# Artistic Renderings For Images and Video

Alexander D. Christoforides
Rensselaer Polytechnic Institute
chrisa4@rpi.edu

Andrew Aikens
Rensselaer Polytechnic Institute
aikena@rpi.edu

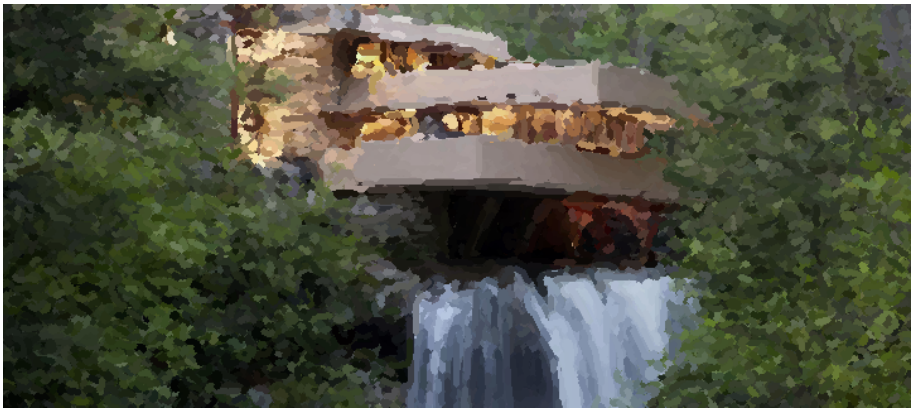

**Figure 1: Frank Lloyd Wright's Fallingwater rendered in the Impressionist style**

## ABSTRACT

A major focus of computer graphics research is the synthesis and production of realistic images from three-dimensional scenes and traditional two-dimensional images. Contrary to this trend, this project aims to tackle the issues of producing non-photorealistic renderings of two-dimensional images in various artistic styles and combining these renderings into a video where visual flickering is significantly reduced. A common problem that many identify when addressing the topic of creating artistic video renderings is that visual flickering occurs between frames, significantly affecting the quality of the synthesized result. To handle this, we propose implementing an algorithm based on difference-masking to efficiently and significantly reduce visual flickering between frames of the rendered video, thus improving the temporal coherency of the overall result.

To accomplish this, we need to devise a maintainable and extendable framework to transform the proposed algorithms into a functional application which can produce artistic renderings of both static images and videos with the assistance of 3rd-party modules (OpenCV, numpy, FFmpeg).

## KEYWORDS

computer graphics, non-photorealistic rendering, artistic styles

## 1 INTRODUCTION

This project encapsulates the implementation defined by the references below. Explained briefly, our implementation imitates the natural process an artist uses when creating paintings. This means that paint is applied to the canvas from larger to smaller detail. Creating a foundation initially, and increasing the level of detail for each new layer of paint. We put forth a reliable and maintainable open source software solution that transforms photo-realistic images into painted renderings. This paper provides an overview of the programming techniques, algorithms, and results from the implementation of the referenced work as well as a discussion on tasks that are tagged as future work.

## 2 PRIOR WORK

Haeberli's work in [4] lays the foundations for all work afterwards with respect to creating artistic non-photorealistic renderings of images. It proposes representing these rendered images as collections of filtered brush strokes which are then iteratively applied to generate an output image. To mimic the brush strokes that artists actually use when creating paintings, the paper introduces a parameterization which describes brush types, stroke sizes, and directions; these all affect the way a given stroke is generated and subsequently rendered to the canvas. By changing the parameterization, different artistic styles can be captured and mimicked in the resulting rendered image. Additionally, Haeberli describes

methods of introducing random color and positional perturbations to the strokes being drawn to achieve a more realistic effect which more closely imitates paintings by actual artists. By utilizing these random features, inherent texture can be added to the rendered image, simulating imperfections in the paint or canvas that an artist uses.

Aaron Hertzmann iterated upon Haeberli's work in [5]. In this work, he expands upon Haeberli's proposed parameterization and defines a broader set of parameters which capture the overall characteristics of the generated brush strokes in the rendered image. In the published work, the parameterizations for a few different artistic styles are delineated. Based on these, Hertzmann identifies the benefits of representing the brush strokes as Cubic B-Splines and outlines a few procedures to generate the brush strokes and render them to the canvas. An important note they make is that finding new parameterizations is quite easy, but finding parameterizations which produce rendered images in a style similar to actual art styles is exceedingly difficult. Based on this fact, we decided not to attempt to identify unique parameterizations which mimic other artistic styles.

Hertzmann further synthesized his prior work in his thesis [6] and delves into more detail regarding the algorithms, procedures, and limitations of the methods that he proposes. He revisited the parameterizations that he defined in [5] and defined them in a more concise way which better interfaces with the other work he displays in the paper. Most notably, he proposes a method of extending his image rendering algorithms to additionally render videos in artistic styles as well. One major concern that many have when rendering videos in this way is temporal coherence, and Hertzmann subsequently presents a method of performing this coherence computation in a frame-to-frame setting; holistic video temporal coherence was not explored and discussed.

## 3 PAINTING STYLES REFRESHER

To better understand the results of our work, below you can find a brief aside on the different types of artistic styles represented through our review of [2], [1], and [3]. Each one is characteristically different from one another and is described by a unique parameterization within the program logic.

### 3.1 Impressionist

The impressionist painting style can be most frequently found in the works of art created by artists during the 19th century. When observing a painting of this type, one can notice small, thin, but visible brush strokes which cover the canvas and coalesce into a coherent image. Typically the composition of these works can be summarized as open and additionally they highlight the accurate depiction of light and its subsequent effects on objects. An example of a painting in this style can be found in figure 2.

**Figure 2: Claude Monet, *Impression, Sunrise*, 1872.**

### 3.2 Expressionist

Unlike the impressionist style, the expressionist style is more of a modern movement in art which initially began in the poetical setting. The goal of these types of artworks is to capture and present the world from a subjective point of view; they typically distort the reality of whatever they are portraying in a radical way to evoke a desired emotional effect or idea in the viewer. The individual strokes in these types of paintings are characteristically more exaggerated when compared to those of the impressionist style. Figure 3 displays *The Scream*, a painting in this style by Edvard Munch.

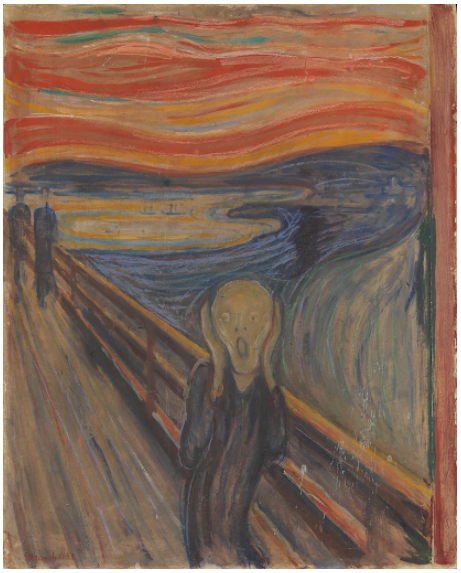

**Figure 3: Edvard Munch, *The Scream*, 1893.**

### 3.3 Pointillist

When compared to the previous two art styles mentioned, the pointillist style stands out as completely disparate. Originally used by art critics to ridicule the work of these artists, the pointillist style presents subjects through the use of carefully placed points on the canvas. Unlike the other art styles which more clearly defines the objects in the scene with the curvature and size of the brush strokes, pointillist images are exclusively comprised of color dots (sometimes restricted to a certain palette). The artist relies heavily on the ability of the viewer to blend the positioned color into a coherent image. Notably, some of Andy Warhol's early works can be characterized as being made in this style. Maximilien Luce's *Morning, Interior* is shown in figure 4 and was painted using the pointillist technique.

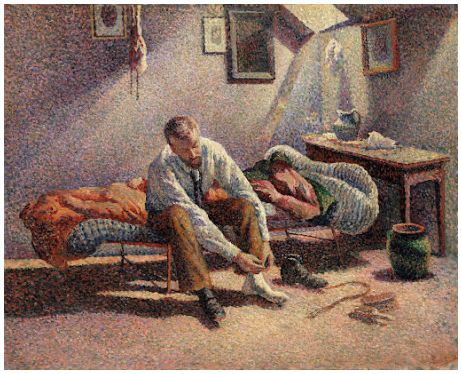

**Figure 4: Maximilien Luce, *Morning, Interior*, 1890.**

## 4 DATA STRUCTURES

To facilitate the artistic rendering of both images and videos, a robust code structure is required to manage all of the data that is being provided to the program. Additionally, information regarding what type of style to render, where to output the render, and the actual process of rendering the image or video is all required, and in order for this application to be modular, a carefully planned architecture is required. This is described further below.

### 4.1 Image

The main purpose of the `Image` class is to abstract away the implementation details of both OpenCV and numpy so that the user of this application doesn't need to know anything about them. Combining the intricacies of both of those libraries in addition to the one we developed would be overbearing, so we thought it fit to alleviate some work from the user.

The `Image` class achieves this by acting as a wrapper to an OpenCV/numpy image representation. It additionally has various methods for accessing and setting data in the image in addition to being able to compute and return a Gaussian blur, horizontal derivative, and vertical derivative of the image. Through this class, we efficiently manage the data regarding images and are able to perform operations which the render algorithm requires with ease.

### 4.2 BrushStroke

An integral part of being able to render input images and video in an artistic, non-photorealistic style is being able to represent the individual brush strokes which make up the image. Because the individual strokes need to contain information regarding control points, previous directions, and color, we decided to develop the `BrushStroke` class. It holds information regarding the given stroke's radius, color, control points, and previous directions. These are generated in the `paintStroke()` algorithm and are returned to the `paint()` algorithm to then render after computing all the given brush strokes for a layer.

`BrushStroke` objects are eventually passed to the `renderStroke()` function which handles the interpolation of the control points and puts color in a given radius on the canvas.

### 4.3 RenderableImage

The `RenderableImage` class is responsible for maintaining two `Image` objects, one for the source (reference) image and one for the destination (canvas to paint on). Additionally, this class contains all the relevant functionality to actually take a given image and render it artistically in a non-photorealistic way to the canvas image. Instead of requiring a user to specify all of the parameters manually to render the image or video, we hide these details and only require users to specify what type of style they would like using the name of the art style (e.g., "impressionist", "expressionist", "pointillist"). If a user would like to add a new style, then all they have to do is add it add the top of the `RenderableImage` class as a new renderable style.

Additionally, the `RenderableImage` class has the capability of getting and setting both of the stored images. If a user would like to store the image, then all they would have to do is get the destination and call the `save()` function on it. By hiding all of the implementation details of the rendering process, we eliminate all non-essential information that a user needs in order to use our application. Similarly, since the rendering function is part of the `RenderableImage` class, a person can specify a new rendering function and call that if they so desired; this further exemplifies the modularity and flexibility of the system that we have developed.

## 4.4 Video

To facilitate the artistic rendering of video in addition to images, we required another data structure to help manage all of the rendered frames for the given video. The `Video` class satisfies these needs as it does this in addition to output all of the rendered frames as they are being processed. Videos and images are actually quite similar, but one major difference is the fact that a video is comprised of many, many renderable images instead of just one. The `Video` class maintains a list of `RenderableImages` and operates on them when the user instructs the program to create a rendered video. OpenCV is utilized to split the input video into individual frames, and based on these, we construct the source-destination pairs required to construct the subsequent `RenderableImages` that will eventually make up the video.

After a video has been read in, then all a user has to do is call the `render()` function and specify the style of rendering they desire the output to be. Similar to all the structures above, we aimed to abstract away the minutiae of OpenCV and numpy, and this further exemplifies how we have done this successfully.

# 5 ALGORITHMS

Rendering the non-photorealistic images of the input image or video is primarily accomplished through the use of two major functions: `paint()` and `paintStroke()`. The `paint()` function is responsible for traversing the image, computing the error between the source and the current painting, and calling `paintStroke()` which generates `BrushStrokes` and returns them back to `paint()`. After all of the strokes are computed for a given brush size, `paint()` then calls `renderStroke()` which simply interpolates the brush stroke and outputs color where the stroke should be drawn on the output image. After both of these functions have terminated, the `pinkCorrection()` routine is ran to eliminate visual pink artifacts due to the algorithms not painting over certain areas in the image.

## 5.1 Parameters

Each style that we are able to render is characterized by a parameterization which captures the nuances and features of given brush strokes which typically make up paintings of the specified style.

*5.1.1 Approximation Threshold (T).* This parameter defines how similar the painting and the source must be. If a higher value of $T$ is specified, then a more coarse approximation of the image is painted.

*5.1.2 Brushes (b).* The list of brushes specifies the different radii that will be used to generate brush strokes for the image.

*5.1.3 Curvature Filter ($f_c$).* The curvature filter parameter specifies how much to constrain or emphasize the stroke curvatures.

*5.1.4 Gaussian Kernel Size (Blur Factor) ($f_\sigma$).* Smaller values for $f_\sigma$ produce more noise in the rendered image while larger values generate less noise in the rendered image.

*5.1.5 Minimum Stroke Length.* This value is the smallest number of control points that a given stroke is allowed to have before being returned to the `paint()` function.

*5.1.6 Maximum Stroke Length.* This value is the largest number of control points that a given stroke is allowed to have.

*5.1.7 Temporal Error Threshold ($T_v$).* In Hertzmann's thesis [6], a difference masking technique to improve temporal coherence when rendering videos is defined. This parameter defines how much difference is allowed from frame to frame in order to justify creating a new paint stroke within a given region.

## 5.2 `paint()`

The `paint()` algorithm is responsible for determining if a brush stroke should be applied to a given area and to what point the brush stroke origin should be. It breaks the image canvas up into a grid, with each grid block the size of the provided radii. If a video is being rendered, a difference masking threshold is also applied to determine if paint should be applied in a given grid block.

For each brush size, provided by the painting style type, a Gaussian blurring of the source image is performed. The Gaussian blurring kernel is calculated to be the size of $F_\sigma * Radii$ where $F_\sigma$ is defined also by the painting style laid out in the Hertzmann publications [5] and [6]. This Gaussian blurred resulting image is then used to determine the difference between the current canvas at a given iteration and of the source. This error is calculated by taking the euclidean distance of the grid block region (which is defined as M in the equation below). We split the comparison images into their corresponding RGB values and then perform the euclidean distance for each pixel. This can be improved by using Numpy operations in order to get the specific channels without having to create copies of the data.

$$\sum_{x,y \in M} \sqrt{(I1_r - I2_r)^2 + (I1_b - I2_b)^2 + (I1_g - I2_g)^2}$$

We use this computed sum to compare against an *Approximation Threshold* as defined in section 5.1.1. This value is pre-defined for every paint style as provided by Hertzmann [6]. This comparison classifies whether a brush stroke should be generated within this grid block. If the error calculated from the summation above is greater than the *Approximation*

*Threshold* then the error for the given grid block is too large and should be adjusted with a new paint stroke.

For videos, as mentioned briefly above, we have an additional check to improve temporal coherence. This check uses the same logic as euclidean error calculation but instead of comparing against the Gaussian blurred source, we compare against the previous frame using the *Temporal Error Threshold* ($T_v$). This lessens the amount of paint that is allowed to be painted from frame to frame, ensuring that only large differences are painted.

For grid blocks that pass the *Approximation Threshold* test we determine which point to begin a brush stroke by determining the pixel within the grid block that has the largest euclidean distance (representing error in the current canvas). The `paintStroke()` procedure is then called using this point as a starting reference.

## 5.3 `paintStroke()`

The `paintStroke()` algorithm is responsible for generating brush strokes which are later employed by `paint()` and `renderStroke()` to output them to the canvas in a randomized order.

This algorithm requires the specification of an initial point $(x_0, y_0)$ so that it is able to compute what color to make the stroke and where to start it. It is important to note that the color is computed from the blurred reference image and not the actual source image, so the identified color may not always be exactly the color present in the source image. After adding the initial point to the list of control points in the `BrushStroke` object, it then computes both the horizontal and vertical derivatives with respect the luminance channel of the blurred reference image. Then the function performs an iteration from $i = 1 \ldots maxStrokeLength$ and computes the next control points for the given brush.

*5.3.1 Computing Additional Control Points.* To compute additional control points, the algorithm looks at the colors of the current position in the canvas and reference image. If we detect that the euclidean distance between the current pixel in the blurred reference and the pixel in the canvas is less than that of the color at the initial control point and the current position in the blurred reference image and if we already have enough valid control points in the brush, then we just return the current brush.

If this is not the case, then we must look at the derivatives of the image to generate a new direction to move in to then generate a new point. We get the horizontal gradient value ($gx$) and vertical gradient value ($gy$) and then take the normal of this to be our $dx, dy$. If we see that we aren't moving enough ($dx_{last} * dx + y_{last} * dy < 0$), then we set the current $dx$ and $dy$ to the opposite direction $(-dx, -dy)$. Once we established a new direction to move in, we then compute a move amount by filtering the brush stroke according to the specified curvature. By doing this and then normalizing these results, we can use them in

$$x, y = x + R * dx, y + R * dy$$

to determine the point at which we want to generate the next control point. At this point, we record the last directions moved $(dx, dy)$ and add the generated point to the list of control points that the current `BrushStroke` object holds.

If we see that our derivative at a given point is 0 or that we are trying to generate a control point out of bounds, we terminate the function and return the generated `BrushStroke`. Additionally, if we were successfully able to generate *maxStrokeLength* control points, then we just return the brush stroke then as well.

## 5.4 `pinkCorrection()`

**Figure 5: Rendered image with pink canvas artifacts.**

This routine is ran after the entire `paint()` function has finished executing. It's primary purpose is to remove visual pink artifacts from the final rendered image. These artifacts exist in the first place because sometimes the algorithms will not paint over certain areas in the image, thus leaving a default canvas color in its place (we chose pink to maximize contrast). Figure 6 shows how the algorithm works. Essentially, we do a final pass over the final image and look for all pixels that are the default background color. If we find one of these pixels, we then average the colors of all non-background neighbor pixels to compute a new color for the identified pixel. This is only ran on the final, rendered image, so we only incur the $O(n^2)$ runtime penalty once.

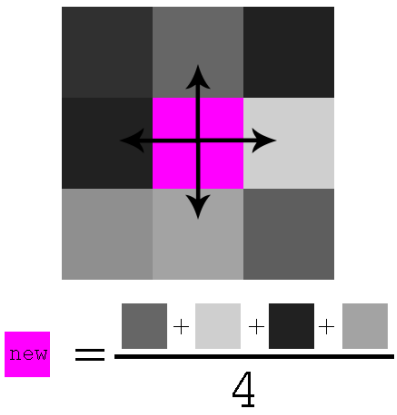


**Figure 6: Visual representation of the pink correction.**

## 6 RENDERING VIDEOS

Rendering videos is very similar to rendering still photos. For the first frame in the sequence we render on a blank canvas. For every following frame we use the "Paint over" method as defined by Hertzmann in his thesis [6]. This method takes each frame, where the canvas for the current frame is the resulting rendering from the previous. This method of rendering from frame to frame slightly increases the coherence of the sequence. To further increase temporal coherence between frames we perform a difference masking as well. This method, similar to the error difference calculation, as discussed in section 5.2 takes the difference between the current painted canvas and the previous frame. This measure effectively limits the amount of paint that can be applied to a region if the region from frame to frame is similar. Although, this won't fully eliminate the effect of "flickering" between frames as applying paint will create imperfections.

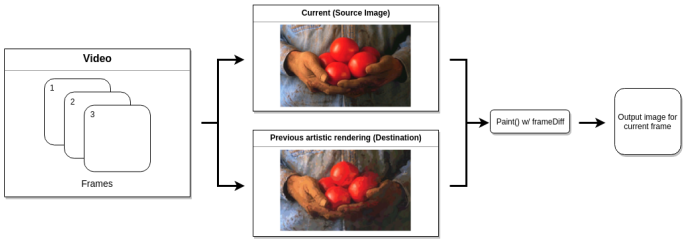


**Figure 7: Flow of rendering a video.**

## 7 CHALLENGES

### 7.1 Unclear implementation details

*7.1.1 Different procedures defined by same author.* In Hertzmann's papers [5], [6] implementation details for `paint()` and `paintStroke()` conflicted. These papers put forth different procedures to render the painterly images. We first implemented `paintStrokes()` from Hertzmann's thesis [6], but soon realized the results were not fully correct. After some debugging we ended up implementing the `paintStroke()` procedure from his other paper [5] which provided clearer details and correct results.

**Figure 8: Buggy rendering of tomatoes in hand as shown in figure 8.**

*7.1.2 Unspecified parameter values (specifically $T_v$).* The threshold for difference masking between frames is undefined (refer to section 5.1.7 for more discussion on this value). We had to investigate and test with different values for the threshold. Long video rendering times made this task difficult.

### 7.2 Difficult to debug

*7.2.1 Long rendering times.* Long render times made it difficult to identify bugs from code changes. This became relevant, especially when rendering video, as our system renders each frame (more discussion on rendering times in section 9).

*7.2.2 Derivative (gradient) image calculation.* For a long time, we were generating images where all brush strokes were moving to the top right portion of the image. As shown in the figure below, this prevented regions of the canvas from being painted. In order to solve this we identified an area where the absolute value of the gradient was being factored into the calculation. After removing this, the generated results were significantly better.

*7.2.3 Error calculation between canvas and Gaussian.* The process of determining whether a brush stroke should be applied in a region is a defined by the summation of difference between the current canvas and a Gaussian blurring on the source image (more discussion on this in section 5.2). Initially we defined the canvas background as white. When taking the difference between a white canvas and an area in the source image that also is white, the system would identify

this area as being similar and would not paint. This resulted in empty areas in our paintings that looked very unnatural. In order to fix this we defined the canvas to be a color of very high contrast with most of the images that we were running *rgb*: (255, 0, 255) or hot pink. This was predefined as a result of our observations.

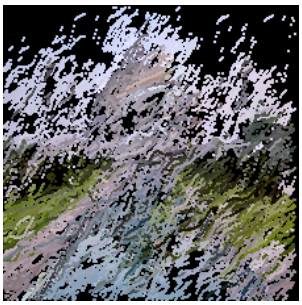

**Figure 9: Incorrect gradient calculation.**

## 8 ANALYSIS

**Figure 10: Comparison between our results (left) to Hertzmann's [6] (right)**

Shown above is a comparison between our final impressionist results and Hertzmann's [6] results for the same corresponding source image. The comparison shown above is of each paint layer (brush sizes of 8, 4, 2). A direct comparison prior to the final layer is difficult to make as brush strokes are applied randomly. Our final layer results prove to show the equivalent level of detail, if not more. As mentioned in future work, slight differences between our final results and Hertzmann's can be attributed to opacity and color jitter parameters. Other differences between our implementation and Hertmann's is the color that we use for our canvas. The canvas color that we observed to generate the best result was hot pink. Hertzmann doesn't specify a canvas color in either [5] or [6]. In addition to this, we struggled with generating a correct cubic spline for the brush path. Since all of the control points are within the given radii for that stroke we simply painted along the brush strokes connecting each with a line. This proved to give very accurate results without having to map the control points to a corresponding spline. We initially investigated using SCI-PY to help map the points along the spline but soon ran into issues with constraints laid out by the library for the corresponding functions.

## 9 PERFORMANCE DISCUSSION

**Table 1: Runtimes on Varying-sized Input Images (Impressionist Style) on Intel i7-4790k**

| Input Image Size | Runtime (seconds) |
|---|---|
| 100x100 | 1.188 |
| 200x200 | 9.094 |
| 300x300 | 34.594 |
| 400x400 | 126.125 |
| 500x500 | 279.979 |

Run time for our implementation grows significantly as input size increases. This is directly due to the brush size radii that are required for each painting style. In impressionist, for example, brush sizes are defined to be 8, 4, 2. Brush sizes 8 and 4 both render in relatively quick time. On the other hand, when we get to the smallest brush size render time begins to slow as the number of strokes that are generated in the final pass is exponentially larger. This is a result of the grid block size decreasing in size for each radii, and generating multiple control points for each stroke. This is especially relevant in the *Pointillist* painting style, where we have a fixed small brush size that needs to be rendered nearly multiple times per area region. For video, this is multiplied by the number of frames that must be rendered. A video that is 12 seconds, 30 frames per second, would require that the algorithms be ran 360 times. This is then multiplied by the resulting image size cost per frame.

## 10 FUTURE WORK

While our current implementation is able to produce quite pleasing rendered images and video, it is still limited with respect to the number of effects that it is able to reproduce.

At the moment our solution does not implement brush stroke opacity and therefore cannot achieve layering effects that many artists use when painting an image. The lack of this could also explain why we experience slight differences from our outputs and the outputs from Hertzmann's work

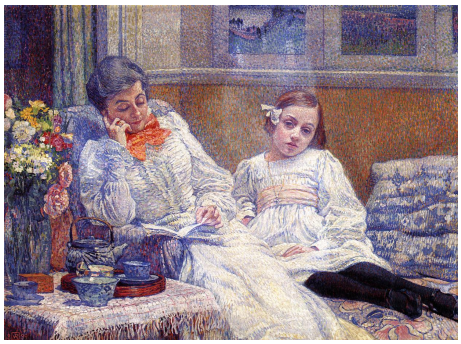

**Figure 11: Example of pointillist randomized color jitter present in actual artwork. Théo Van Rysselberghe, *His wife Maria and daughter Elizabeth*, 1899.**

in [6]. We additionally do not implement the randomized color jitter effect described by Haeberli in [4]. This is very noticeable in our results for pointillist images since all of the points we generate are solid colors. With respect to how artists actually paint in the pointillist style, this is not how it actually works; sometimes there are perturbations in the quality of the paint or canvas which leads to a different color being placed at the specified point. This can be seen in figure 11. Besides this, further work can be done to add an extra parameter which constrains the total number of allowed colors in the image. It is not feasible or practical to make every new point on the canvas a different color, thus artists reuse colors in creative ways to achieve the same effect without utilizing an extraordinary amount of colors. By including these features, there is a large possibility that the quality of our rendered images and video would be much more similar to those actually painted by artists in the real world on physical mediums.

Moreover, a better strategy could be researched to select a more appropriate background color for the initial blank canvas. We chose pink since we believed it to cause the most contrast between the source and canvas, but in reality, this is not always the case. A pre-processing step could occur on the image and a best suited color could be chosen which provably causes the highest level of contrast to occur between the source and blank canvas. This could be done by identifying the color palette of the image and calculate or use online resources to determine colors of highest contrast.

With respect to temporal coherence, our method reduces a minimal amount of noise; the video is largely unchanged and noticeable jittering in the video is still present. While it is impossible to remove all jittering, we believe that further improvements can be made to the temporal coherence computations which would generally increase the noise present between frames. Another thing to consider is the fact that our method only considers frame-to-frame coherence and not holistic video coherence from start to end. A natural extension to this algorithm would be to devise a strategy for trying to achieve coherency across the entire input space in the video rendering setting.

Additionally, further work can be put into general optimizations which would make the rendering process significantly faster. As described in Hertzmann's [6], we could utilize a summed-area table to reduce the amount of computation necessary to detect errors between the source and canvas being painted to. At the moment, rendering large images ($\sim 1000x1000$) or videos takes an inordinate amount of time, so this optimization could truly be worth while so that larger inputs can be processed in a more efficient manner.

## 11 PROJECT ROLES

The following section describes what each member of the group contributed to the project.

### 11.1 Alexander D. Christoforides

**Figure 12: Impressionist rendering of Alexander D. Christoforides**

Alexander D. Christoforides was responsible for developing the required data structures (`Image`, `BrushStroke`, `RenderableImage`, `Video`) to facilitate the efficient rendering of both images and video. Additionally, Alexander wrote the logic for the `paintStroke()` routine and developed the functions for rendering video as well instead of just images. Lastly, he performed a much-needed refactor of the program so that it matched the overall architecture that both he and Andrew initially devised.

## 11.2 Andrew Aikens

**Figure 13: Impressionist rendering of Andrew Aikens**

Andrew Aikens was responsible for the implementation of the *paint*() routine and developed functions to manipulate images that were required for `paint()` and `paintStroke()`. These include: Luminance, Gaussian blurring, Error difference, Difference Masking for video temporal coherence, Gradient Calculation, and code optimization. Andrew also devised code to take brush strokes that were generated and randomly apply them to the canvas per layer.

## 11.3 Both

Both Alexander and Andrew spent countless hours debugging the program and together devised the methodology to perform the final-pass correction on the rendered image to remove artifacts from the algorithms not painting over certain parts of the image. Additionally, they both put together the presentation and final paper as well.

## ACKNOWLEDGMENTS

To Barb Cutler and Evan Macius for providing the necessary support and input required to make this project a success.

## REFERENCES

[1] 2019. Expressionism. https://en.wikipedia.org/wiki/Expressionism
[2] 2019. Impressionism. https://en.wikipedia.org/wiki/Impressionism
[3] 2019. Pointillism. https://en.wikipedia.org/wiki/Pointillism
[4] Paul Haeberli. 1990. Paint by numbers: Abstract image representations. In *ACM SIGGRAPH computer graphics*, Vol. 24. ACM, 207–214.
[5] Aaron Hertzmann. 1998. Painterly rendering with curved brush strokes of multiple sizes. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. ACM, 453–460.
[6] Aaron Philip Hertzmann. 2001. *Algorithms for rendering in artistic styles*. Ph.D. Dissertation. New York University, Graduate School of Arts and Science.

**Figure 14: Impressionist rendering of Licorice and Nutmeg**

**Figure 15: Each paint layer with pink correction**

## 11.4 Impressionist

**Figure 16: Impressionist rendering results**

## 11.5 Pointillist

**Figure 17: Pointillist rendering results**

## 11.6 Expressionist

**Figure 18: Expressionist rendering results**