A Greedy Algorithm for Generative String Art

Baptiste Demoussel, Caroline Larboulette, Ravi Dattatreya

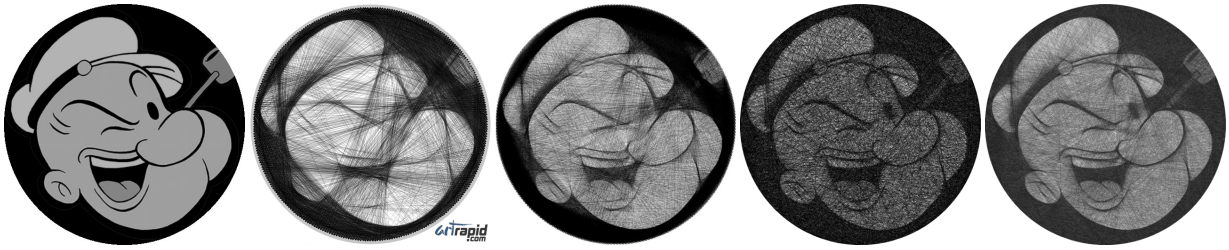
A Greedy Algorithm for Generative String Art

Baptiste Demoussel[1], Caroline Larboulette[2] and Ravi Dattatreya[3]

[1]ENS Rennes, France; baptiste.demoussel@ens-rennes.fr
[2]IRISA, University of South Brittany, France; caroline.larboulette@univ-ubs.fr
[3]Neonyx Technology, USA; ravi@neonyx.com

Abstract

Computational String Art is a method of creating a non-photorealistic rendering using lines drawn between pairs of pins. Given a set of pins and an image to render, we propose a greedy algorithm to determine a good set of lines to imitate the input. Implementation details are discussed, as well as some possible optimization to speed up the algorithm. We also propose some improvements to the basic greedy approach, provide a stopping condition and allow for the use of non-opaque strings. We illustrate our work with various results.



(a) (b) (c) (d) (e)

Figure 1: *(a) Original Popeye image. (b) Art by ArtRapid [2](*3993 *strings, online simulation). (c) Art by Birsak et al. [4] (simulation). (d) Our result with* 3990 *opaque black strings (*147 *seconds, RMS* = 0.218*). (e) Our result with* 9570 *black strings of opacity* 30% *(*354 *seconds, RMS* = 0.158*). Both our results used* 1000 *pins, strings of width* 1/4 *pixel, line sets of size* 5000*,* 30 *lines per set.*

Introduction

Generative or algorithmic art aims to create artwork from scratch (e.g., landscape paintings in the style of Bob Ross [10]). Non-photorealistic rendering is a field of Computer Graphics that focuses on digital art and expressiveness rather than realism. An example is String Art that aims at creating an abstraction of a given input image or photograph. More recently, this work has been extended to the stylization of videos in addition to still images (see for example the work of Delanoy et al. [7]) and many algorithms now make use of Artificial Intelligence techniques such as Neural Networks (e.g., Jang et al. [9] generate a caricature from an input photograph using Deep Learning algorithms).

String Art discussed in the literature as well as demonstrated on the Internet aims at creating an artwork by stretching a single thread, usually of black color, around pins or nails. It finds its origin in the 19th century, as an activity that was invented to help children get familiar with mathematical ideas. Computational String Art emerged around 2016, as a tool to help artists find a good path to follow in order to best produce similar images, while keeping the metaphor of the original idea that consists in using a single thread and pins arranged in a circle. From a Computer Graphics perspective, it can be seen as a method to automatically generate a stylization of an input picture [4]. Early algorithms followed a greedy approach where, at each step, the best possible line (i.e. string) to be drawn is selected. Others have used different optimization criteria [4]. In

this paper, we focus on the heuristics necessary to drive the greedy approach, to which we add variations: (i) we highlight the importance of the choice of a good heuristic to pick the next line, given a current state of the program; (ii) we propose a way to optimize computation time when choosing a line to be drawn; (iii) we propose a stopping criteria for the algorithm; (iv) we explore the advantages of using non-opaque grey strings and background; and finally (v) we show the benefits of using an importance map to guide the result.

The String Art Problem

String Art is produced by stretching a string from one pin to another, selected from a set of pins, usually placed in circle on the boundary of the artwork. Each string thus goes from one pin to another, and the superimposition of the strings creates contrast at appropriate places and thus forms an image. If done well, it can resemble a given image taken as input, but the process of getting a good looking visual is non-trivial if done by hand. It is a tedious and precise work of patience, as it usually takes between 2500 and 5000 strings to stretch to get a meaningful result. String artists therefore find computer programs very helpful.

Earlier algorithms reported in the literature use strings stretched into straight lines, each string being black and opaque, and pins placed in circle. Some variations use colored strings [3] or place additional pins inside the canvas [12]. We propose to extend those ideas by removing the thread constraint and moving to pen plotter prototyping by computing a set of pairs of pins as an output. In that case, two steps are missing to be able to physically reproduce the artwork: (i) merge string segments into a single thread of string, and (ii) take into account the width of the pin (reaching a pin and leaving a pin correspond to two different locations on the canvas). The first problem can be solved by an Eulerian graph extraction such as proposed in [4], that connects all segments into a single polyline. This step does not change the artwork in any way. The second problem has also been addressed in [4] but is not the focus of this paper.

The inputs/outputs to a computational String Art problem are:

Input

An input image (here in grayscale) represented by a grid of pixels.
A set of pins on a 2D surface.

Output

A set of pairs of pins, so that stretching a string between such pin pairs would produce a visual that imitates the input image.
A grey level assigned to each generated string in the range 0-255 (0 is black and 255 is white), 0 by default.

The main idea behind the algorithm is to (1) look at all possible lines that can be drawn, (2) select the darkest line, (3) draw this line and finally (4) subtract this line from the input image. This process is repeated for each line (or string) – that is, if 2500 strings are available to be used, then we run this process 2500 times. Note, however, that if the addition of a line adds no value to the image, or if the stopping criteria (see section *Stopping Condition* below) have been reached, then the algorithm stops.

Obviously, in practice (for debugging and testing), such output is very often simply rendered as a grid of pixels to be displayed on a raster screen, instead of being turned into a real-life concrete artwork. For this reason, as well as for the sake of simplicity, the accuracy of the produced output is often evaluated as the sum of the differences between each pixel in both images (porting the resolution of the output to the resolution of the input if necessary), or the sum of the squares of the differences (this being a squared RMS). As we work with pixel grids instead of real strings, we will refer to the thread either as strings or lines between pairs of pins in the remainder of this paper.

Greedy Algorithms

We believe that a satisfying solution to this problem can be approximated by a greedy algorithm. A greedy algorithm tries to choose the best move available at each step. It is short sighted as it does not consider the consequences of the succession of all the moves it makes together. The relevancy of such algorithms comes from their execution speed that is high, at the cost of not always providing an optimal output. An example of such algorithm is the one that, to solve an instance of the traveling salesman problem, decides that the next step to append to the current path is the closest city that is not in the path.

The way to process one such step can be called a heuristic, and, as greedy algorithms are not always optimal and there is rarely a need for a clear definition of *best move*, finding a good heuristic can turn out to be non-trivial. In the case of String Art it consists in selecting a string to be drawn among all possible pairs of pins (see the *Line Selection Heuristic* section).

Previous Work

Very little work exists in the form of published papers that discuss this art form because it developed by commercial artists with no incentive for sharing. String art is known to originate at the end of the 19th century. It was famously invented by Mary Everest Boole as *curve stitch* activities to make mathematical ideas more accessible to children [5]. It was then popularised as a decorative craft in the late 1960s through kits and books. In 2016, the artist Petros Vrellis was the first to propose an algorithm to create computational string art [13]. This algorithm has then been extended by Demaine et al. [8] in 2017 to the generation of string art font. In 2019, the artist Ani Abakumova and her husband, Andrey Abakumova, a mathematician, developed an algorithm to create string art with colored strings [1]. Commercial versions also propose the use of colored strings [3] or place additional pins inside the canvas [12]. Their methods and processes are not public.

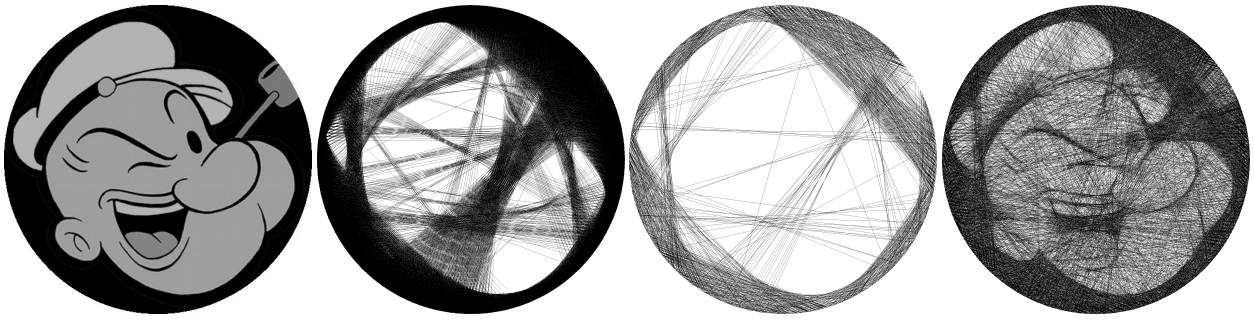
The first scientific paper on computational string art appears to have been published in 2018. In their paper, Birsak et al. [4] propose a formal treatment of the problem and present results produced by a greedy algorithm that is allowed to remove strings added in any previous steps. The fact that any previously added string can be removed at any point improves the algorithm, hence the quality of the output. However, it comes at the cost of speed, as the average duration of execution on a standard laptop is over 2 hours when it takes less than a minute for commercial software or for our implementation, using the same input images. Indeed, adding a new string between two pins is fast as it consists in drawing the new string on top of the current image, but removing an arbitrary string is not as it requires to store and render all the other strings again.

Greedy Approach in Practice

Our approach, unlike what is proposed in [4], does not allow removal of lines drawn; thus every line that is generated is permanently engraved and will be present in the rendering of the image. Our experience is that even though this can clearly be sub-optimal, this trade-off does not seem to be unacceptable and we have obtained quality results. This limitation allows for the execution to complete in a reasonable amount of time. In order to optimize the quality of the result and the speed of the algorithm, heuristics are necessary to (i) select the best possible line, (ii) to stop the algorithm when it reaches some criterion rather than fixing in advance the total amount of lines, and (iii) to reduce execution time by allowing multiple line choices per iteration.

Line Selection Heuristic

Let us consider two images: the string image (the current picture) and the target image (the original picture). Let us also define an error function between the two images. There are many possible definitions for this error function, such as the sum of the differences in grayscale of each pixel, or the sum of the squared grayscale



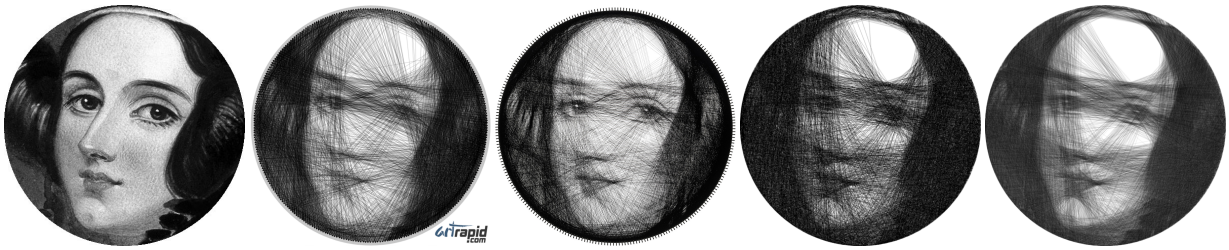
(a) (b) (c) (d)

Figure 2: *(a) Original Popeye image. (b) Our result minimizing the error at each iteration. (c) Our result picking the darkest line on the target image from which each drawn line is erased after* 1000 *iterations. (d) Final result of (c). All results use opaque black strings.*

differences (RMS) and others.

As part of the greedy algorithm, the first step is to create a set of strings to choose from. We can select the group of strings over which we optimize in many ways: e.g., (1) select a pin (at random or in sequence) and look at all the strings originating from it, or, (2) select a number of pairs of pins at random and look at the strings connecting them.

The second step of the algorithm is to select the best string among the set. A first possibility is to select the string that minimizes the selected error function (see Figure 2 (b)). A second possibility is to select the darkest line in the set (see Figure 2 (d)). In this case, it is important to delete the selected line from the target image. These are not equivalent, as the darkest possible line in a very bright image may increase the error by darkening large areas that are supposed to be white. These two steps form the line selection heuristic.



(a) (b) (c) (d) (e)

Figure 3: *(a) Original Ada image. (b) Art by ArtRapid [2](*3994 *strings, online simulation). (c) Art by Birsak et al. [4] (photography of the produced artwork created using an importance map). (d) Our result with* 3270 *opaque black strings (*114 *seconds, RMS* = 0.198*). (e) Our result with* 8130 *black strings of opacity* 30% *(*275 *seconds, RMS* = 0.140*). Both our results used* 1000 *pins, strings of width* 1/4 *pixel, line sets of size* 5000*,* 30 *lines per set.*

Stopping Condition

The three steps of the algorithm are run iteratively. Finally a stopping condition is used to terminate the algorithm. We have several choices for the stopping condition. The simplest way is to stop when a given number of strings have been drawn. A second way is to stop when the addition of a new string does not reduce the total error function (stagnates or even increases the error). Finally, we can stop when the total error has fallen to an acceptable level. Of course, the error function can be chosen to be as detailed and complex as desired including grayscale values, color values, RMS or other computations. We used RMS in

our implementation and our results can be seen on Figures 1, 3, 5, 6.

Reducing Execution Time

String Art algorithm is by necessity compute-intensive. At each step, we are computing the error function of a set of a large number of strings – ideally all possible strings. For example, if we have a total of 1000 pins, each pin would be able generate 999 possible strings (for n pins, it amounts to $n(n-1)/2$ possibilities, hence 499500 possible strings for 1000 pins). One way to speed up the process is the reduce the size of this set of strings, perhaps by selecting, randomly, a subset of p possible strings. Obviously, we are not guaranteed a better result though a faster one. Another way to reduce the time taken is to save not just one best string per iteration, but a number more that satisfy certain criteria. For this, we assign a score to each string, establish a ranking, and select the top s ranked candidates – as many as we choose. s is small compared to p. In our examples, we used pools of $p = 1000$ strings and selected the best $s = 30$ of them (see Figures 1, 3, 5, 6).

Variations, Details and Parameters

To obtain more interesting results, we can vary the string thickness (or the picture resolution), the string brightness and color, or use an importance map to highlight some salient features of the original image such as the eyes or mouth of a subject.

String Thickness, Resolution and Anti-aliasing

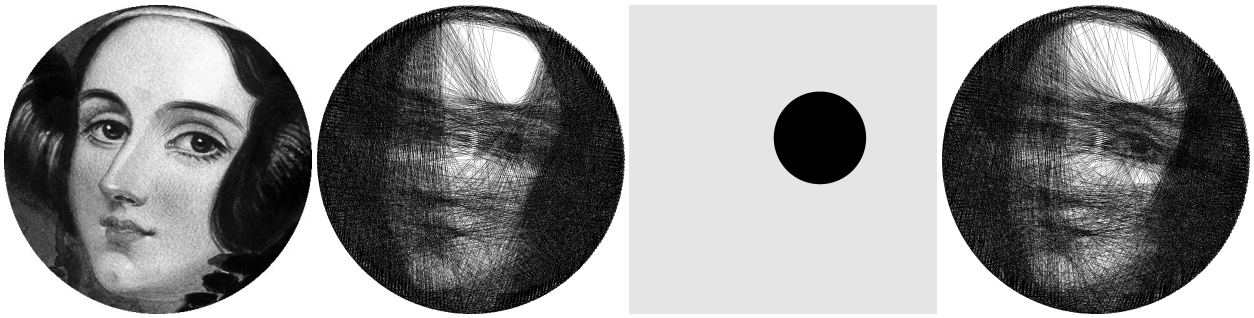
In most implementations, the string thickness is about one pixel, as they are rendered using some fast line drawing algorithms such as Bresenham's mid-point line algorithm [6] or Xiaolin Wu's anti-aliasing algorithm [14]. As thinner strings allow for more precision and thus outputs of higher quality, the algorithm of Birsak et al. [4] renders strings in a high-resolution pixel grids, and the low-resolution pixels that get to be compared with the pixels of the target image are of a grayscale color obtained by the mean of the corresponding set of high-resolution pixels. In our implementation, we also increase the resolution of the input image to get thinner strings and use Bresenham's line drawing algorithm. No anti-aliasing is used in the line drawing algorithm itself.

String and Background Brightness, Color and Transparency

In the algorithm presented in [4], strings are assumed to be perfectly opaque and black, the covering of one pixel by multiple strings will not be different from the covering of this pixel by only one string.

Some implementations that can be found online [3] allow for strings to be colored (in the full RGB color space, not just the grayscale color space), the color is chosen from a restricted color palette, and one string covering older strings of a different color overwrites them (all strings being opaque here too).

Our implementation allows strings to have colors with an arbitrary grayscale component and an arbitrary opacity component, the background of the area on which the strings are rendered can also be set to an arbitrary grayscale color. All these colors are fixed beforehand. For example, strings can be set to be black but only 30% opaque, that would grant the algorithm more freedom as in can play with overlapping strings to fine-tune colors step by step thanks to transparency. It can be considered as an easier variation of the String Art problem in which only opaque black strings are allowed. Indeed, allowing transparent strings makes the darkening of some parts of the image more precise, hence producing more detailed results. On Figures 1, 3 and 6, we can see that semi-transparent lines (rightmost images) allow for finer details such as the eyes and mouth of the character compared to any other technique using opaque strings. Note that is also reduces the final error between input and output images as the result is more precise. We can consider non-opaque black lines to be a useful new feature analogous to the resolution lowering of the algorithm of Birsak et al. [4] as it allows gray pixels (in low resolution).



(a) (b) (c) (d)

Figure 4: *(a) Original Ada image. (b) Our result without the importance map (*3500 *opaque black lines,* 256 *pins). (c) Proposed importance map to prioritize her left eye. (d) Our result using this importance map (*3500 *opaque black lines,* 256 *pins).*

Importance Map

Drawing a new line with the idea of improving some part of the image eventually causes collateral damage to the output, as the line has to pass through other parts of the image to get there. To make sure that the algorithm makes the right choice with regard to which area to sacrifice for the benefit of another one, it is possible to guide it with a map of importance [4]. It can be a grid of scalar values describing how important each pixel of the target image is, and each potential line is to be evaluated accordingly. In that case, each of its pixel score is multiplied by its corresponding importance value. An example is shown in Figure 4. The map can be carefully handcrafted for better results [4] (see Figure 3 (c)).

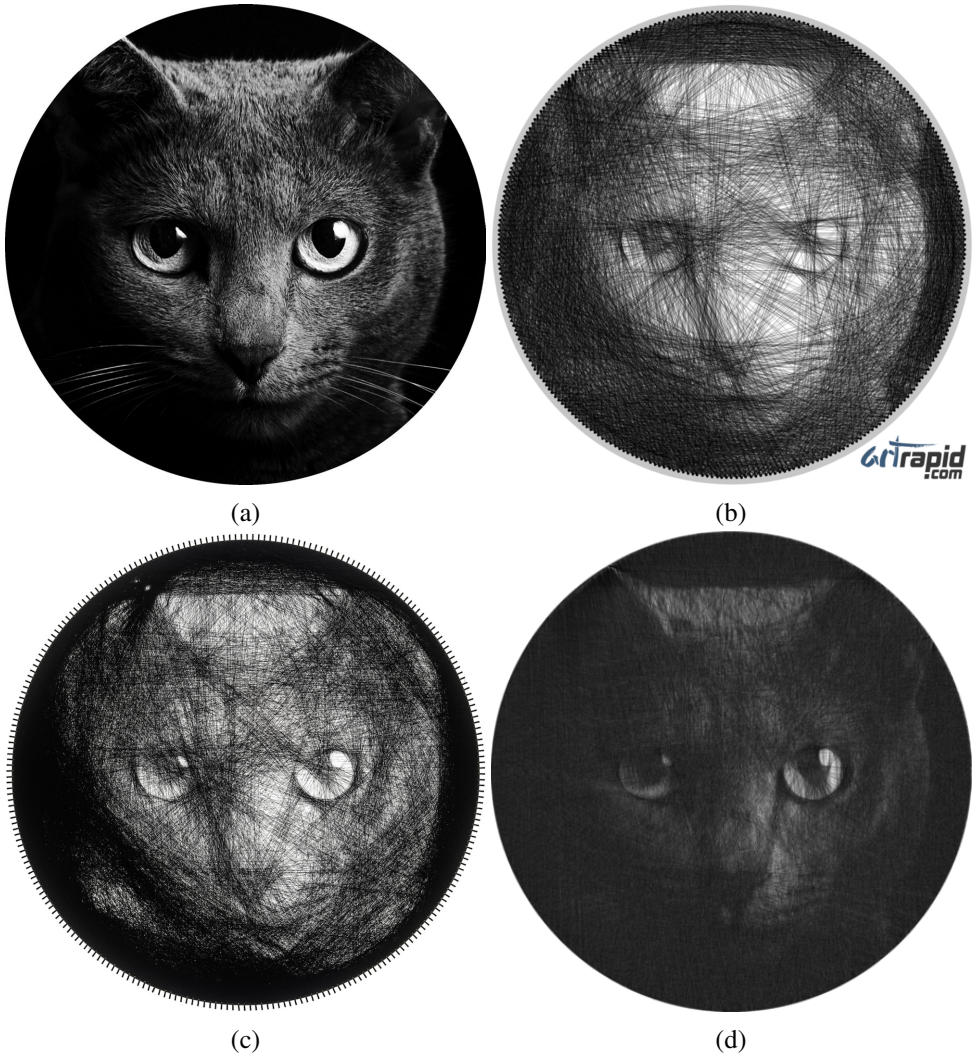
Pins

In the algorithms and implementations discussed in this paper, the set of pins in between which strings can be stretched is arranged uniformly in a circle, with only the number of pins making a difference. A larger number of pins allows for more precision in the rendering of some details of the target image. In the rendering of Figure 5 (d) we used 1000 pins and a string of opacity 30%. The algorithm ended after drawing 23300 lines, hence producing more details. The unrealistically high amount of lines in our result is a consequence of the high resolution and the darkness of the image. The production feasibility is directly related to the size of the support. If it is large enough, it can be produced. The high number of pins also reduces the appearance of Moire patterns. Variations include having pins inside the canvas as well [12].

Conclusion and Future Work

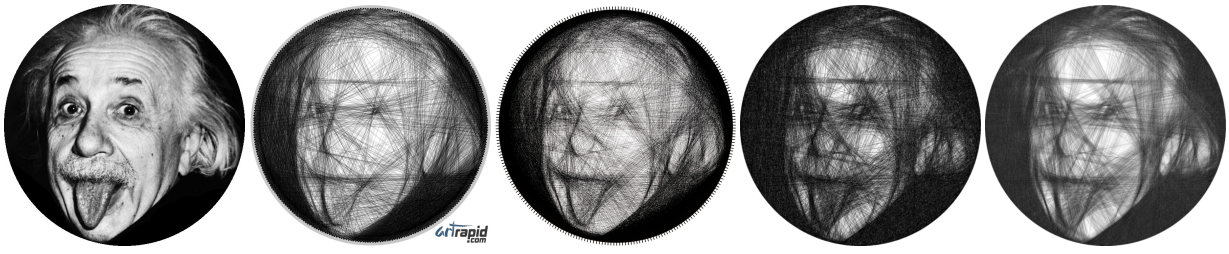
In this paper, we have proposed a greedy algorithm together with heuristics necessary to pick the best line and stop the algorithm while keeping the computation time reasonable (less than a minute). We have also proposed some variations of the original algorithm such as non-opaque strings and the use of an importance map to improve the results. There are several limitations that we would like to investigate in future work.

One problem with the automation of String Art is that the expressiveness of the output is hard to measure without a human. Our perception of some details may be unbalanced, and it may be that trying to replicate exactly the base image is not the best way to allow humans to perceive the base image when looking at the output. Insisting on relevant edges (such as those drawn by [11]) may be a way to enhance the human perception of the input through the output. Using machine learning to automate the making of human-relevant importance maps may be another relevant direction in this field. To evaluate such work, it is necessary to design and run perceptual studies to validate the results.



(a) (b) (c) (d)

Figure 5: *(a) Original cat image. (b) Art by ArtRapid [2](*3993 *strings, online simulation). (c) Art by Birsak et al. [4] (photography of the produced artwork). (d) Our result with* 1000 *pins,* 23300 *black strings of width* 1/4 *pixel and of opacity* 30%*, line sets of size* 5000*, picking* 30 *lines per set (*1713 *seconds, RMS* = 0.135*).*



(a) (b) (c) (d) (e)

Figure 6: *(a) Original Einstein image. (b) Art by ArtRapid [2](*3994 *strings, online simulation). (c) Art by Birsak et al. [4] (photography of the produced artwork). (d) Our result with* 4710 *opaque black strings (*225 *seconds, RMS* = 0.202*). (e) Our result with* 11370 *black strings of opacity* 30% *(*538 *seconds, RMS* = 0.140*). Both our results used* 1000 *pins, strings of width* 1/4 *pixel, line sets of size* 5000*,* 30 *lines per set.*

If possible, the evaluation of the choices can be done in parallel for even better performances (numerous and simple evaluations can be efficiently made parallel by dispatching them on the many cores of some available GPU).

When String Art algorithms perform reading and writing from the manipulated pixel grids, the successions of memory accesses are often of the form of diagonal lines across these pixel grids. If the pixel data is stored in memory in a row-major or column-major fashion, then such accesses will induce regular cache misses (as the access regularly changes of line or column). To reduce cache misses and thus improve performances, one possible method is to order the pixels in memory in a fashion that tries to preserve locality (small distances between pixels in the 2D grid induces small distances in the 1D memory mapping of those pixels) as much as possible, such as by ordering them as they appear in a Hilbert Curve or a Z-Ordering Curve that fills the pixel grid.[1]

Additional variations of the algorithm would be interesting to investigate:

- Support strings that are colored with a more diverse palette, as is done by [3] in their experimental results using strings of different colors;
- Allow the pins to be arranged as to accommodate the algorithm (in a fashion similar to what is done by [12]);
- Generalize the algorithm to be able to work with more shapes that just lines, such as not-so-stretched strings that are curved by gravity, or arbitrarily curved lines that cannot be crafted with strings, or even arbitrary surfaces. There are too many ways to generalize this problem for them to be listed here.

References

[1] A. Abakumova and A. Abakumova. 2019, https://news.artnet.com/art-world/ani-abakumova-thread-art-computer-1626352.

[2] ArtRapid. https://artrapid.com/.

[3] ArtRapid CMYK. https://artrapid.com/cmyk/.

[4] M. Birsak, F. Rist, P. Wonka and P. Musialski. "String Art: Towards Computational Fabrication of String Images." *Computer Graphics Forum*, vol. 37, no. 2, 2018, pp. 263–274.

[5] M. E. Boole. "The Preparation of the Child for Science." *Oxford : At The Clarendon press*, 1904.

[6] J. E. Bresenham. "Algorithm for Computer Control of a Digital Plotter." *IBM Systems Journal*, vol. 4, no. 1, 1965, pp. 25-–30.

[7] J. Delanoy, A. Bousseau and A. Hertzmann. "Video Motion Stylization by 2D Rigidification." *Proceedings of the 8th ACM/EG Expressive Symposium*, 2019.

[8] E. Demaine, M. Demaine and P. Vrellis. "String Art Font.", 2017, http://erikdemaine.org/fonts/stringart/.

[9] W. Jang, G. Ju, Y. Jung, J. Yang, X. Tong and S. Lee. "StyleCariGAN: Caricature Generation via StyleGAN Feature Map Modulation." *ACM Transactions on Graphics (SIGGRAPH 2021)*, vol. 40, no. 4, 2021.

[10] A. Kalaidjian, C. Kaplan and S. Mann. "Automated Landscape Painting in the Style of Bob Ross." *Proceedings of Computational Aesthetics in Graphics, Visualization, and Imaging*, 2009.

[11] H. Kang, S. Lee and C. Chui. "Coherent Line Drawing". *Proc. ACM Symposium on Non-photorealistic Animation and Rendering*, 2007, pp. 43–50.

[12] Laarco Studio. https://laarco.com/.

[13] P. Vrellis. "A New Way to Knit.", 2016, http://artof01.com/vrellis/works/knit.html.

[14] X. Wu. "An Efficient Antialiasing Technique." *ACM SIGGRAPH Computer Graphics*, vol. 25, no. 4, 1991, pp. 143—152.

[1] This is one type of optimization that is used by GPUs.