes. Two programs, Clip Studio Paint and OpenToonz, offer users a "vector layer" system. This system records strokes using vector curves and renders them on raster images using CPUs. There has not been much research on this process, mainly because the classical brush rendering with CPUs is well-established and straightforward. Applying this rendering method to vector paths is straightforward for experienced engineers without considering the performance. These programs barely provide a real-time experience when manipulating a single stroke.

Two classical stroke rendering approaches are sweep [Whitted 1983] and stamp [Haeberli 1990]. In the sweep model of stroking, the geometric shape of the pen sweeps out a region such that all the pixels within that region are considered to belong to the stroke. In the stamp model, a brush pattern (typically modeled with a raster image) follows the stroke's trajectory and stamps out the brush pattern at regular intervals. In practice, the stamp model is the core of most digital raster brush engines. More than 90% of brushes can be categorized as stamp brushes in Photoshop, and the art community has created thousands of brushes with Photoshop's brush engine. Stamp strokes are dominant in 2D production.

However, there is a lack of techniques to render large-scale stamp strokes in real time, due to the potential heavy computation and memory access incurred by a moving brush stroke [Fishkin and Barsky 1984]. The real-time experience is especially essential to enable easy manipulation of a vector representation. In contrast, real-time 3D rendering with GPUs has been common for almost two decades. Engineers and artists have long anticipated 2D GPU rendering techniques. Therefore, our research focuses on rendering stamp brushes with a GPU, aiming to meet the requirements for fast rendering speed.

2.1.3 GPU-Accelerated Stamp Strokes. Two commercial paint programs, Corel Painter and BlackInk, explicitly advertise using GPUs to accelerate stroke rendering. However, they only support creating raster images. For academic research, Joshi [2018] attempted to render stamp brush strokes but with a critical limitation—brush's footprints are constrained to a continuous isotropic function. Our approach aims to fill this gap.

There has been other research on GPU-accelerated brushes, but these techniques may not apply to vector graphics, such as watercolor [DiVerdi et al. 2012], oil [Chen et al. 2015], and ink [Chu and Tai 2005]. Some of them involve time-varying simulations, so manipulating curves after they are drawn is not practical.

2.1.4 Integration. Our program and Disney's Meander [Whited et al. 2012] combine all three advantages, rendering vector brush strokes with GPU acceleration. However, Disney did not disclose the technical details of Meander, so digital paint content users, such as game engines and GUI, cannot benefit from it.

Our work is inspired by the polyline representation used in OverCoat [Schmid et al. 2011] and Blender Grease Pencil. Polylines have many benefits. They can approximate curves and support variable width by storing a width value at each vertex, and Grease Pencil has developed many useful tools for polyline editing. However, OverCoat and Blender Grease Pencil render brush strokes by placing a footprint at each vertex. This method has a critical limitation: strokes have an inconsistent appearance with different vertex density, as Figure 4 shows. Our method places stamps equidistantly along a polyline to address this issue.

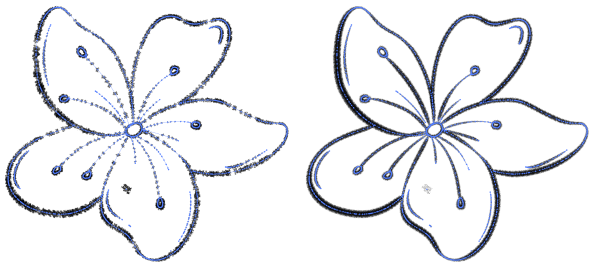


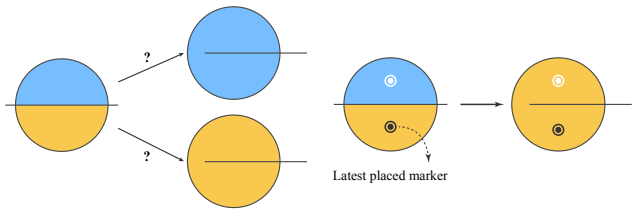
Figure 4: Stroke rendering compared to Blender Grease Pencil (GP). GP and OverCoat render a polyline by placing a footprint at each vertex (left). Therefore, the strokes can have an inconsistent and discontinuous appearance due to the uneven vertex distribution. Our method (right) places footprints equidistantly along the polylines resulting in a consistent and continuous stroke appearance.

2.2 Painting Systems

Besides stroke rendering, color filling is another fundamental element in paint programs. To fill color into our drawings represented in polylines, we use the planar map technique introduced by Baudelaire and Gangnet [1989] and Asente et al. [2007], and address two issues identified by Asente et al. [2007].

The Live Paint [Asente et al. 2007] in Adobe Illustrator addressed the edibility problems in the naive planar map by utilizing planar graph matching algorithms to match the fills and strokes in the planar maps before and after editing. However, Asente et al. [2007] highlighted their challenges, "the problem of fill and stroke assignment (matching) is inherently ill-posed" and "the major performance bottleneck with Live Paint is reconstruction of the planar map and assigning fills and strokes to all regions and edges."

We address these two issues by introducing the *Fill Marker*, which stores the positions and colors to fill in. The markers are rendered with dotted circles ⊙ in the order of users' original drawing. Each marker locates a fill from the planar map, and fills are rendered in alignment with the markers' order. Therefore, the positions and orders to fill are determined, as Figure 5 shows. Since we do not need to assign fills and strokes, the system's performance is improved.



(a) The ill-posed case (b) Our solution

Figure 5: An ill-posed case and our solution. (a) shows the ill-posed case demonstrated in [Asente et al. 2007]. We address it with fill markers in (b) by filling each region in the order of enclosed markers following the painter's algorithm. In this example, the circle is filled twice after the line segment is moved to the right. Therefore, the newly filled color should fully overlap the old one, like bucket fill twice into the same region in Photoshop.

We call the filling system "vector" fill for two reasons. First, users can easily translate or copy fills by clicking and dragging markers on the canvas. These vector-style manipulations are very intuitive to artists. Second, the planar map only works with the vector representation instead of the raster representation. Moreover, the vector fill can facilitate artists' color animations by copying and translating color markers between frames, as demonstrated in Section 5 and the supplementary video.

Translating markers to manipulate where to color has been extensively explored in research related to diffusion curves [Orzan et al. 2008], particularly in the system by Finch et al. [2011]. However, they emphasized smooth-shaded coloring and did not fully realize its potential in animation production. In animation, frames exhibit a high level of correlation with one another. Utilizing this intrinsic relationship and reusing coloring positions can significantly accelerate the flat shading process.

As for other work closer to our interest, vector graphics complexes [Dalstein et al. 2014, 2015] bring novel data structures for vector graphics and animations. However, they are relatively difficult for engineers to implement and require artists to expend additional effort to learn a new approach. Sýkora et al. [2011] focus on texture mapping on filled regions. Other methods handle contour gaps [Parakkat et al. 2022; Sýkora et al. 2009; Yin et al. 2022] or color drawings automatically with AI [Yan et al. 2022; Zhang et al. 2021, 2018].

3 RENDERING ALGORITHMS

We develop GPU-based rendering algorithms for three types of strokes: vanilla, stamp, and airbrush. Vanilla strokes have a solid color with varied widths that are the foundation for the other two types of strokes. Stamp strokes use a stamp texture equidistantly along the drawing trace, giving a more expressive appearance. Airbrush strokes have a transparent gradient from the middle axis to the rim, suitable for creating shading effects. Below we describe the detailed derivations of our GPU-based stroke rendering techniques for these types of strokes.

All three types of strokes use the same vertex placing method. We parameterize the stroke based on its length and place a vertex equidistantly along the stroke based on a given interval. These vertices $V = \{v_0, v_1, \ldots\}$ are connected consecutively by edges, forming a polyline. Then, we push the position p_i and radius r_i of each vertex v_i on the polyline into vertex buffers before rendering. This is the only time of data synchronization between CPU and GPU memory. To render a drawing, we loop through all the strokes and issue a draw call for each stroke. The order of draw calls follows the original stroke order. Finally, we enable stroke blending and set a default alpha blending function in the graphics pipeline.

3.1 Vanilla

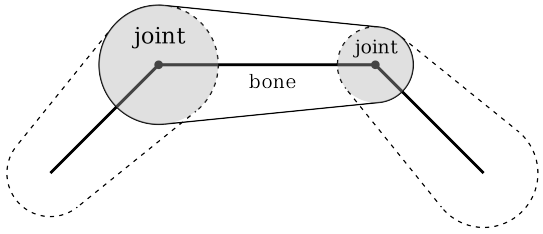
Our rendering process of a stroke is analogous to manufacturing an articulated arm, so we call it the *articulated* method. The rendering of stamp and airbrush strokes also share the same idea. Given an edge between two vertices v_0 and v_1, we create two disks centered at the two vertices based on their positions and radii. We call the disks *joint areas*. The area not covered by the joints is called the *bone area*, as shown in Figure 6a.

To render this edge in a vanilla stroke, we start by creating a trapezoid around the two vertices based on their radii. Pixels inside the trapezoid are the invoked pixels by this edge. The four sides of the trapezoid are tangent to the two joint circles. Then, we calculate the trigonometric value of θ, $\cos\theta = (r_0 - r_1)/||p_0 - p_1||$, and the four corners of the trapezoid, as shown in Figure 6b. Next, we discard the pixels in the four corners of the trapezoid to generate a capsule. After blending, all capsules connect to each other naturally, as shown in Figure 6a.

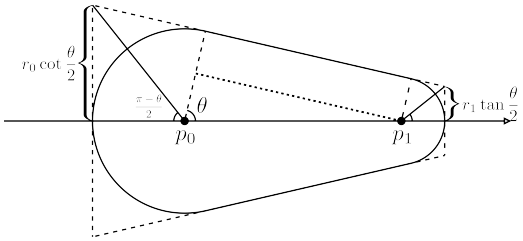
We used both the geometry shader and instanced rendering to render vanilla strokes, where the geometry shader is for our desktop program on Windows and instanced rendering for the Web.

3.2 Stamp

Stamp strokes are widely used in painting software, where diverse stamp textures or footprints offer a high degree of expressiveness. To find where to place footprints, a typical CPU program starts from the first point of the polyline, moves along it with a fixed interval, and places a footprint at each step. To parallelize this process, we compute the distance from each vertex to the first vertex along the polyline. A compute shader is used to calculate the prefix sum of edge length in parallel. By passing the values into the fragment



(a) Edge connection.



(b) Parameters to place the trapezoid.

Figure 6: Vanilla stroke rendering. We use an *articulated* method to render an arm shape on each edge (b) and connect them to each other (a).

shader, we calculate the stamp positions on an edge so each invoked pixel can sample the stamp texture to determine its color.

For better performance, an invoked pixel should only sample stamps that can cover it, rather than looping through all stamps. For each invoked pixel $\boldsymbol{p}$, we find a segment on the edge, so only stamps inside this segment can cover the pixel. The two circles center at the start and end points of the segment, e.g., s_0 and s_1, and they intersect at the invoked pixel $\boldsymbol{p}$, as shown in Figure 7. If there is a footprint at s_0 or s_1, the footprint just touches pixel $\boldsymbol{p}$.

We use a coordinate system originating at p_0, and the X and Y axes align to the tangent and normal direction. Denote the position of $\boldsymbol{p}$ as (x, y). To derive s_0 and s_1, we represent the radius at s_i as a function of s_i, i.e., $\tilde{r}(s_i)$. The circle centered at s_i touches the invoked pixel $\boldsymbol{p}$, so there are two identical equations $\tilde{r}(s_0)^2 = (x - s_0)^2 + y^2$ and $\tilde{r}(s_1)^2 = (s_1 - x)^2 + y^2$.

We also observe the relationship between $\tilde{r}(s_i)$ and two radii r_0 and r_1 via similar triangles:

$$\tilde{r}(s_i) = (1 - s_i/L)r_0 + (s_i/L)r_1,$$

where $L = ||p_0 - p_1||$. After replacing $\tilde{r}(s_i)$, we get a single quadratic equation whose two roots are s_0 and s_1, so we can determine all stamps that affect the color of each invoked pixel $\boldsymbol{p}$.

Overall, we separate the stroke rendering into two stages. The first happens in the rasterization stage defined by the fragment shader. The second happens at the blending stage defined by the pipeline's blending function. We compute pixel colors for each edge in parallel in the first stage, then articulate the edges together in the second stage. The two stages follow the same blending algorithm to render the stamp stroke correctly.

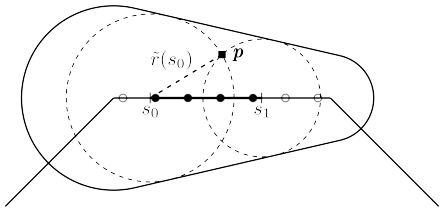


Figure 7: Segment of stamps covering a pixel. Stamp positions that can cover the pixel are marked with solid circles, otherwise hollow circles. The start and end points of the segment are marked with vertical ticks.

3.3 Airbrush

Airbrush is a special type of stamp brush whose footprint is a dot with transparency or a transparent gradient from its center to rim, as shown in Figure 8. When the footprints are close enough, they form a transparent gradient from the middle axis to the rim of the airbrush stroke. The blurring effect enabled by its gradient property makes airbrush strokes suitable for coloring and shading in digital drawing, while the other strokes are typically used for outlines.

Figure 8: Footprints and their corresponding airbrushes rendered with the method in Section 3.2.

A desirable airbrush appearance requires footprints that are very close to each other, causing each pixel to sample many footprints. For better performance, we derive a mathematically continuous representation of airbrush to avoid excessive sampling when the stamp interval is infinitely small.

Consider a continuous, effectively infinite, sequence of stamps on an edge whose length is L. Denote the number of stamps as n, and the interval between stamps as $\Delta L = L/n$. For each pixel at position $\boldsymbol{p}$ invoked by the edge, its alpha value $A(\boldsymbol{p})$ is equal to all alpha values $A_s(\boldsymbol{d}_i)$ for $i = 1...n$, blended from all stamps on the edge:

$$A(\boldsymbol{p}) = 1 - \prod_{i=1}^{n} (1 - A_s(\boldsymbol{d}_i)),$$

where $\boldsymbol{d}_i$ is the vector from stamp i and the current pixel.

Denote the *alpha density* value as α. Let $A_s(\boldsymbol{d}_i) = \alpha_s(\boldsymbol{d}_i)\Delta L$, $\alpha_s(\boldsymbol{d}_i)$ is called an alpha density field and is defined by the footprint. The notations refer to probability density and probability values. Replace $A_s(\boldsymbol{d}_i)$ and get:

$$A(\boldsymbol{p}) = 1 - \prod_{i=1}^{n} (1 - \alpha_s(\boldsymbol{d}_i)\Delta L).$$

Therefore, given any function $\alpha_s(\boldsymbol{d}_i)$ representing a footprint, calculate a continuous form of the stamp stroke by substituting this function into the formula.

In the old coordinate system originating at p_0, the X and Y axes align to the tangent and normal direction. Assume $\boldsymbol{p} = (x, y)$ and $\boldsymbol{d}_i = (x - l_i, y)$ in the coordinate system, where l_i is the X position

of stamp i. As $n \to \infty$ and $\Delta L \to 0$, apply the Volterra product integral to the formula and get:

$$A(x,y) = 1 - \exp\left(-\int_0^L \alpha_s(x-l,y)\mathrm{d}l\right).$$

Consider a constant alpha density, indicating that the footprint is a consistently transparent dot, defined by the function:

$$\alpha_s(d) = \begin{cases} \alpha_c & ||\boldsymbol{d}_i|| \le R \\ 0 & ||\boldsymbol{d}_i|| > R \end{cases},$$

where R is the radius of the footprint.

Substitute $\alpha_s(d)$ into $A(x,y)$ and get:

$$A(x,y) = 1 - \exp\left(-\alpha_c L_r\right),$$

where L_r is the segment length that can cover the current pixel, i.e., the distance between s_0 and s_1 in Figure 7. The method may also render other variants of continuous airbrushes given a different alpha density field, but the mathematical derivation becomes significantly more complex than the constant alpha density.

4 PAINTING SYSTEM IMPLEMENTATION

We developed a prototype painting program with vector fill to showcase our stroke rendering technique. For rendering, our implementation uses C++, OpenGL, and Dear ImGui, as shown in Figure 9. Our user interface includes a paint tool to draw on the canvas, an edit tool to transform or deform strokes, and a timeline feature to create animations. The user chooses a brush type and specifies a few brush parameters, including stamp texture (footprint), stamp interval, rotation, and fractal noise. Then, the user can draw strokes on a canvas with the specified brush. The drawn strokes are stored as polylines and re-rendered at every single frame. After the user finishes drawing a stroke, they can use the edit tool to manipulate the polyline while keeping the original stroke appearance.

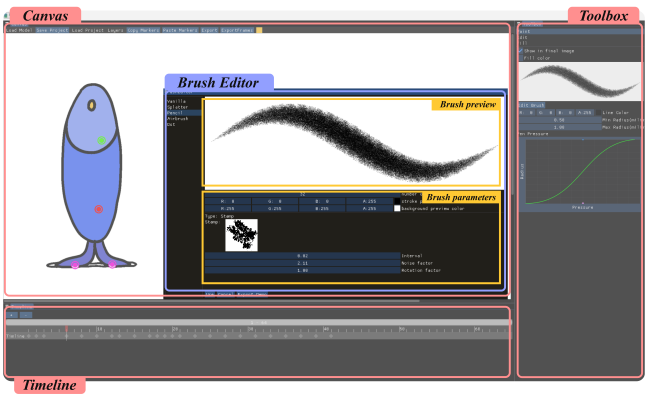


Figure 9: User interface of our prototype system. It includes a canvas, a timeline for animation, and a toolbox. The toolbox includes *paint* for tuning brush and stroke parameters, *edit* for editing strokes, and *fill* to create fill markers.

We use the CGAL 2D Arrangement library [Wein et al. 2007] to construct planar maps and use the `Arr_walk_along_line_point_location` strategy for face query with fill markers. After obtaining the faces, the program renders the corresponding polygon underneath the strokes using the "stencil, then cover" method [Kilgard and Bolz 2012]. Different from Asente et al. [2007], the program only constructs a single planar map for each drawing. When the user transforms a stroke, the program does not need to reconstruct the planar map from scratch. Instead, it updates the modified stroke with two CGAL functions, `remove_curve` and `insert`.

5 RESULTS

We demonstrate the effectiveness and versatility of our stroke rendering techniques with a variety of artistic drawings and animations, and stylized rendering of existing drawing datasets. We also report the computational performance of our method.

5.1 Drawing and Animation

To understand the expressiveness of our techniques, we invited professional artists to create illustrations or animations using our prototype system.

5.1.1 Illustration. Figure 10 shows an illustration produced with our program. The illustration includes all three brush types: 1) vanilla brush for the lines in the background, 2) stamp brush (pencil) for the character, and 3) airbrush for coloring the sky and face. The user found our canvas very responsive during the drawing process, which benefits from our fast stroke rendering speed.

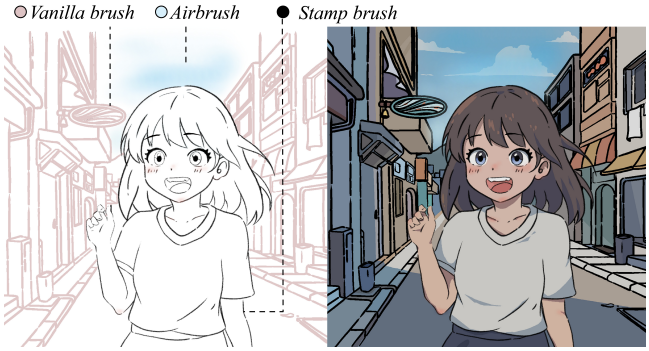


Figure 10: An illustration with various brushes rendered using our techniques. The left shows all the brush strokes including airbrushes used for coloring. The right shows the final illustration with color fills.

5.1.2 Animation. To demonstrate the vector fill feature, fill marker reuse, we create two animations, as shown in Figure **??**. Users can copy fill markers from one frame and paste them onto another frame, so they can easily adjust the fill markers to create filled regions rather than filling from scratch. Our supplementary video shows the screen recording of the entire coloring process. Reusing color markers between frames can significantly save the effort of coloring an animation. We invited three professional animators to test our prototype system and they all appreciated this feature and desired it to be available in commercial animation software.

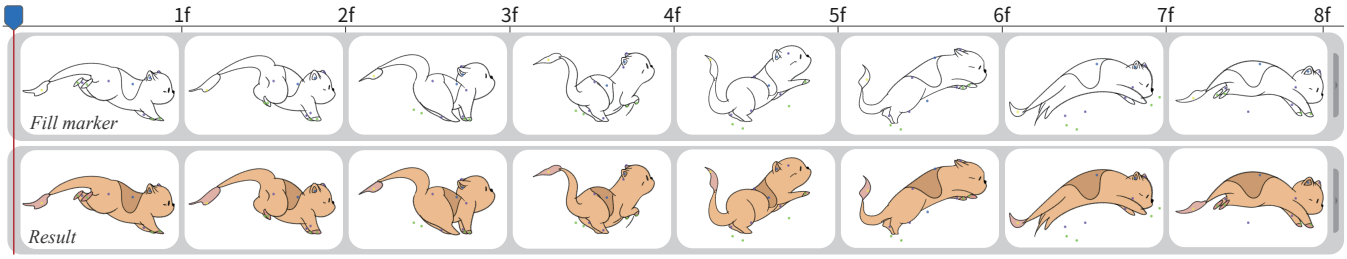


Figure 11: Fill markers in an animation sequence. Users can copy the fill markers from one frame to another and easily manipulate them to achieve a desired color filling result.

5.2 Rendering Drawing Datasets

Our brush rendering technique enables stylized rendering of drawing datasets with a variety of brush styles that users can flexibly specify and adjust in real time. We rendered drawings from the OpenSketch [Gryaditskaya et al. 2019], SpeedTracer [Wang et al. 2021], and DifferSketching [Xiao et al. 2022] datasets, which further demonstrate the quality and expressiveness of our approach. These datasets have recorded a pen pressure value for each vertex on the polylines, implying the varying width of stroke paths. To achieve visual aesthetics, we used simple linear and quadratic functions to map the pen pressure to stroke thickness and opacity. The rendering results are shown in Figure 15. More stylized rendering results can be found in Figure 16 and 17.

5.3 Performance

While 24 FPS already provides a decent drawing experience [Braga Sangiorgi et al. 2012], a higher refresh rate can facilitate game rendering and players' experience [Kim et al. 2019]. Therefore, we designed a stress performance test for our prototype system with the GUI functions removed. The results are shown in Figure 18.

We only used stamp brushes for testing since the three types of brushes only differ in the fragment shader, and stamp brushes require the highest computational complexity. We manipulate two main factors that affect the performance:

Stamp interval. The parameter in stamp brushes, stamp interval, can significantly affect the performance. It controls how many times a pixel should sample footprints. We tested brushes in which a pixel samples footprints 5 to 50 times at maximum.

Number of pixels. The number of pixels covered by strokes determines how many times the fragment shader is executed. Several determining rendering parameters include stroke width, canvas size, and DPI. In practice, outline strokes cover a relatively small area of the whole canvas, making stroke rendering efficient. We deliberately use larger stroke widths to create the pressure test for our stroke rendering algorithm.

As Figure 18 shows, our prototype system can render strokes very efficiently using a consumer-grade GPU. It renders the drawing at 700 FPS under the common setting using a canvas resolution of 2560×1440 and a maximum stroke width of 12.0 pixels. This performance does not even leverage the GPU's full capacity as it is bottlenecked by the CPU. Even under the drastic setting with a canvas resolution of 3840×2160 and a maximum stroke width of 38.6 pixels, our system still runs at 75 FPS. This shows the efficiency of our GPU-based stroke rendering techniques for digital drawing in practice while providing expressive and editable vector graphics.

Moreover, a potential "naive" stamp approach is to calculate all the stamp positions on a CPU, then rasterize each stamp footprint separately on a GPU. This naive approach is a straightforward variation of Blender Grease Pencil and Overcoat's method. We implemented this approach that calculates the stamp positions and radii at each frame and then converts the position points into texture squares via a geometry shader. Experimental results indicate that our method outperforms the naive method but performance gains vary significantly depending on different drawings and parameters. The results and implementation details are shown in Figure 19.

6 CONCLUSION

In this paper, we presented GPU-accelerated brush stroke rendering techniques and a prototype system with vector fill to facilitate more expressive and efficient vector graphics. For demonstration and evaluation, we have produced a variety of examples including illustrations, animations, and stylized renderings of drawing datasets. The results indicate that our techniques can support digital drawing by providing expressive and editable vector graphics at an interactive speed.

Our work has a few limitations. A critical property of vector art is zooming in without losing details. While our vanilla and airbrush strokes support this, stamp brushes use raster footprints that limit the zoom-in capability. In the future, we plan to add mipmaps for stamp textures and incorporate procedural vector-based stamps to achieve resolution-agnostic effects.

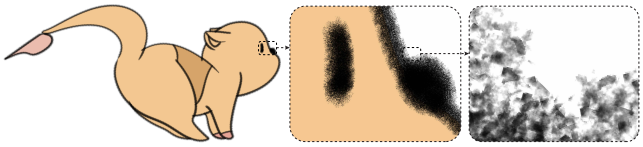


Figure 12: Zoomed-in view of stamp strokes. We render the cat using a stamp brush with a 512×512 resolution footprint. Although our stamp brushes do not support infinite zoom-in, artifacts are only noticeable when zoomed in at an exceptional level.

Our current techniques only support alpha blending, and have limitations when rendering transparent vanilla strokes, as Figure 13 shows. We will explore other possible blending methods in the future. A potential challenge is performance—given that the blending

stage of a graphics pipeline is not programmable, supporting other types of blending would be achieved by enabling extra OpenGL extensions, which can increase computation demand.

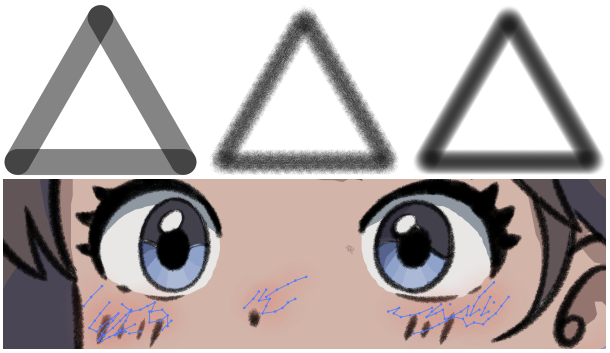


Figure 13: Transparency and self overlapping. When rendering transparent strokes, our strokes create self-overlapping areas. While this may be considered an undesired limitation for vanilla strokes, it is actually a desirable feature for stamp and airbrush strokes. Artists anticipate that strokes will self overlap when using an airbrush or stamp brush. To illustrate this, we show the wireframe of airbrush strokes in Figure 10. The artist drew strokes repeatedly with the airbrush to create a blush effect on the character's face.

Our current prototype system only offers a single stroke layer for users to paint on, and the program always renders the colored regions underneath the strokes to ensure correct rendering order, so no stroke is partially covered by the underlying fills. Extending the current single-layer system to a multi-layer one requires extra design decisions, such as how to fill regions bounded by strokes drawn across multiple layers, and how to infer proper depth ordering of multiple boundaries and fill regions.

We mainly focus on 2D drawings in this paper, but our stroke rendering techniques can be extended to 3D, such as implementing 3D stamp strokes with Blender Grease Pencil. Other interesting directions include extending the range of stamp brush parameters, providing realistic blending modes [Sochorová and Jamriška 2021], handling gaps with Delaunay Painting [Parakkat et al. 2022], and applying vector graphics complexes to fill markers [Dalstein et al. 2014, 2015].

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their detailed feedback, in particular R3 who went above and beyond in validating our approach, R6 who gave exceptionally detailed suggestions on writing and references, and committee members for extensive proofreading and insightful suggestions during the shepherding process. The first author would like to express his sincere gratitude to the Blender Grease Pencil team, in particular Falk David and Clément Foucault, for helping him understand Grease Pencil's source code. He would not have been able to raise the core technical idea behind this research alone at home without their support.

REFERENCES

Adobe Community. 2009. How to Use Photoshop Brushes in Illustrator. https://community.adobe.com/t5/illustrator-discussions/how-to-use-photoshop-brushes-in-illustrator/td-p/2124812. Accessed: 2024-01-22.

Paul Asente, Mike Schuster, and Teri Pettit. 2007. Dynamic Planar Map Illustration. *ACM Trans. Graph.* 26, 3 (Jul 2007), 30–es. https://doi.org/10.1145/1276377.1276415

Patrick Baudelaire and Michel Gangnet. 1989. Planar Maps: An Interaction Paradigm for Graphic Design. *SIGCHI Bull.* 20, SI (Mar 1989), 313–318. https://doi.org/10.1145/67450.67511

Ugo Braga Sangiorgi, Vivian Genaro Motti, François Beuvens, and Jean Vanderdonckt. 2012. Assessing Lag Perception in Electronic Sketching *(NordiCHI '12)*. Association for Computing Machinery, New York, NY, USA, 153–161. https://doi.org/10.1145/2399016.2399040

Zhili Chen, Byungmoon Kim, Daichi Ito, and Huamin Wang. 2015. Wetbrush: GPU-Based 3D Painting Simulation at the Bristle Level. *ACM Trans. Graph.* 34, 6, Article 200 (Nov 2015), 11 pages. https://doi.org/10.1145/2816795.2818066

Nelson S.-H. Chu and Chiew-Lan Tai. 2005. MoXi: Real-Time Ink Dispersion in Absorbent Paper. *ACM Trans. Graph.* 24, 3 (Jul 2005), 504–511. https://doi.org/10.1145/1073204.1073221

Boris Dalstein, Rémi Ronfard, and Michiel van de Panne. 2014. Vector Graphics Complexes. *ACM Trans. Graph.* 33, 4, Article 133 (Jul 2014), 12 pages. https://doi.org/10.1145/2601097.2601169

Boris Dalstein, Rémi Ronfard, and Michiel van de Panne. 2015. Vector Graphics Animation with Time-Varying Topology. *ACM Trans. Graph.* 34, 4, Article 145 (Jul 2015), 12 pages. https://doi.org/10.1145/2766913

Stephen DiVerdi, Aravind Krishnaswamy, Radomir Mech, and Daichi Ito. 2012. A Lightweight, Procedural, Vector Watercolor Painting Engine *(I3D '12)*. Association for Computing Machinery, New York, NY, USA, 63–70. https://doi.org/10.1145/2159616.2159627

Mark Dokter, Jozef Hladky, Mathias Parger, Dieter Schmalstieg, Hans-Peter Seidel, and Markus Steinberger. 2019. Hierarchical Rasterization of Curved Primitives for Vector Graphics Rendering on the GPU. In *Computer Graphics Forum*, Vol. 38. Wiley Online Library, 93–103.

Mark Finch, John Snyder, and Hugues Hoppe. 2011. Freeform Vector Graphics with Controlled Thin-plate Splines. *ACM Trans. Graph.* 30, 6 (Dec 2011), 1–10. https://doi.org/10.1145/2070781.2024200

Kenneth P. Fishkin and Brian Barsky. 1984. Algorithms for Brush Movement in Paint Systems. In *National Computer Graphics Association of Canada Conference—Graphics Interface '84*. 9–16.

Yulia Gryaditskaya, Mark Sypesteyn, Jan Willem Hoftijzer, Sylvia Pont, Frédo Durand, and Adrien Bousseau. 2019. OpenSketch: A Richly-Annotated Dataset of Product Design Sketches. *ACM Trans. Graph.* 38, 6, Article 232 (Nov 2019), 16 pages. https://doi.org/10.1145/3355089.3356533

Paul Haeberli. 1990. Paint by Numbers: Abstract Image Representations. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques* (Dallas, TX, USA) *(SIGGRAPH '90)*. Association for Computing Machinery, New York, NY, USA, 207–214. https://doi.org/10.1145/97879.97902

Siu Chi Hsu and Irene H. H. Lee. 1994. Drawing and Animation Using Skeletal Strokes. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '94)*. Association for Computing Machinery, New York, NY, USA, 109–118. https://doi.org/10.1145/192161.192186

S. C. Hsu, I. H. H. Lee, and N. E. Wiseman. 1993. Skeletal Strokes. In *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology* (Atlanta, Georgia, USA) *(UIST '93)*. Association for Computing Machinery, New York, NY, USA, 197–206. https://doi.org/10.1145/168642.168662

Jennifer Jacobs, Joel Brandt, Radomír Mech, and Mitchel Resnick. 2018. Extending Manual Drawing Practices with Artist-Centric Programming Tools. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3173574.3174164

Apoorva Joshi. 2018. Efficient Rendering of Linear Brush Strokes. *Journal of Computer Graphics Techniques (JCGT)* 7, 1 (Feb 2018), 1–16. http://jcgt.org/published/0007/01/01/

Mark J. Kilgard. 2020. Polar Stroking: New Theory and Methods for Stroking Paths. *ACM Trans. Graph.* 39, 4, Article 145 (Aug 2020), 15 pages. https://doi.org/10.1145/3386569.3392458

Mark J. Kilgard and Jeff Bolz. 2012. GPU-Accelerated Path Rendering. *ACM Trans. Graph.* 31, 6, Article 172 (Nov 2012), 10 pages. https://doi.org/10.1145/2366145.2366191

Joohwan Kim, Josef B. Spjut, Morgan McGuire, Alexander Majercik, Ben Boudaoud, Rachel A. Albert, and David P. Luebke. 2019. Esports Arms Race: Latency and Refresh Rate for Competitive Gaming Tasks. *Journal of Vision* (2019). https://jov.arvojournals.org/article.aspx?articleid=2750799

Brad A. Myers, Ashley Lai, Tam Minh Le, YoungSeok Yoon, Andrew Faulring, and Joel Brandt. 2015. Selective Undo Support for Painting Applications. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (Seoul, Republic of Korea) *(CHI '15)*. Association for Computing Machinery, New York, NY,

USA, 4227–4236. https://doi.org/10.1145/2702123.2702543

Alexandrina Orzan, Adrien Bousseau, Holger Winnemöller, Pascal Barla, Joëlle Thollot, and David Salesin. 2008. Diffusion Curves: A Vector Representation for Smooth-Shaded Images. *ACM Trans. Graph.* 27, 3 (Aug 2008), 1–8. https://doi.org/10.1145/1360612.1360691

Amal Dev Parakkat, Pooran Memari, and Marie-Paule Cani. 2022. Delaunay Painting: Perceptual Image Colouring from Raster Contours with Gaps. *Computer Graphics Forum* 41, 6 (2022), 166–181. https://doi.org/10.1111/cgf.14517

PNGEgg. 2024. PNGEgg - High Quality PNG Images. https://www.pngegg.com/ Accessed: 2024-01-22.

Johannes Schmid, Martin Sebastian Senn, Markus Gross, and Robert W. Sumner. 2011. OverCoat: An Implicit Canvas for 3D Painting. *ACM Trans. Graph.* 30, 4, Article 28 (Jul 2011), 10 pages. https://doi.org/10.1145/2010324.1964923

Maria Shugrina, Chin-Ying Li, and Sanja Fidler. 2022. Neural Brushstroke Engine: Learning a Latent Style Space of Interactive Drawing Tools. *ACM Trans. Graph.* 41, 6, Article 269 (Nov 2022), 18 pages. https://doi.org/10.1145/3550454.3555472

Šárka Sochorová and Ondřej Jamriška. 2021. Practical Pigment Mixing for Digital Painting. *ACM Trans. Graph.* 40, 6, Article 234 (Dec 2021), 11 pages. https://doi.org/10.1145/3478513.3480549

Daniel Sýkora, Mirela Ben-Chen, Martin Čadík, Brian Whited, and Maryann Simmons. 2011. TexToons: Practical Texture Mapping for Hand-Drawn Cartoon Animations. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering* (Vancouver, British Columbia, Canada) *(NPAR '11)*. Association for Computing Machinery, New York, NY, USA, 75–84. https://doi.org/10.1145/2024676.2024689

Daniel Sýkora, John Dingliana, and Steven Collins. 2009. LazyBrush: Flexible Painting Tool for Hand-Drawn Cartoons. *Computer Graphics Forum* 28, 2 (2009), 599–608. https://doi.org/10.1111/j.1467-8659.2009.01400.x

W3C. 2001. Scalable Vector Graphics (SVG) 1.0 Specification. https://www.w3.org/TR/SVG10/ Accessed: 2024-03-26.

W3C. 2014. Variable Width Stroke. https://www.w3.org/Graphics/SVG/WG/wiki/Proposals/Variable_width_stroke

Zeyu Wang, Sherry Qiu, Nicole Feng, Holly Rushmeier, Leonard McMillan, and Julie Dorsey. 2021. Tracing Versus Freehand for Evaluating Computer-Generated Drawings. *ACM Trans. Graph.* 40, 4, Article 52 (Jul 2021), 12 pages. https://doi.org/10.1145/3450626.3459819

John Warnock and Douglas K. Wyatt. 1982. A Device Independent Graphics Imaging Model for Use with Raster Devices. In *Proceedings of the 9th Annual Conference on Computer Graphics and Interactive Techniques* (Boston, Massachusetts, USA) *(SIGGRAPH '82)*. Association for Computing Machinery, New York, NY, USA, 313–319. https://doi.org/10.1145/800064.801297

Ron Wein, Efi Fogel, Baruch Zukerman, and Dan Halperin. 2007. Advanced Programming Techniques Applied to CGAL's Arrangement Package. *Computational Geometry* 38, 1 (2007), 37–63. https://doi.org/10.1016/j.comgeo.2006.11.007 Special Issue on CGAL.

Brian Whited, Eric Daniels, Michael Kaschalk, Patrick Osborne, and Kyle Odermatt. 2012. Computer-Assisted Animation of Line and Paint in Disney's Paperman. In *ACM SIGGRAPH 2012 Talks* (Los Angeles, California) *(SIGGRAPH '12)*. Association for Computing Machinery, New York, NY, USA, Article 19, 1 pages. https://doi.org/10.1145/2343045.2343071

Turner Whitted. 1983. Anti-Aliased Line Drawing Using Brush Extrusion. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques* (Detroit, Michigan, USA) *(SIGGRAPH '83)*. Association for Computing Machinery, New York, NY, USA, 151–156. https://doi.org/10.1145/800059.801144

Chufeng Xiao, Wanchao Su, Jing Liao, Zhouhui Lian, Yi-Zhe Song, and Hongbo Fu. 2022. DifferSketching: How Differently Do People Sketch 3D Objects? *ACM Trans. Graph.* 41, 6, Article 264 (Nov 2022), 16 pages. https://doi.org/10.1145/3550454.3555493

Jun Xing, Hsiang-Ting Chen, and Li-Yi Wei. 2014. Autocomplete Painting Repetitions. *ACM Trans. Graph.* 33, 6, Article 172 (Nov 2014), 11 pages. https://doi.org/10.1145/2661229.2661247

Chuan Yan, John Joon Young Chung, Yoon Kiheon, Yotam Gingold, Eytan Adar, and Sungsoo Ray Hong. 2022. FlatMagic: Improving Flat Colorization through AI-Driven Design for Digital Comic Professionals. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 380, 17 pages. https://doi.org/10.1145/3491102.3502075

Jerry Yin, Chenxi Liu, Rebecca Lin, Nicholas Vining, Helge Rhodin, and Alla Sheffer. 2022. Detecting Viewer-Perceived Intended Vector Sketch Connectivity. *ACM Trans. Graph.* 41, 4, Article 87 (Jul 2022), 11 pages. https://doi.org/10.1145/3528223.3530097

Lvmin Zhang, Chengze Li, Edgar Simo-Serra, Yi Ji, Tien-Tsin Wong, and Chunping Liu. 2021. User-Guided Line Art Flat Filling with Split Filling Mechanism. *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2021), 9884–9893. https://api.semanticscholar.org/CorpusID:235726622

Lvmin Zhang, Chengze Li, Tien-Tsin Wong, Yi Ji, and Chunping Liu. 2018. Two-Stage Sketch Colorization. *ACM Trans. Graph.* 37, 6, Article 261 (Dec 2018), 14 pages. https://doi.org/10.1145/3272127.3275090

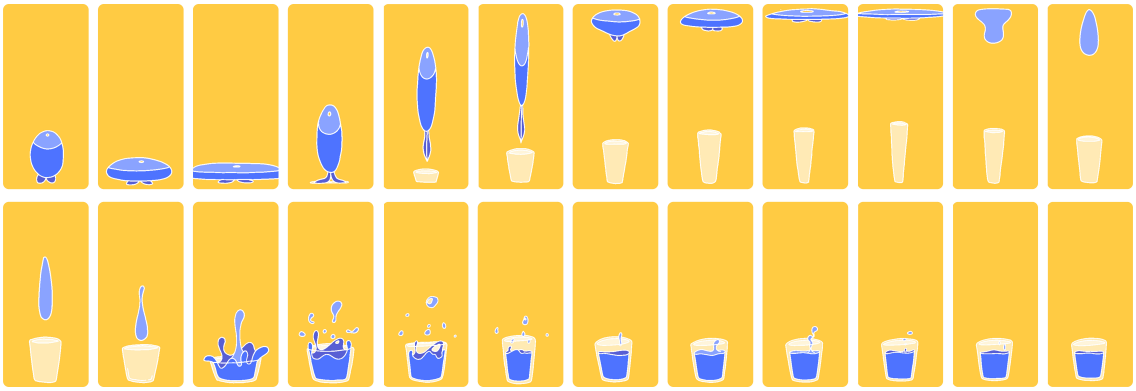


Figure 14: An example animation created using our prototype system. The animation has 24 keyframes, 244 strokes, and 11228 vertices. The full animation can be found in the supplementary video.

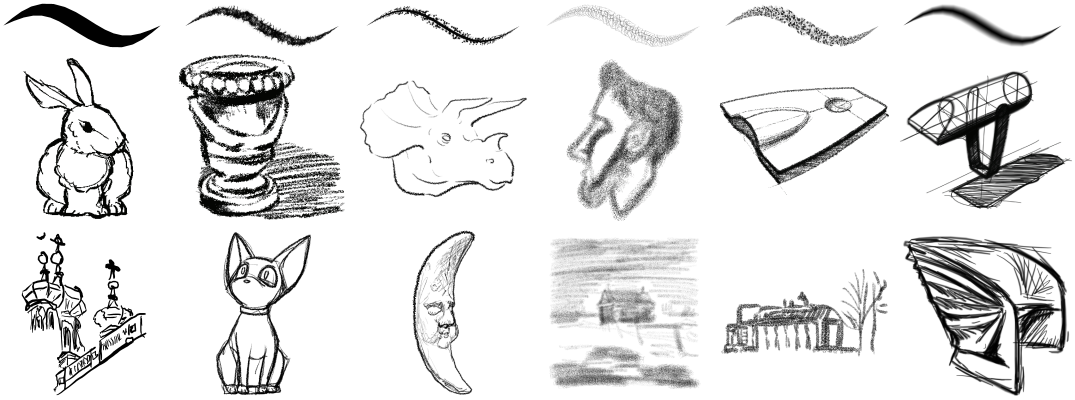


Figure 15: Selected brushes and the corresponding rendered vector drawings from existing datasets [Wang et al. 2021; Xiao et al. 2022]. Each column starts with a brush, followed by rendered drawings with the brush applied. Brushes from left to right are Vanilla, Thorn, Vortex, Polygon, Cobblestone, and Airbrush.

Figure 16: More brush rendering results using stylized stamp brushes and drawings from existing datasets [Wang et al. 2021; Xiao et al. 2022]. These stamp brushes sample colors from footprints [PNGEgg 2024], which is uncommon in practice. Stamps and the corresponding stamp brushes are shown at the top.

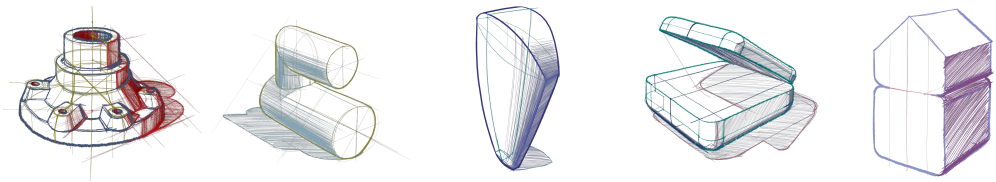
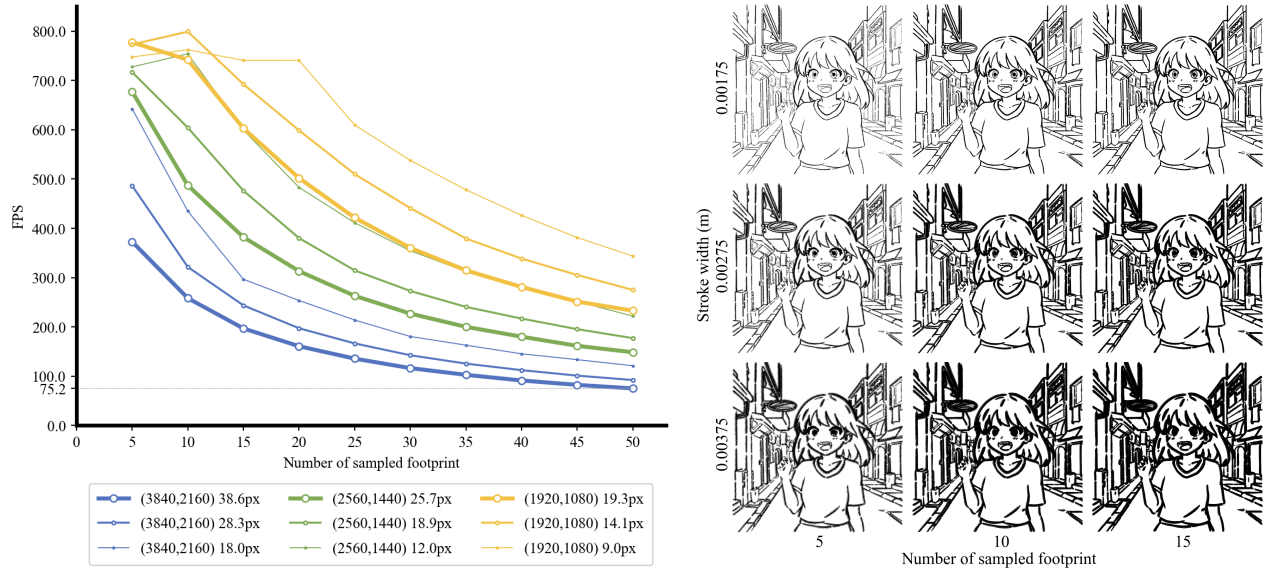
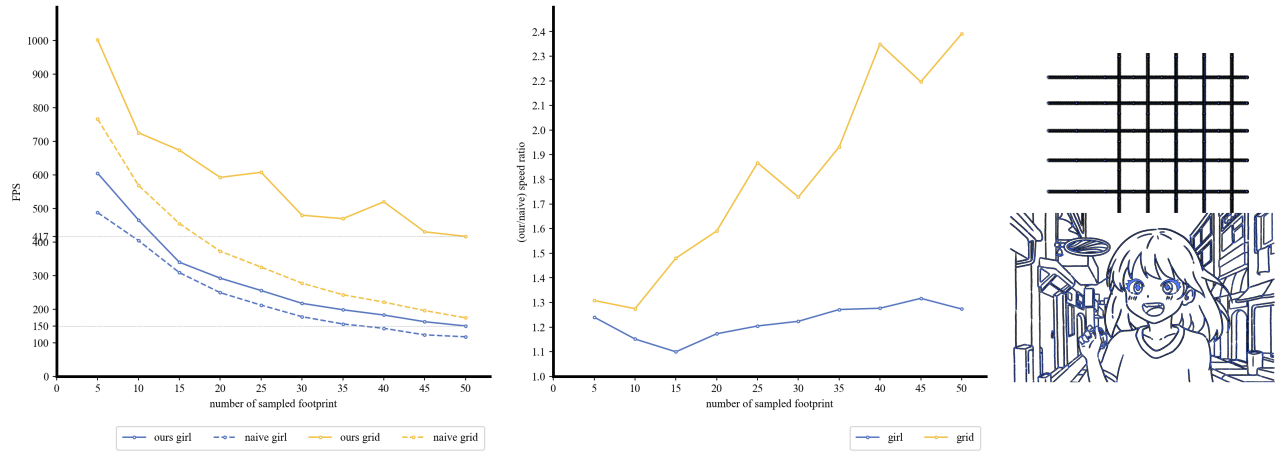


Figure 17: Drawings rendered with multiple brushes. We gathered polylines based on the stroke category labels provided by OpenSketch [Gryaditskaya et al. 2019] and applied different brushes to each group while maintaining visual aesthetics.



(a) FPS Performance (b) Tested Drawings

Figure 18: Performance test of our stamp stroke rendering technique. The experiment was conducted on an NVIDIA GeForce RTX 3050 GPU. We used the unfilled version of the illustration in Figure 10 with 698 strokes and 15578 vertices and rendered all strokes with selected stamp brushes for the performance test. The stamp brush is sampled from a stamp image at a 512×512 resolution without mipmapping. In the experiment, we controlled the stroke width (0.00175, 0.00275, 0.00375 in meters) and canvas size (3840×2160, 2560×1440, 1920×1080 in pixels, 0.37×0.21 in meters) as shown in (a). We recorded the FPS for rendering the drawing with OpenGL when changing the number of sampled footprints. (b) shows selected renderings with different stroke widths and numbers of sampled footprints.



(a) FPS Performance (b) Speed ratio (c) Test drawings

Figure 19: Performance compared to a "naive" approach, which calculates the stamp positions and radii on an Intel i7-127000H CPU at each frame, then converts the position points into texture squares via a geometry shader. We run the test on two drawings, grid and girl, with lower and higher vertex densities, as shown in (c). The vertex positions are visible as small blue dots. They are rendered at 3840×2160 resolution using the same settings in Figure 18. (a) shows the FPS results, while (b) presents the speed ratio. Our method did not show a significant performance advantage ($\times 1.2$) when rendering the drawing with high vertex density but performed well in another case with a high number of stamps ($\times 2$). This difference can be attributed to the need for calculating and synchronizing large amounts of data between CPU and GPU memory with low vertex density. When the vertex density is high, there are very few stamps on short line segments, so there is no significant difference in data synchronization. As a result, the performance gain is not too significant.