Generating and Manipulating Painterly Renderings of Images

By
Grace Todd

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Scholar)

Presented May 22, 2023
Commencement June 2023

AN ABSTRACT OF THE THESIS OF

Grace Todd for the degree of Honors Baccalaureate of Science in Computer Science
presented on May 22, 2023. Title:
Generating and Manipulating Painterly Renderings of Images

Abstract approved:

Eugene Zhang

We present a tool that converts any image into a painterly rendered one, or one that looks as though it was hand-painted. To do this, we implemented the Sobel filter to calculate the edge field that is used to create streamlines, which serve as the foundation for brush strokes. We implement a simple user interface that allows users to apply four different artistic styles to the image, including watercolor, impressionism, expressionism, and pointillism. In addition to providing default styles for an image, the system allows the user to adjust a few stylistic parameters regarding the brush strokes (width, color, opacity, etc.) to provide additional control over the rendered output. These brush strokes are then composited to create the final painterly rendering. This paper also analyzes a few key elements of what could be considered an effective painterly rendering based on the composition of four unique input images. We rely on subject size, position, and overall contrast of an image to determine its influence of each default painterly rendering.

Key Words: painterly rendering, graphics, openGL, images

Corresponding e-mail address: toddgr@oregonstate.edu

Generating and Manipulating Painterly Renderings of Images

By
Grace Todd

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Scholar)

Presented May 22, 2023
Commencement June 2023

Honors Baccalaureate of Science in Computer Science project of Grace Todd presented on May 22, 2023.

APPROVED:

Eugene Zhang, Mentor, representing Computer Graphics and Information Visualization

---

Mike Bailey, Committee Member, representing Computer Graphics and Information Visualization

---

Raffaele de Amicis, Committee Member, representing Computer Graphics and Information Visualization

---

Toni Doolen, Dean, Oregon State University Honors College

I understand that my project will become part of the permanent collection of Oregon State University Honors College. My signature below authorizes release of my project to any reader upon request.

Grace Todd, Author

Contents

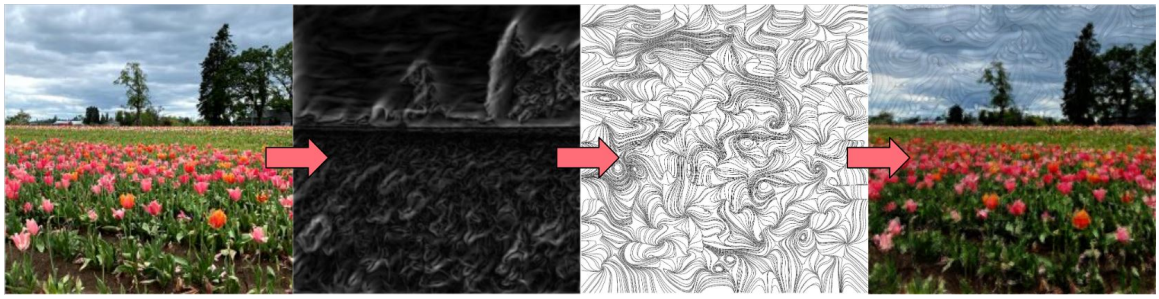


Figure 1: Painterly rendering process on tulip field. From left: original image, smoothed edge field, streamlines, painterly rendering.

1 Introduction

This project aims to generate painterly renderings from an input image. In the context of this project, the phrase *painterly rendering* refers to a series of streamlines derived from an image's edge field, which are then stylized (imitating brush strokes, altering the sizes of the streamlines, adjusting the colors, etc.) to create a new version of the original image that is visually similar to that of a physical painting. Through this method users can emulate different painterly rendering styles from a single image by altering the style parameters for the collection of brush strokes.

1.1 Overview

Although [1] have implemented a similar project through their research, their work lacks a practical interface that allows end users to apply their findings. Previous implementations and crucial developments on this topic often provide access to source code, but could be applied more effectively to increase usability. Therefore, this project also applies a graphical user interface (GUI) that allows for an audience with a wider range of technical skills to be able to implement painterly rendering with greater ease.

This project hopes to expand on the visualization research done by others in the field by presenting unique painterly styles on a variety of input images. This is a difficult problem to solve because it not only emulates the current methods of painterly rendering, but also expands upon them by comparing the effects of different style parameters on images with unique composition. In this context, *composition* refers to the arrangement of elements (subjects, colors, patterns, etc.) within an image. This project also aims to optimize performance by applying these algorithms to adjust the parameters of a painterly rendering while seeing results in relatively real time.

To achieve these goals, we implemented algorithms such as the Sobel operator (Section 4.1) and the fourth-order Runge-Kutta method (Section 4.3) to efficiently composite a painterly rendering from an image. The Sobel operator is used to detect the edges of the

image, and will construct an edge field in the form of a two-dimensional array. Once the edge field is computed, it will be used to create streamlines that follow the general contour of the image using the Runge-Kutta method. These streamlines will then be used to construct the painting's brush strokes after style parameters are applied (Section 3). We apply this project through the lens of a GUI, which will aim to be simple and efficient to ensure peak usability [Figure 1].

1.2 OpenGL, GLUT, and GLUI

OpenGL, or Open Graphics Library, is a cross-language, cross-platform application programming interface (API) used for rendering 2D and 3D vector graphics [10]. This API was crucial for the implementation of images as textures, drawing streamlines, and applying brush strokes. Similarly, the OpenGL Utility Toolkit (GLUT) is a library for implementing OpenGL, specifically performing the I/O operations of the program [5]. In this project, GLUT provides the window system for which the input images– as well as their painterly renderings– are displayed. Finally, the OpenGL User Interface (GLUI) is implemented in this project as a means for user control over the output image [4]. GLUI allows users to adjust style parameters of the program and update automatically, without requiring the user to recompile the project.

1.3 Input Parameters

When the image is loaded as an input, it is stored within the program as a texture. The Sobel filter, which is derived from the original image texture, produces a secondary texture in which the edge field vectors are calculated. When the edge field is calculated from this texture, the pixels of the original image and their associated edge vectors are stored in three-dimensional arrays; specifically, the color and alpha channels of the image are stored in an array of size $NPN * NPN * 4$ and the edge field vectors are stored in an array of size $NPN * NPN * 2$, where NPN is the length of the image.

When the streamlines of the image are drawn, they are rendered in the canvas space. The term *canvas*, for the purposes of this project, refers to a mesh of vertices of equal length and width apart, which uses the entire window space of the program. The number of vertices used to create this canvas can be adjusted, but generates a 20 by 20 mesh of vertices by default. These vertices are used to define the boundaries of the canvas and are used as references to calculate the streamlines of the image.

Streamlines and their derived brush strokes are stored in C++ standard library vectors, where the streamline vector holds lines, and the brush strokes are stored as a vector of vertices. Objects of the Line class hold information about the start and end of each line segment of the streamline. Vertex class objects hold information about the point's position on the canvas and its color, which is sampled from the original image.

In this paper, four different input images are used to demonstrate the effect that image composition has on the output: a picture of mountains, a dog, a tulip field, and a koi pond.

Each of these images has a unique color palette with varying subject size to portray the differences in edge field and brush stroke output. For example, the dog, which is the sole subject of the image, takes up the entire frame, whereas each tulip in the tulip field is smaller in the frame. This will yield a higher variation in painterly results and provide a deeper context to the rendering steps.

2 Previous Work

This project aims to extend the efforts of [1] with painterly rendering by developing a GUI through which users can adjust the rendering results. We also considered implementing techniques found in Image-Based Flow Visualization [9] to quickly simulate two-dimensional flows on the same plane as the image, which is a crucial element of work done by [1] as well.

[1] introduced a method transforming images and videos into painterly animations. Their work implements streamlines calculated from contours in an image to create more dynamic brush strokes. For the purposes of our work, we plan to implement similar style parameters for our own brush stroke application, and expand upon their brush stroke parameters to allow for increased user control over the final output.

[9] presents a method to synthesize two-dimensional flows with their technique, Image Based Flow Visualization (IBFV). IBFV assists in the generation of an image's edge field, which is crucial for creating accurate brush strokes in our composited painterly rendering. Generally speaking, IBFV is a quick and efficient method of populating an image or video's edge field and applying a moving flow to the initial input. While this method of visualization was considered for this project, we ultimately determined that calculating the edge field directly from the image yielded more dynamic and abstract results.

[2] describes a method to create an image with a hand-painted appearance from a photograph. Their framework encompasses a wide variety of artistic styles, so a single photograph as an input can produce several unique images. Hertzmann's research into these methods is paramount for our own project, wherein we implement a range of style parameters for the user to choose from in our user interface. Several of the parameters implemented by [2] such as brush size, opacity, color jitter, and blur factor are implemented in this project as well.

There have also been a wide range of painterly rendering applications that derive outputs using artificial intelligence, such as DALL-E [7]. While these types of painterly renderings can produce significant results, they tend to hide the rendering process from the user. While the user is typically able to manipulate most aspects of the output, the anticipated outcome can be somewhat unpredictable, and rarely provide parameter alterations in real time. Because our implementation applies brush stroke stylization without the assistance of neural networks or randomization, the user is potentially able to manipulate the output to a finer and more predictable degree, and without deriving results from sources outside of the initial input.

3 System Overview

To expand upon previous concepts presented, this project aims to implement painterly rendering with a GUI that allows its users to work with style parameters directly.

3.1 Style Parameters

In order to implement brush strokes in such a way that a user can stylize the painterly rendering output, there are a number of parameters that must be mutable. For our purposes, we have implemented the following style parameters initially presented by [2]:

- **Brush width:** The user is able to specify the average size of the brush stroke used.
- **Blur factor:** Controls the size of the Gaussian kernel. As the blur factor increases, less noise is presented in the image. Blur factor is not required for the rendering of an image.
- **Stroke lengths:** Used to restrict the maximum stroke length. This style parameter is useful for rendering different artistic styles.
- **Opacity:** Specifies the opacity of the paint, between 0 and 1. Lower opacities can be used to create a more washed-out effect.
- **Brush spacing:** This determines the amount of space between the discs that are rendered as part of a streamline.
- **Brush stroke concentration:** Controls how many brush strokes are drawn on the screen.
- **Color jitter:** Factors to randomly add jitter to red, green, and blue color components. A color jitter of 0 produces no random jitter, and larger values increase the factor.
- **Brightness:** The brightness of the output image. Higher values of brightness will produce a painting with lighter color values, contributing to a faded appearance in the image.

In addition to these style parameters, the user is able to adjust the painterly rendering based on a select number of preset styles, which apply stylizations specific to popular painting styles. The following are a subset of styles that the user can apply directly:

Pointillism: Composed of small, tiny dots forming the image. To achieve this effect, we scale down the brush stroke size and space out the discs of a brush stroke to make the dots more prominent (Section 5.3).

Impressionism: Characterized by small, loose brush strokes that offer the bare impression of form, typically implementing lighter colors. We achieve this effect by increasing

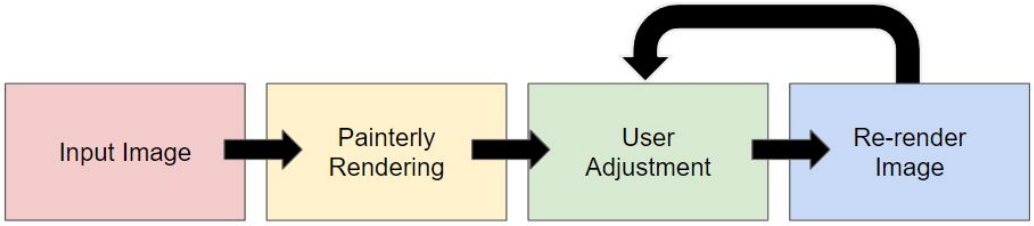


Figure 2: System Process Diagram

the brightness of the overall image, increasing the stroke concentration and decreasing the brush size to form smaller, longer brush strokes (Section 5.4).

Watercolor: Large, light, and transparent brush strokes lightly layered over each other. This effect is achieved by increasing the brightness, brush spacing, and brush width, and lowering the opacity and brush stroke concentration on the canvas (Section 5.5).

Expressionism: Thick brush strokes, angular lines, and vivid colors. We implement this style by increasing both the brush stroke size and the overall concentration, lowering the brightness of the image (Section 5.6).

3.2 System Goals

To ensure the proper usability for this project, the user interface is simple and objectively straightforward. The system's interface will allows the user to do the following [Figure 2]:

1. **Import an image file in PPM format:** The user inputs a PPM file with 1:1 aspect ratio and a minimum of 256 by 256 pixels.

2. **Render a painterly composite of the original image:** While the user may be able to adjust the style parameters of the painterly rendering, they are not required to do so. A user can simply import the image and view a painterly rendering, without requiring any changes on the user's part. The program outputs a preview rendering, which would allow the user to further edit the painting before creating a complete rendering.

3. **Adjust the style of brushstrokes on the image:** The system will include a GUI that allows the user to directly edit the aforementioned style parameters.

4. **Re-render the image according to the new style parameters:** The preview window in the application should be able to hotload updated style parameters as the user adjusts them; in other words, the changes made to the painterly rendering should be applied in real time. For example: a user imports an image to the application and the application window presents a painterly rendering, but the user wants to adjust the brush size. The user decreases the brush size by changing the value of the brush size, and the application updates the painterly rendering. The user is able to view their

style alterations to the image, and then decides that they would rather increase the brush stroke size. The user adjusts the value for the brush size accordingly, and the preview image is updated.

4 Algorithms

In order to create a painterly rendering of an input image, our primary focus is on algorithms regarding edge field computation, streamline creation, and brushstroke stylization for the final output. The following subsections describe the primary algorithms that were implemented in this project.

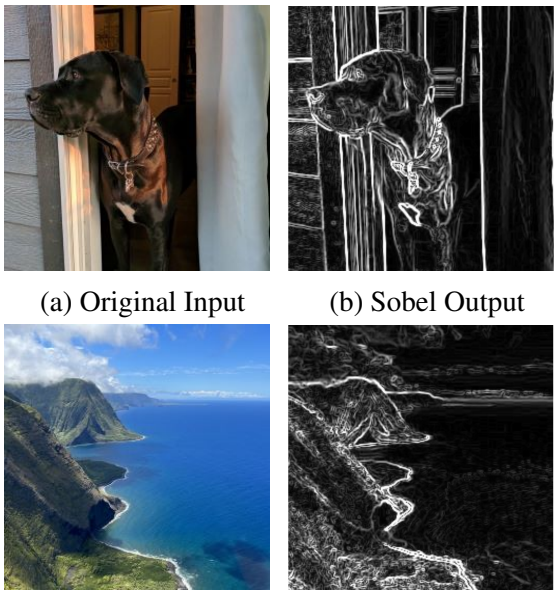
4.1 Edge Detection

The final rendering of the image– particularly the direction and quantity of the brushstrokes– is based on the computed edge field of the original image. This is done through edge detection, more specifically using the Sobel filter [8]. The Sobel filter takes an image as input and creates a new image with emphasized edges. Figure 3 demonstrates how the edges of the original images (3a and 3c) are extracted and displayed on a black background (3b and 3d).

In this process, each pixel of the image is multiplied by a luminance coefficient in order to determine the intensity for each pixel. The following formula is used to determine the intensity:

$$c = 0.299 * p_r + 0.587 * p_g + 0.114 * p_b$$

Where c is the intensity coefficient applied to the red, green, and blue channels of each pixel p. The red, green and blue channels each have their own unique coefficient, which is weighted based on how easily the human eye can see that particular color. Green is the most visible color to the human eye, followed by red and ending with blue.



(a) Original Input (b) Sobel Output

(c) Original Input (d) Sobel Output

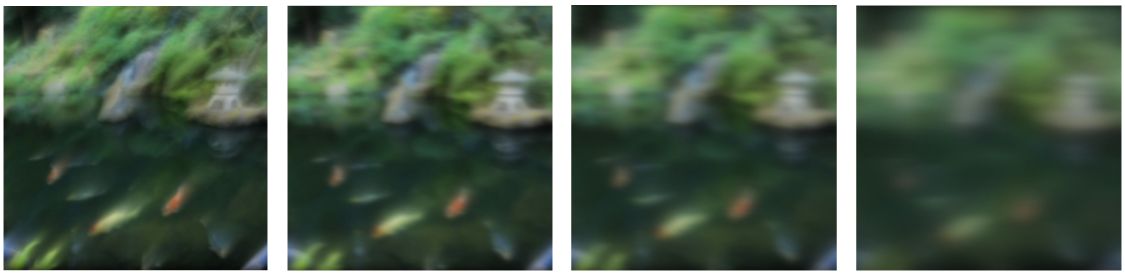
Figure 3: Picture of a dog (top) and mountains (bottom) as input for the Sobel filter.

The image is then convolved with the Sobel kernels in both the horizontal and vertical directions. In terms of image processing, *convolution* refers to a 3x3 matrix of surrounding pixels, which are used to compare the change of intensity in the selected pixel in any given direction. This intensity change is used to compute the gradient vector for that pixel, which points in the direction of the greatest change in intensity. The horizontal and vertical derivative approximations which are used to determine the influence of each of the pixel neighbors as follows:

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * A, \; G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

Where A is the input image, and G_x and G_y are two images which at each point contain the horizontal and vertical derivative approximations. For our purposes, the Sobel operator is the ultimate candidate for to extracting the original image's edge field with efficiency. The edge field, once produced as a texture, is produced by sampling the gradient vector at each texel and directly assigning it to the corresponding vertex on the canvas. When this process is complete, this edge field is implemented for streamline creation. This operation is relatively inexpensive in terms of computing. However, the gradient approximations produced by the Sobel filter can be fairly crude, especially in high-frequency variations. Although we do not expect to encounter high-frequencies in our input images, since we are using this Sobel filter to compute an edge field instead of rendering the edge field directly, artifacts found in higher-frequency variations are easier to dilute through edge smoothing.

4.2 Edge Field Smoothing



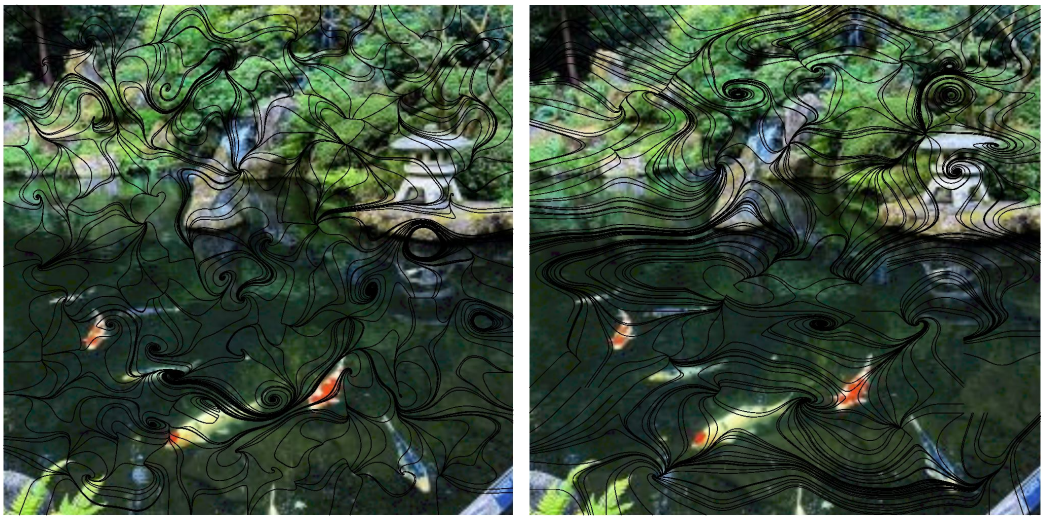
(a) $\sigma = 1$ (b) $\sigma = 3$ (c) $\sigma = 5$ (d) $\sigma = 10$

Figure 4: Gaussian blurring applied to image of koi pond with varying sigma values.

Smoothing is achieved using Gaussian smoothing [11], and can be applied to more complex images to reduce artifacts in the edge field. Gaussian smoothing is used in image processing to reduce image noise and detail. To achieve this effect, we convolve each pixel of the image with a Gaussian function:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

Where σ is the standard deviation and x is the position of the pixel within the image. As σ increases, the image will become more blurred [Figure 4]. The Gaussian function also expresses normal distribution in statistics [6]; in a similar fashion, the Gaussian function is used to sample the color of the image at neighboring pixels and assign decreasing amounts of weight as the function moves farther from the central pixel. The size of the kernel applied



(a) Without smoothing (b) With smoothing, $\sigma = 10$

Figure 5: Streamlines applied to image of koi pond

is dependent on the size of σ, which means that as σ increases, so does the size of the pixel neighborhood used for smoothing.

Smoothing is not required to produce an effective painterly rendering, however it may provide more interesting results by reducing potential artefacts and forming more cohesive streamlines. [Figure 5].

4.3 Streamline Creation

Streamlines in computer graphics and visualization are used to denote the contours of the subjects in the input image. The resulting edge field from the Sobel operation and Gaussian smoothing will be used as the foundation for creating streamlines.

Our methodology for this algorithm is similar to that of [2]. We implement these streamlines by mapping the Sobel filter texture onto the canvas, so that we have quads and vertices from which to reference for our streamline creation. The gradient vectors of the calculated edge field determine the direction of each of the initial streamlines, but we need to create a reference image from which we can determine the length of each streamline. The reference image from which we compute length and sample color will be our original image, optionally with a Gaussian blur applied, which will allow us to average the colors of pixel neighborhoods in order to compute significant color changes.

We will also implement a maximum streamline length, to ensure that brushstrokes remain within a reasonable range, even when considering user input. For the purposes of this project, the user should be able to choose from a variety of brush stroke lengths, from small dots [Figure 21] to long, sweeping brush strokes [Figure 25].

4.3.1 Texture Conversion

In order to navigate the vertices of the canvas with respect to the results from the edge field texture, we can find x and y in the canvas by sampling the texture at point (c, r):

$$c(x) = \frac{x \cdot (\Delta c) + (c_{min} \cdot x_{max} - c_{max} \cdot x_{min})}{\Delta x}$$
$$r(y) = \frac{-y \cdot (\Delta r) + (r_{max} \cdot y_{max} - r_{min} \cdot y_{min})}{\Delta y}$$
$$c^{-1}(x) = \frac{c \cdot (\Delta x) - (c_{min} \cdot x_{max} - c_{max} \cdot x_{min})}{\Delta c}$$
$$r^{-1}(y) = \frac{r \cdot (\Delta y) - (r_{max} \cdot y_{max} - r_{min} \cdot y_{min})}{-\Delta r}$$

Where $r_{max} = y_{min}$ and $r_{min} = y_{max}$. This is because of the way that the pixels in the texture are arranged in relation to the canvas [Figure 6]. These equations are used to convert vertex values from the canvas space to the vertex space and vice versa. Once we have found that point on the texture, we can sample the magnitude vector at that texel, which was derived by using the Sobel filter. We use this vector to find the next point on the line, the coordinate at which the vector is pointed. These coordinates are then converted to the canvas space using $c^{-1}(x)$ and $r^{-1}(y)$, and the two points in the canvas space are used to derive the vector at the control vertex on the canvas. When all of the streamlines are drawn, the brush stroke stylization can take place. The resulting streamlines will be used for the general position and direction of the painting's brush strokes.

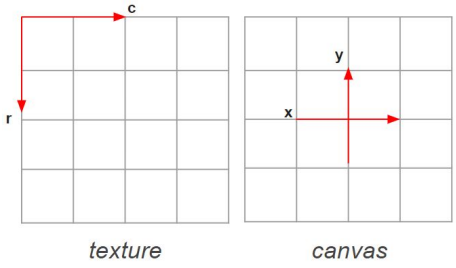


Figure 6: Texture vs. canvas coordinates

4.3.2 Calculating Streamlines

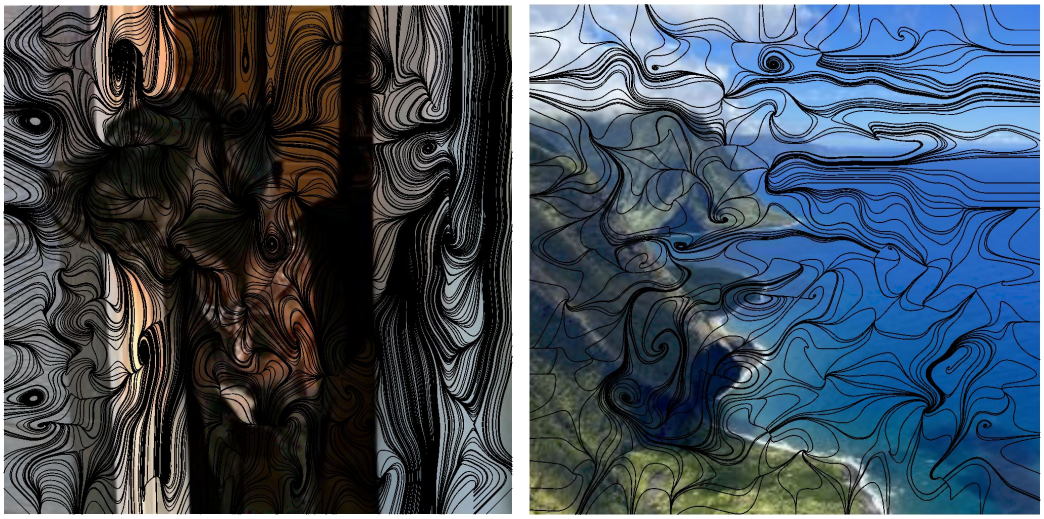
When the conversion from pixel to vertex space is established, the streamlines can then be calculated. We implement our streamlines via the fourth-order Runge-Kutta algorithm.

```
function build_streamline(x, y)
    curr_quad = find_quad()
    curr_position
    next_postion

   while curr_quad not NULL and i < step_max
       curr_quad = streamline_step(forward)
       create_line(curr_position, next_position)
       add_point(curr_position)
       add_line()
```

```
    curr_quad = find_quad()
    curr_position
    next_postion

    while curr_quad not NULL and i < step_max
        curr_quad = streamline_step(backward)
        create_line(curr_position, next_position)
        add point(curr_position)
        add_line()
```

```
function streamline_step(curr_position, next_position, curr_quad,
    forward)

    for vertex in curr_quad
        find_texture_coord()

    for vertex in canvas
        if in curr_quad then assign_vector_to_vertex()

    vector = bilinear_interpolation(curr_quad)
    normalize(vector)
    next_quad = curr_quad

    if vertices in next_quad
        cross_vectors(vector, vertices)
        dot_product(vector, vertices)
        next_quad = find_next_quad()

    return next_quad
```

The algorithm itself is fairly straightforward: informally, we start with a seed point on the image, and continue in the direction which is normal to the vector gradient. If the normal points to the inside of a quad– that is, not directly at another vertex– then we add its vector to the streamline through bilinear interpolation. The streamline building algorithm follows this general procedure until the maximum number of streamline steps has been reached, or the streamline has reached a boundary of the canvas, at which point we begin the process over again to back trace the streamline from the initial vertex. Later in this process, when we stylize brushstrokes over these streamlines, the streamline will be broken into brush strokes of varying lengths, with maximum and minimum lengths implemented as style parameters (Section 3.1).



(a) Dog (b) Mountains

Figure 7: Streamlines results displayed on top of original image.

In more complex images, streamlines do not produce precise outlines of the input. Because our implementation uses bilinear interpolation [3], each step of the streamline is the averaged result of the edge vectors from each vertex in a quad. The position of the next streamline step, shown as the black point (x,y) in Figure 8, is determined by the sum of the products of each corner vertex and their opposite partial area of the quad (coordinated by colors blue, red, green and yellow in the figure), divided by the total area of the quad.

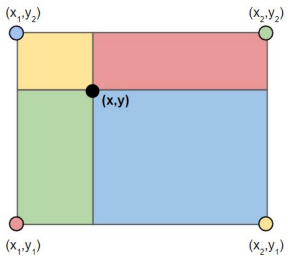


Figure 8: Bilinear interpolation visualization.

4.4 Brush Stroke Stylization

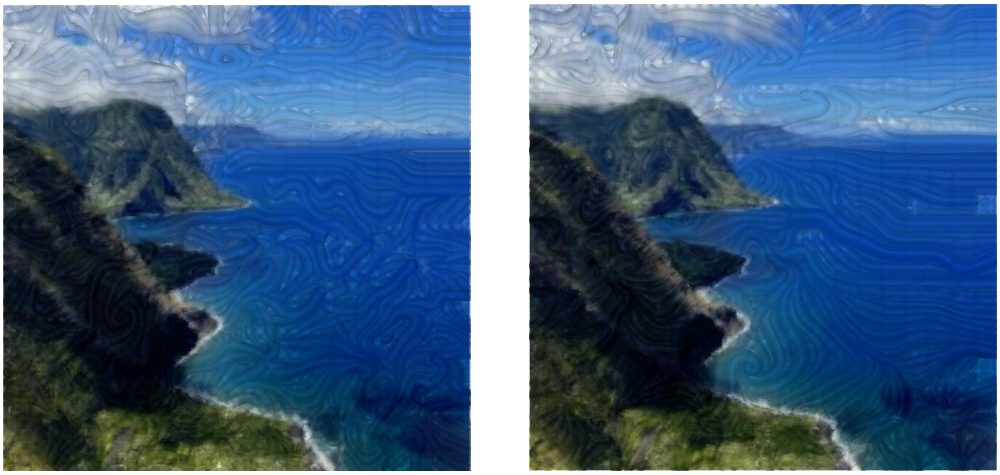
Once the streamlines for the image or video have been derived, we then take the collection of streamlines and stylize them to simulate the brush strokes of a paintbrush.

Within the context of this application, *stylization* of brush strokes refers but is not limited to the size, shape, color, and opacity of the brush stroke. These qualities can be determined by the user, which will ultimately allow them to customize a unique painterly rendering of an image. Depending on these style parameters, the user is also able to emulate a variety of painting styles such as Pointillism, Impressionism, Watercolor or Expressionism from the same original image.

Brush strokes are applied by drawing discs at each step of the calculated streamline. The collective rendering of discs for a single streamline dictates a single brush stroke; however,

there is also the option of drawing the discs sufficiently far apart such that the discs are implemented as their own brush strokes. This can be particularly effective in watercolor images (Section 5.5), where the brush strokes tend to be more blotted. Our final brush stroke style is determined by either the provided user input or the default values. Regardless of the stylization applied to the brush strokes, the final brush strokes are rendered over the original image. This allows more transparent brush strokes to maintain a finer level of detail by blending the brush stroke with the original image underneath.

5 Artistic Exploration

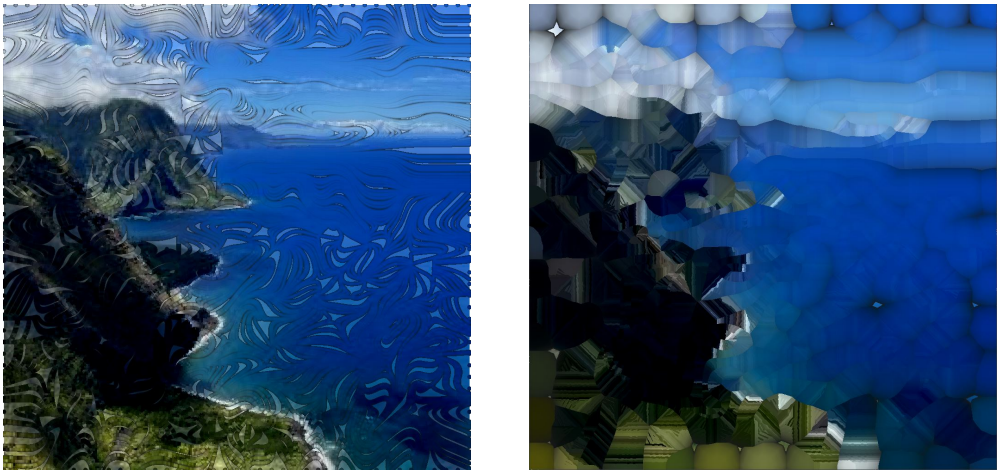


(a) No smoothing (b) $\sigma = 3$

Figure 9: Impressionistic painting with smoothing

Through testing various style parameters for each image we noticed the most desirable outcomes are sufficiently opaque, with an adequate brush width to stroke concentration ratio such that the entire canvas is covered. Smoothed edge fields produced painterly renderings with fewer sharp edges and artifacts, but were less likely to preserve the contours of specific objects in the scene. For instance, Figure 9 demonstrates with an image of a mountain that as the standard deviation of the Gaussian smoothing increases, the less the brush strokes take into account the shape of the clouds or the mountains. The brush strokes flow more smoothly from object to object, allowing the colors to define the edges instead of the direction of the brush stroke. This could be more or less desirable, depending on the composition of the painterly rendering; if the input image contains smaller subjects and requires finer details to be accurately rendered, then Gaussian smoothing would likely not produce satisfying results. However, if the subject of the painterly rendering is large and has a composition that can be maintained even with the degradation of minute details, Gaussian smoothing has the potential to enhance the results and portray a more painterly form.

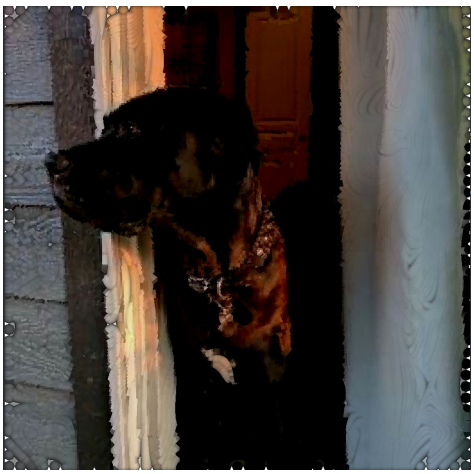
5.1 Varying Stroke Widths



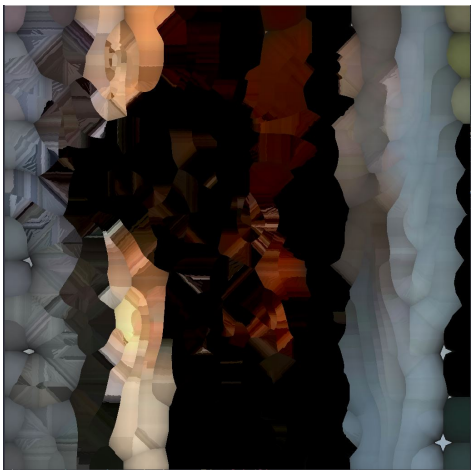
(a) Stroke width = 0.1 (b) Stroke width = 1.25

Figure 10: Varying stroke widths for mountain.

While generating different presets to create the painterly renderings of each of the default styles (Pointillism, Impressionism, Watercolor, Expressionism), one of the more relevant parameters was brush stroke width, which is paramount in achieving a specific look or feel [Figures 10, 11, 12, 13]. Smaller brush strokes easily portray the finer details of an image, regardless of the subject matter, whereas larger stroke widths give the painting a more blocky, abstract appearance. Larger stroke widths could be desirable in cases where the intention of the painting is more abstract and the subject of the painting is sufficiently large [Figure 11b]. Smaller brush strokes may be more desirable when the pattern of the brush stroke directions needs to be more refined [Figure 10b], or when the subjects of the painting are small and require more detail [Figure 13b].



(a) Stroke width = 0.1



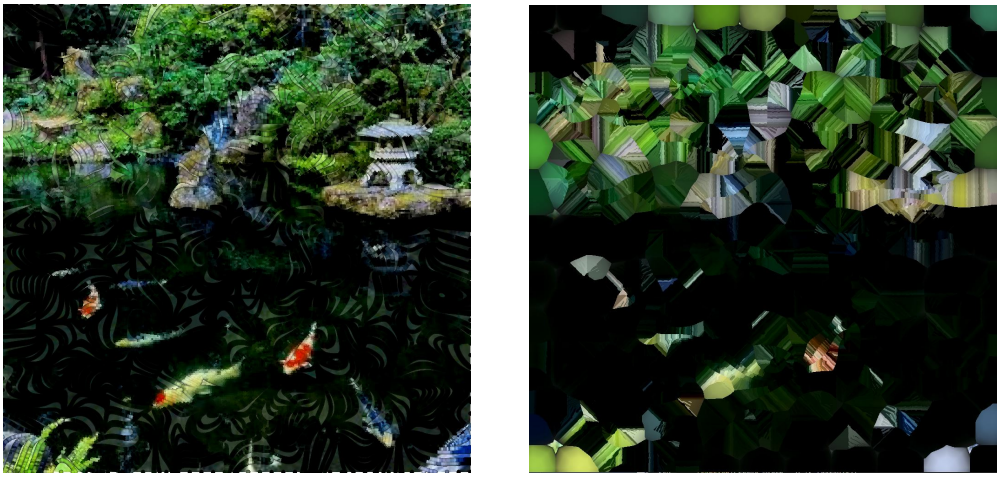
(b) Stroke width = 1.25

Figure 11: Varying stroke widths for dog.

(a) Stroke width = 0.1

(b) Stroke width = 1.25

Figure 12: Varying stroke widths for tulip field.



(a) Stroke width = 0.1 (b) Stroke width = 1.25

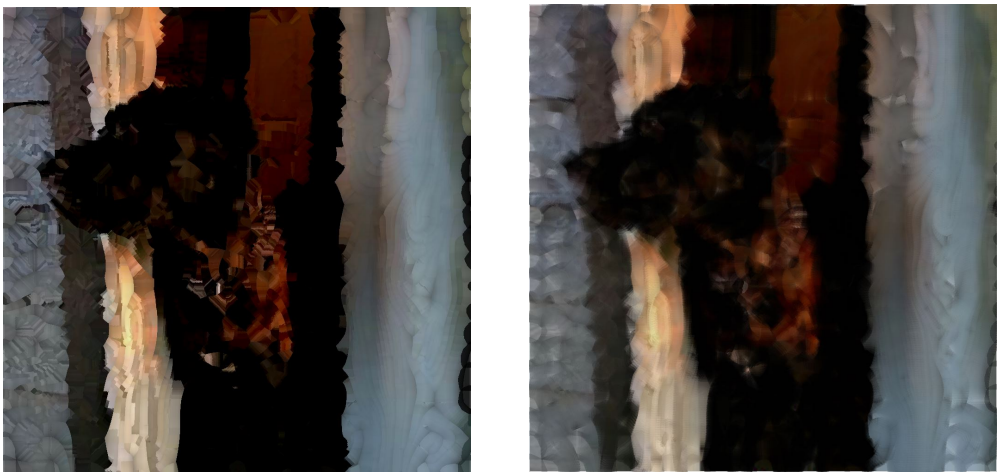
Figure 13: Varying stroke widths for Koi pond.

5.2 Varying Opacity

(a) Opacity = 75% (b) Opacity = 25%

Figure 14: Varying opacity for mountain painterly renderings.

In addition to varying stroke width, opacity is also a primary contributor to the quality of the painterly rendering [Figures 14, 15, 16, 17]. A higher opacity rendering tends to portray more vivid colors and sharper lines, whereas lower opacity gives a painting a more blended and softer look. High opacity may be desirable in painterly renderings that demand sharp, vivid lines with high contrast and blocky detail [Figures 16a, 15a]. Since the brush strokes



(a) Opacity = 75%

(b) Opacity = 25%

Figure 15: Varying opacity for dog painterly renderings.

are blended with the original image, lower opacity is effective in maintaining the detail of a painting, while also giving the brush strokes a smoother and faded appearance. This could be ideal for input images with more complex subjects [Figures 16b, 17b], or to smooth the brush stroke lines to soften the image [Figure 14b].

(a) Opacity = 75%

(b) Opacity = 25%

Figure 16: Varying opacity for tulip and koi pond painterly renderings.

(a) Opacity = 75%

(b) Opacity = 25%

Figure 17: Varying opacity for tulip and koi pond painterly renderings.

5.3 Pointillism

Each of the default styles implemented a specific set of style parameters, which would produce the desired effect on each of the four images. For Pointillism [Figures 18, 19, 20, 21], brush stroke width was reduced significantly to accentuate the circularity of the points. The brush stroke concentration was increased to ensure that the output was sufficiently covered. The opacity was set to 50 percent, which was low enough to achieve a softer brush stroke appearance, but high enough that the integrity of the brush stroke was maintained. The brush stroke length was increased, which allowed for more dots to compose the painterly rendering, and the brush stroke spacing was set far apart such that the points would not overlap with each other. This style produced interesting results across all of the images. The detail in each photo was sufficiently maintained, and the path of each brush stroke is subtle but distinct. The precise parameters for this painterly rendering are as follows:

Brush width: 0.25
Brush stroke concentration: 0.5
Opacity: 0.5
Streamline step maximum: 1000
Brush stroke spacing: 0.5

The following pages are the Pointillism painterly renderings adhering to these parameters.

Figure 18: Pointillism on mountains

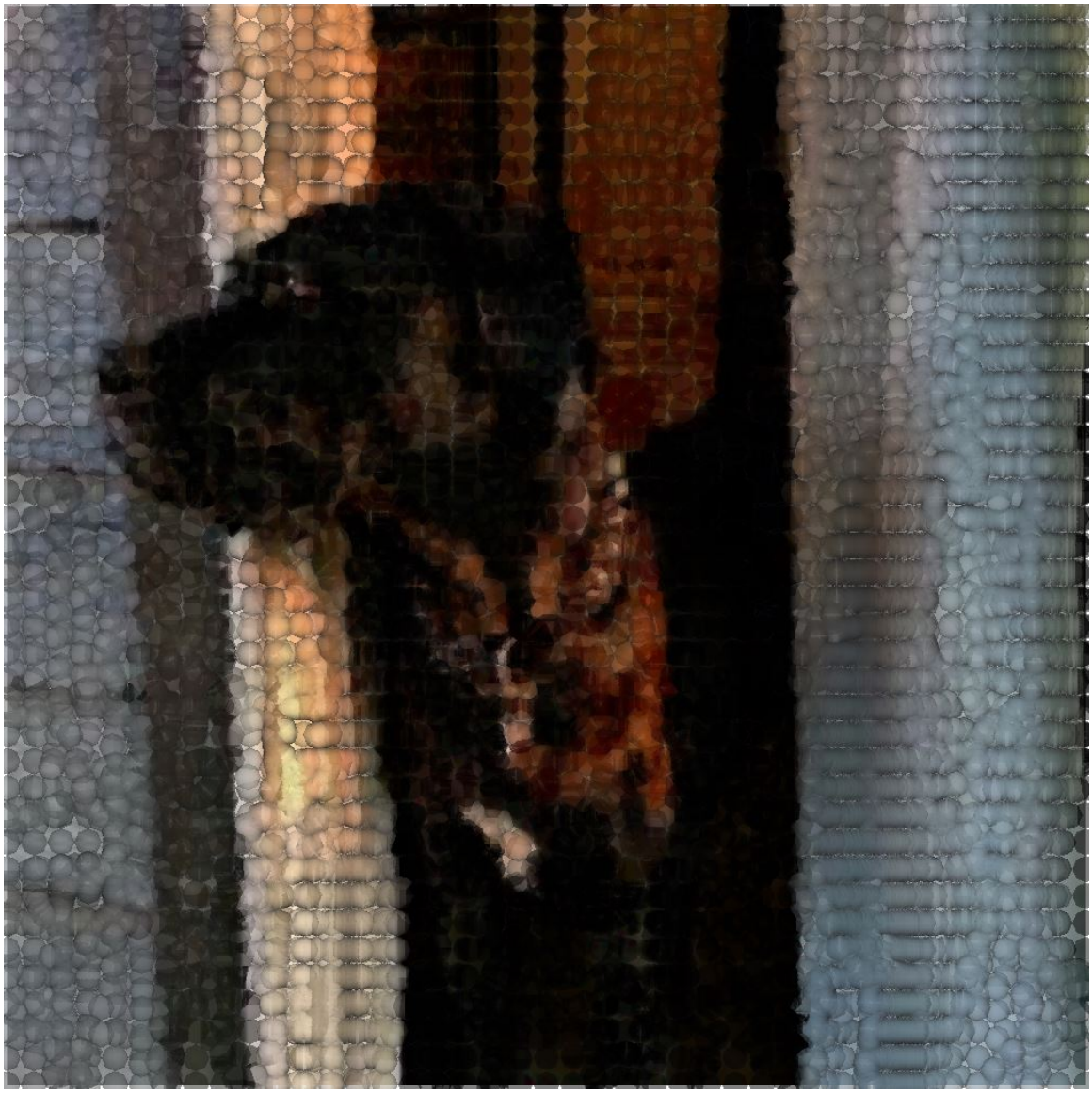


Figure 19: Pointillism on dog

Figure 20: Pointillism on tulip field

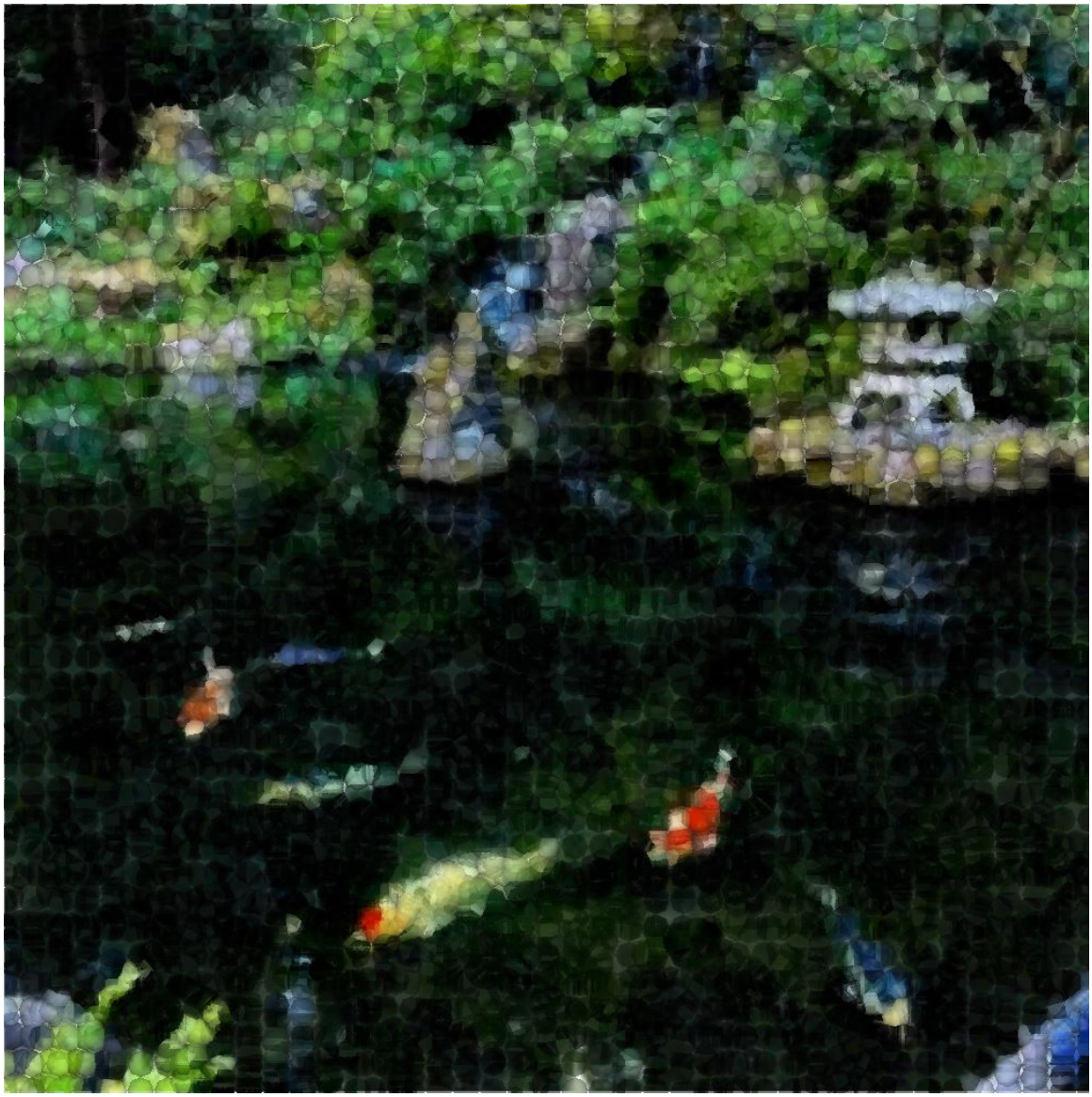


Figure 21: Pointillism on Koi pond

5.4 Impressionism

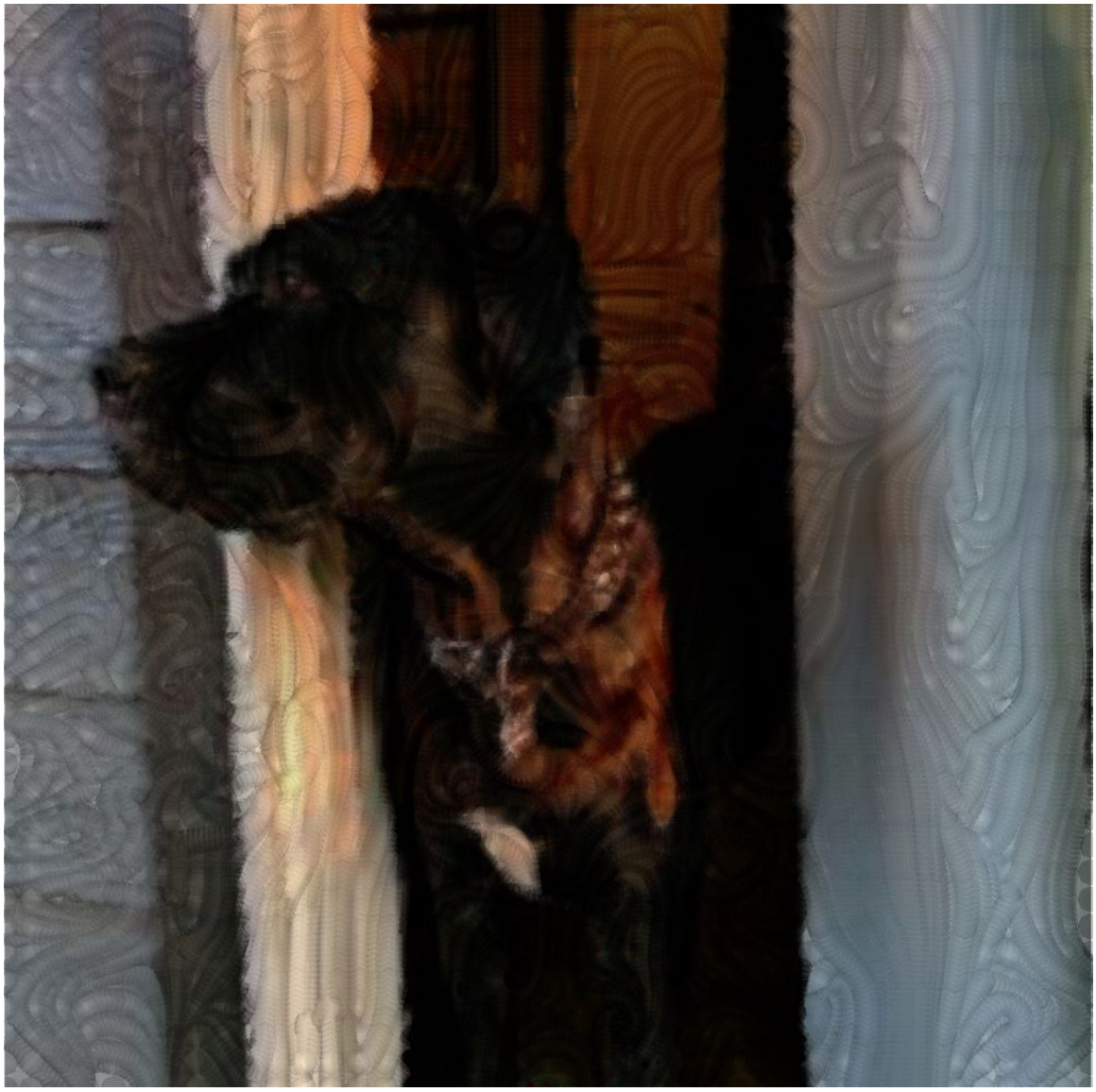


Figure 22: Impressionism on dog

For Impressionism [Figures 22, 23, 24, 25], brush stroke width was also reduced significantly to increase the definition in the brush strokes. The brush stroke concentration was also increased to ensure that the canvas was sufficiently covered. The opacity was set to 25 percent, which was low enough to achieve a softer brush stroke appearance with the amount of detail portrayed by the small strokes. The brush stroke length was decreased, which resulted in brush strokes that are long enough to portray the shape of the painting,

but short enough that the brush strokes do not excessively overlap with each other. The brush stroke spacing was set far apart such that the opacity would blend properly with the image, but close enough together such that the discs formed a cohesive line. The precise parameters for this painterly rendering are as follows:

Brush width: 0.25
Brush stroke concentration: 0.5
Opacity: 0.25
Streamline step maximum: 750
Brush stroke spacing: 0.1

The following pages are the Impressionistic painterly renderings adhering to these parameters.

Figure 23: Impressionism on mountains

This style produced interesting results across all of the images, especially in areas where the image's detail was not previously known. For instance, in the mountain painting [Figure 23], the blue sky and the ocean are drawn with brush strokes that seem to swoop and swirl. In the original images for these renderings, the sky does not show these changes in color. Effectively, these painterly renderings are adding patterns and styles to the image where there may have been no detectable patterns before.

Figure 24: Impressionism on tulip field

Figure 25: Impressionistic style on fish pond

In images which have more complex composition, such as Figure 25, the brush strokes themselves are not as prominent. There are a few select spots where the brush strokes can be seen, such as on the fish and the rocks, but are otherwise blended into the details of the image. This may be because the detail of each brush stroke, whose color darkens around the edges, is lost in the details of the painting itself. The shadow of the brush strokes in Figure 25 are not as noticeable around the details of the shrubbery in the top half of the photo. While Impressionism still yields satisfying results in more complex images, perhaps this style is better suited to input images of simpler composition [Figures 22, 23].

5.5 Watercolor

Figure 26: Watercolor style on tulip field

For Watercolor [Figures 26, 27, 28, 29], brush stroke width was increased significantly to make the painterly rendering appear more faded and reduce the feature definition of the image. The brush stroke concentration was reduced to ensure that each brush stroke is displayed prominently. The opacity was set to 25 percent, which was low enough to achieve a softer brush stroke appearance and sample some of the definition from the original image to blend into the larger brush strokes. The brush stroke length was decreased, which resulted in shorter, less turbulent streamlines for the brush strokes to follow. The brush stroke

spacing was set sufficiently far apart such that the individual discs could be seen without excessively overlapping with each other; this ensured that the opacity would blend with the background instead of other brush strokes, and that the brush strokes themselves would be properly defined. The brightness was also increased from the -0.2 baseline in order to give the painterly rendering a brighter, more washed-out effect. The precise parameters for this painterly rendering are as follows:

Brush width: 1.25
Brush stroke concentration: 2.
Opacity: 0.25
Streamline step maximum: 500
Brightness: 0.

The following pages are the Impressionistic painterly renderings adhering to these parameters.

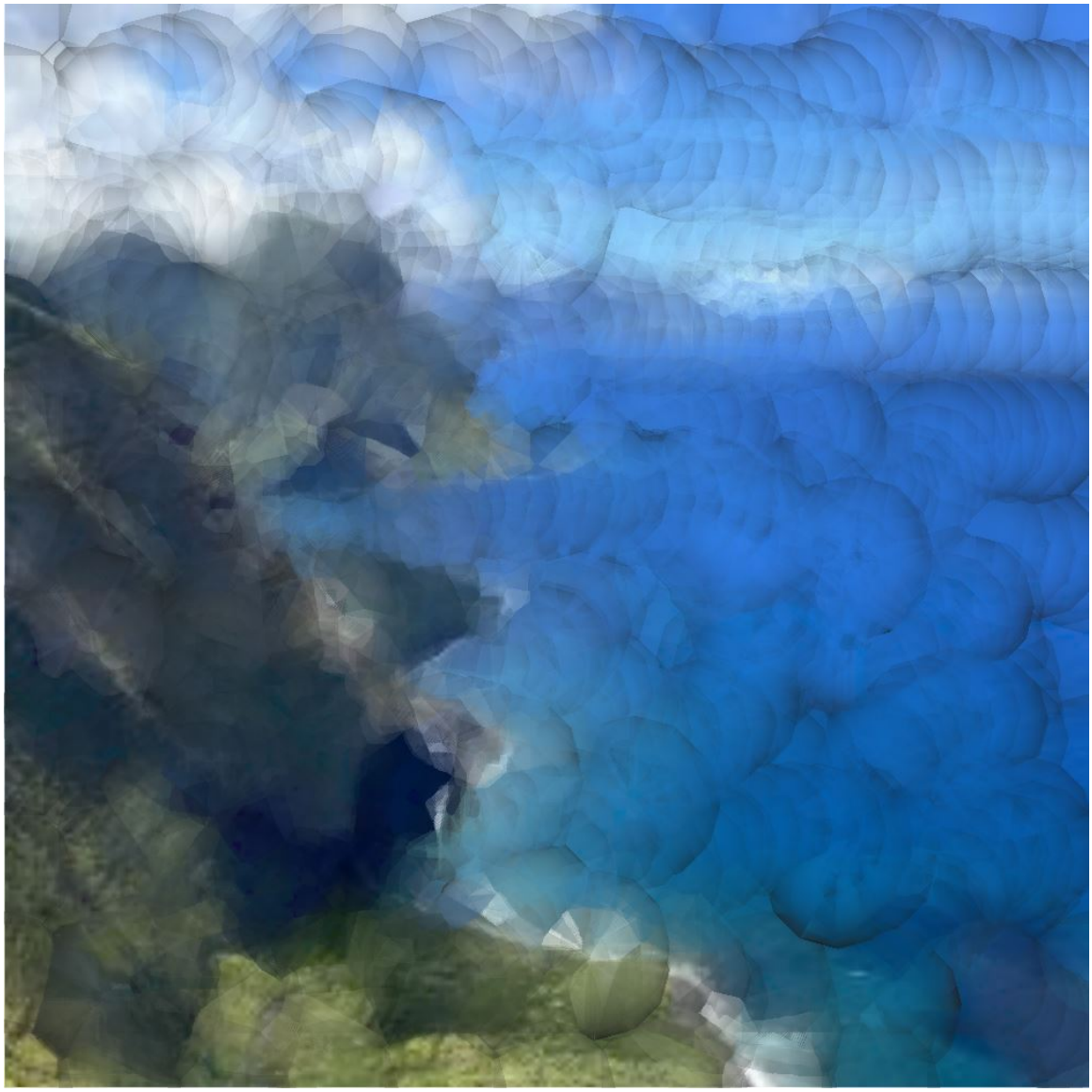


Figure 27: Watercolor on mountains

Combined with the definition of the brush stroke's shadows, the watercolor style creates a blotting effect on the image, which can be ideal on painterly renderings where the subject of the painting is sufficiently large and finer details of the image can be withheld [Figures 27, 28]. However, in painterly renderings whose input image is more complex [Figures 26, 29], the watercolor style can also have an interesting effect. While it may reduce the level of detail in the image, the watercolor styling contributes a more abstract effect to the painterly rendering, which may be desirable even in more detailed compositions.

Figure 28: Watercolor style on dog

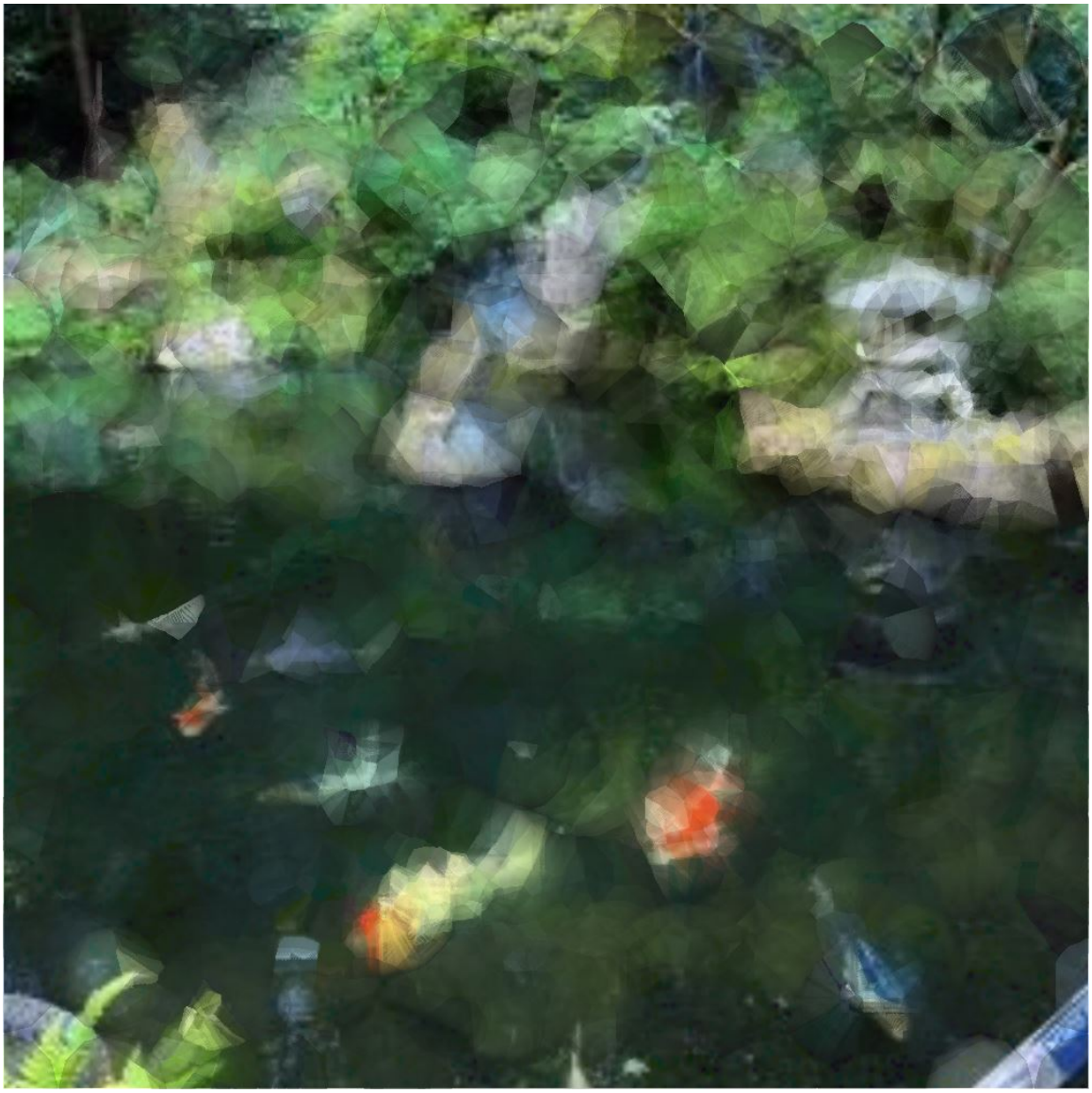


Figure 29: Watercolor style on fish pond

5.6 Expressionism

Figure 30: Expressionistic style on fish pond

For Expressionism [Figures 30, 33, 31, 32], the brush stroke width was decreased to add detail to the brush strokes. The brush stroke concentration was increased to ensure that the entire canvas is covered. The opacity was set to 50 percent, which was low enough to achieve a softer brush stroke appearance and high enough such that the vivid colors of the brush strokes were maintained. The brush stroke length was increased, which resulted in sufficiently long streamlines for the brush strokes to follow. The brush stroke spacing was

set to be close together, in order to maintain the level of detail in the original image. The brightness was also decreased from the -0.2 baseline in order to give the painterly rendering a darker, more vivid coloring. The precise parameters for this painterly rendering are as follows:

Brush width: 0.33
Brush stroke concentration: 0.5
Opacity: 0.5
Streamline step maximum: 750
Brightness: -0.3

The following pages are the Expressionistic painterly renderings adhering to these parameters.

Figure 31: Expressionism on mountains

Figure 32: Expressionistic style on dog

Figure 33: Expressionistic style on tulip field

6 Performance

A user is able to interact with this program by adjusting the provided style parameters for the user's input image. When the user imports an image into the application, a painterly rendering of the image is automatically applied with default values for each style parameter. Once the default values for each of the parameters is applied, the user can then adjust these values with a series of listboxes presented on the right hand side of the application. As the user increases or decreases each value– for instance, if a user wanted to change the brush stroke size– the application will respond somewhat instantly, and the new parameter will be automatically applied. This implementation is robust and effective; however, it takes a significantly greater amount of time to update the image with new style parameters. While reducing the streamline step size significantly reduces the update time, there is still a noticeable lag from one stylization to the next.

7 Conclusion

Our primary goal with this project was to expand upon painterly rendering seen in [1] and allow for more user customization within a simple graphical user interface. With this program, users are able to customize the painterly rendering produced from an image by altering brushstroke size, opacity, and concentration– all of which are updated in real time within the application. The underlying edge field computation is the foundation for developing the streamlines which are used to determine the direction of each brush stroke.

The application is able to update changes made to the style parameters in relative real time, with only a few seconds in between stylization updates. While previous works such as [2] and [1] implemented these core concepts in their work, we expanded upon this by implementing a simplistic user interface through which users could apply the painterly rendering. We also implemented unique painterly styles, and compared painterly results between input images with varying composition.

There are numerous ways in which this project can be expanded upon in the future. A few interesting ideas are as follows:

- **Expand file types for input images:** The current options for image input are limited to PPM, or Portable Pixmap format. While this image format does a fair enough job of maintaining the quality of the image, it is not as accessible to general audiences. To use this format, it is likely that a user would have to convert their JPEG or PNG images to PPM formats, which adds unnecessary complications to the painterly rendering process. File input types such as JPEG, PNG, or even GIF and MP4 could yield interesting results.

- **Add option to export images:** While the current program does not allow a user to export the image directly, this would be a convenient addition to the program to increase usability. Converting to different output file types– similar to the potential for different input file types– could make the program more practical for general users.

- **Add edge field preservation:** As briefly mentioned in Section 6, edge field smoothing works to reduce artifacts in the image edge fields, which improves the general flow of the brush strokes. However, smoothing tends to degrade the edge field and smooth out the prominent edges of an image. Edge preservation would help to smooth the edge field while maintaining a particular tolerance for the edges of objects in the painterly rendering. This has the potential to improve many aspects of the painterly rendering process, but would primarily aid in preserving the integrity of an image's edge field.

- **Accommodate more style parameters:** The style parameters of this project are fairly limited. The possibilities of painterly renderings could be significantly expanded with parameters such as contrast and saturation. Varying brush stroke tex-

tures, such as square or star-shaped brushes instead of circular, could also yield more dynamic and visually interesting painterly renderings.

- **Add options for adjusting lighting on the output:** The current iteration of this project implements delicate lighting on the brush strokes, with the light source pointing directly perpendicular to the canvas. Allowing for changes in the direction of the lighting– perhaps from slightly above or below the painting– could allow for even more user control, and possibly let users design more realistic paintings. Light intensity could also be an interesting style parameter for similar reasons.

- **Improved user interface:** The user interface for this program is simple in nature and fairly straightforward. While the program itself is usable, improved design and aesthetics of the GUI would undoubtedly improve user experience. A cleaner, more aesthetically pleasing user interface (i.e. more contrast between the menu background and the parameters, bolder color choices, smoother window edges, etc.) would be better suited to general audience, and would give the program a more professional feel.

- **Optimization with shaders:** This program could be significantly optimized through the use of fragment shaders. Using shaders to calculate the edge field and apply changes to the brush strokes has the potential to not only speed up the calculation process, but also decrease the wait time between style parameter updates.

Acknowledgements

Special thanks to Dr. Yue Zhang, as well as Xinwei Lin for help with streamline development and Jinta Zheng for edge field computation and debugging assistance.

References

[1] J. Hays and I. Essa. “Image and Video Based Painterly Animation”. In: *NPAR* (2004).

[2] A. Hertzmann. “Painterly Rendering with Curved Brush Strokes of Multiple Sizes”. In: (1998).

[3] Richard Keys. “Cubic Convolution Interpolation for Digital Image Processing”. In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 29.6 (1981), pp. 1153–1160. DOI: 10.1109/TASSP.1981.1163711.

[4] Mark J. Kilgard. *GLUI: A GLUT-Based User Interface Library (Version 2.36)*. [Online]. Available at: http://www.cs.unc.edu/~rademach/glui/. 1999.

[5] Mark J. Kilgard. *The OpenGL Utility Toolkit (GLUT) (Version 3.7)*. Software. [Online]. Available at: http://www.opengl.org/resources/libraries/glut/. 1996.

[6] Karl Pearson. “Contributions to the Mathematical Theory of Evolution. II. Skew Variation in Homogeneous Material”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 186.0 (1895), pp. 343–414. DOI: 10.1098/rsta.1895.0010.

[7] Aditya Ramesh, Alec Radford, and Ilya Sutskever. “DALL-E: Creating Images from Text”. In: *arXiv preprint arXiv:2102.12092* (2021).

[8] Irwin E. Sobel. “An Isotropic 3x3 Gradient Operator for Image Processing”. In: *Presented at the Stanford Artificial Project*. 1968.

[9] J. van Wijk. “Image Based Flow Visualization”. In: *ACM SIGGRAPH* (2002).

[10] Mason Woo et al. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[11] Ian T. Young. “Gaussian Smoothing”. In: *IEEE Transactions on Signal Processing* 43.9 (1995), pp. 2075–2078. DOI: 10.1109/78.413650.