Knit report

Malo Malcom Drougard

Supervisor:Mina Aleksandra Konakovic
Laboratory: LGG, EPFL

June 9, 2017

Abstract

This project explores how we can draw an image using only a single thread. This thread is constrained to draw lines between a set of points. We will, first, study an algorithm published by Christian Siegel [1]. Then, we will improve this algorithm and extend it to colored images. At last, we will considerate the artistic opportunity of this process and develop a graphical user interface to interact with it at initialization and during the running time. You can find all the code used during this project on my github [5].

Contents

1 Introduction

This project was inspired by the work of Petros Vrellis [2]. This artist creates portraits using a single thread. At first, there is a ring with some pins nailed at the circumference. Then, he "knits" the thread from one pin to another. Finally, a face appears from the successive circle's chords. You can see an example of his artwork in figure 1. Petros Vrellis claims that he uses an algorithm to generate the path of the thread. The goal of this project is to explore deeply a knit algorithm.

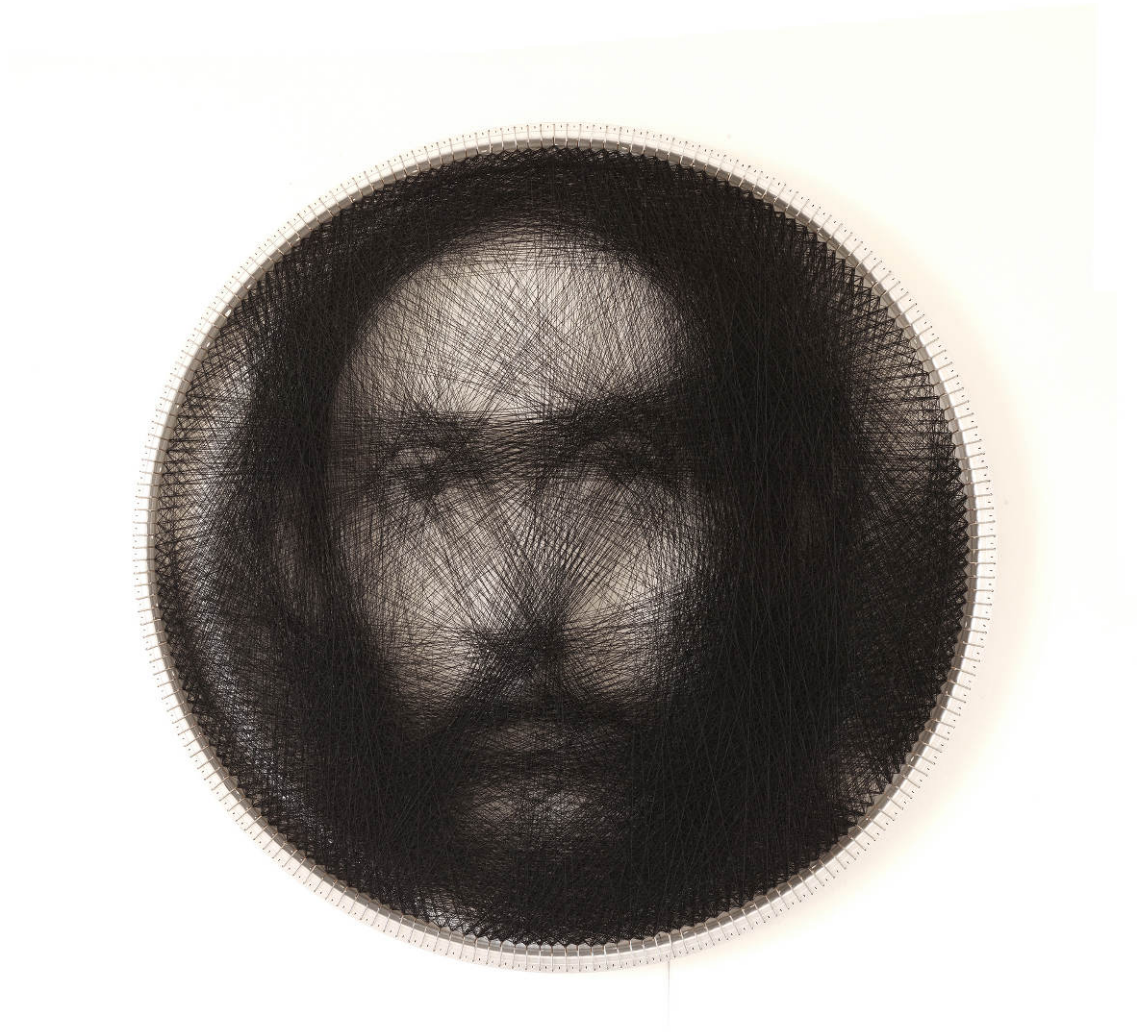


Figure 1: Artwork, Petros Vrellis, El Greco's Christ

2 Related work

Petros Verllis keeps the code that he uses to produce the thread path secret. But, Fortunately, Christian Siegel published a code [1] to produce a such path under the MIT License. So, we are free to reuse and to copy his code. The algorithm of Christian Siegel is the basement for our research and it is described in the section 3. Moreover, we didn't found any papers that talk about this topic.

3 Base algorithm

3.1 Overview

The algorithm of Christian Siegel takes as input a grayscale image and output a path to produce a "knit" artwork. It's an iterative heuristic algorithm. It starts at pin index 0. At each step, it evaluates every possible lines from the current pin to all other pins using a score function. The line that has the best score is chosen. The current pin is updated and the algorithm starts the loop again. The score function return simply the darkness value of the the line on a particular image called the *sketchImage*. Here is the trick. The *sketchImage* is at the beginning a copy of the input image and at each step the chosen line is subtracted from this image. This means that the *sketchImage* becomes whiter and whiter during the process. This particularity prevents the algorithm to chose every time the same line.

3.2 Base algorithm pseudo-code

```
1: procedure MAIN(originalImage, maxStep)
   \\ The following grid variable is an object that contains a 2D array of lines and pins
2:     grid = INITIALIZEGRID()
3:     sketchImage =COPY(originalImage)
4:     resultImage =CREATEWHITEIMAGE(originalImage.Size())
5:     step = 0
6:     currentPin = 0
7:     nextPin = −1
8:     path = {currentPin} \\ path variable is a list
9:     while step ≤ maxSteps do
10:        nextPin =FINNEXTBESTPIN(sketchImage, grid, currentPin )
11:        DECREASELIGHTNESS(resultImage, grid.lines[currentPin][nextPin])
12:        DECREASEDARKNESS(sketchImage, grid.lines[currentPin][nextPin])
13:        currentPin = nextPin
14:        path+ = currentPin
15:        step + +
16:    end while
17:    return resultImage, path
18: end procedure

19: function FINDNEXTBESTPIN(sketchImage, grid, currentPin)
20:    bestScore = −∞
21:    tmpScore = 0
22:    retPin = −1
23:    for all tmpPin ∈ grid.FindAllPossibleNextPinsFrom(currentPin) do
24:        tmpScore = LINESCORE(grid.lines[currentPin][tmpPin])
25:        if tmpScore > bestScore then
26:            retPin = tmpPin
27:            bestScore = tmpScore
28:        end if
29:    end for
30:    return retPin
31: end function

32: function LINESCORE(sketchImage, line)
33:    score = 0
34:    for all pixel ∈ line do
35:        color = sketchImage.GetColor(pixel)
36:        score = score + color.GetDarkness()
37:    end for
38:    score = score/line.Length()
```

```
    return score
end function

procedure DecreaseDarkness(image, line)
    decrementValue = line.GetColor().GetDarkness()
    for all pixel ∈ line do
        color = image.GetColor(pixel) \\ color are encoded as RGB
        color = color + decrementValue
        image.SetColor(pixel, color)
    end for
end procedure

procedure DecreaseLigthness(image, line)
    decrementValue = line.GetColor().GetDarkness()
    for all pixel ∈ line do
        color = image.GetColor(pixel)
        color = color − decrementValue
        image.SetColor(pixel, color)
    end for
end procedure


function InitializeGrid( )
    grid.pins = CreatePinsPositionedOnACircle()
    grid.lines = 2DArray
    for all pin1 ∈ grid.pins do
        for all pin2 ∈ gris.pins do
    \\ A line is simply a list of pixel indexes
            grid.lines[pin1][pin2] = GetLineUsingDDAAlgorithm(pin1, pin2)
        end for
    end for
    return grid
end function
```

3.3 Underlying model

All the infrastructure: the ring, the thread and the input image are model with digital images, more precisely 2D arrays of pixels. Each pin is located on a precise pixel. The thread from one pin to another pin is model as a straight line and transform into a list of pixels using the digital differential analyzer algorithm (DDA) [3]. The DDA algorithm basically take the biggest delta between the deltas on the x axis and the y axis, then it iterates over the chosen delta while updating at each step the other axis by an increment ($increment = notChosenDelta/chosenDelta$).

So far, we assume that pins are located on particular pixels and that the thread draw perfectly straight line. This seems quite reasonable, but, now comes the tricky part. The "DecreaseLightness" function, on line [49], uses the darkness of the line as the decrement value. This function updates the result image. This means that when two lines overlapping, because they cross each other or because they are mostly parallel at some point, the color of the lines are combine. You can see at figure 2 a magnify version of the model of a small ring with 39 pins after 9 steps with a thread that have a color of RGB(170,170,170).

The thread overlapping behavior of our model allows to have gray scale result images! The question is: "is it realistic?". We know that human eyes are not perfect and make some quite of summation when they can not distinguish details. We can imagine that this phenomena will happens if two threads become close. In fact, in the image of Petros Vrellis [1] we see some gray and not only black and white colors. Furthermore, a guy had produced a knitted artwork using only the code of Christian Siegler, see [4] and on his artwork, you can see different gray intensities. Moreover, another way to turn the problem is to find physical materials that fit our

model. Until now we have assume a perfectly opaque single-colored thread, one solution may be to use a translucent thread. Finally, I would said that some physical experiences would be needed to real answer this question. But, if on one hand, we chose the right parameters (number of pixel, decrement value) for our model and on the other hand, we tune the physical model to fit the model, we have good hope to have a usable model.

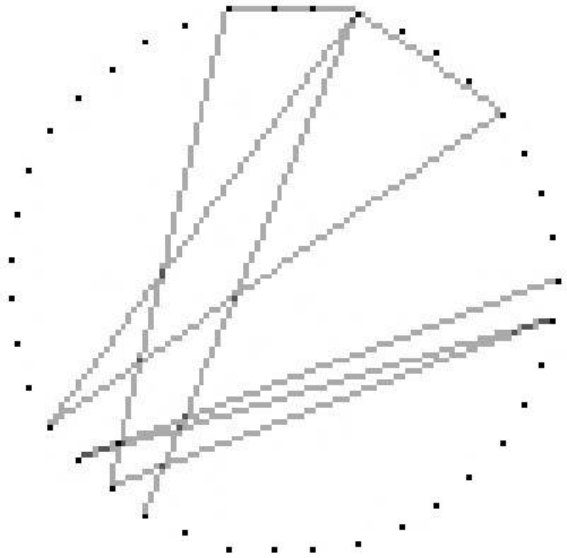


Figure 2: Magnified model of a small ring of 39 pins after 9 steps with a thread color of RGB(170,170,170)

4 Experiments & evaluations

4.1 Grid

4.1.1 Experiment

In our model, the thread can only go from one pin to another. Thus, there is only a limited number of lines that the thread can take. We call this set of all possible lines the grid. For example, if no line cross a pixel, this particular pixel as no chance to become dark. In this experiment, we want to understand better and improve this grid.

To understand the gird, we want a way to represent it. So, we choose to draw each lines of the grid with a small gray intensity (RGB(251,251,251)) on a white image. When two or more lines overlap the same pixel, the gray intensities are cumulate. This representation has the benefit of showing where the lines are and how many they are.

When we draw the representation of circular and symmetric pins arrangement, see figure 3a, we observe two issues: the apparition of a Moiré effect and the none uniform distribution of lines overlapping. Let's start by resolving the Moiré effect and then, focusing one the lines overlapping.

In fact, by the symmetry of the arrangement, the grid becomes a mandala and the lines cross each other's regularly. To reduce this Moiré effect, we introduce for each pin position some randomness. We implement a method that take any pins arrangement and move slightly each pins by zero, one, two or three pixel in each direction. We apply this method on the previous circular grid 3a to obtain figure 4a.

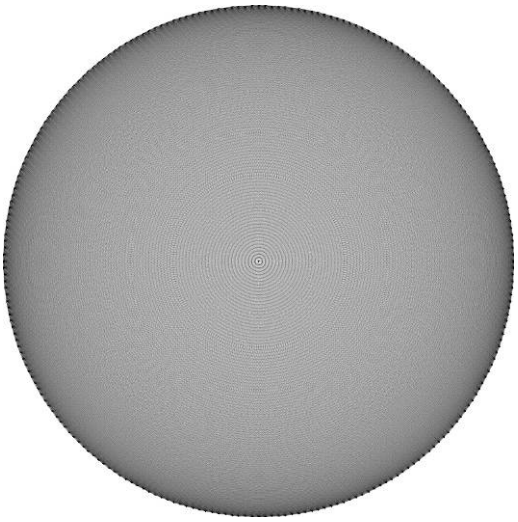
In the symmetric and randomized grid, we observe than the border are darker. If we look at all lines starting from a single pin, we see that on the pin all lines are overlapping and then, the distance between the lines are directly related to the distance from the starting pin. Remember that our model work on a pixel grid using DDA algorithm to draw lines. Thus, if the distance between to line is to small, both lines are on the same pixel and their intensities are cumulated.

Based on these two facts, we can claim that the region around a pin will always have a darker grid. Aware of this phenomena, we implement other pins arrangement to have more darker grid at center. The idea was that if we have a more darker grid at center, we have more lines in this area and, thus a more precise picture. So, we have implemented a method that allow us to add pins everywhere via a user interface and a tribal pin arrangement, see figure 5a . We also implement other grid shape: the ellipse grid shape, the square grid shape, more for curiosity than for optimization.

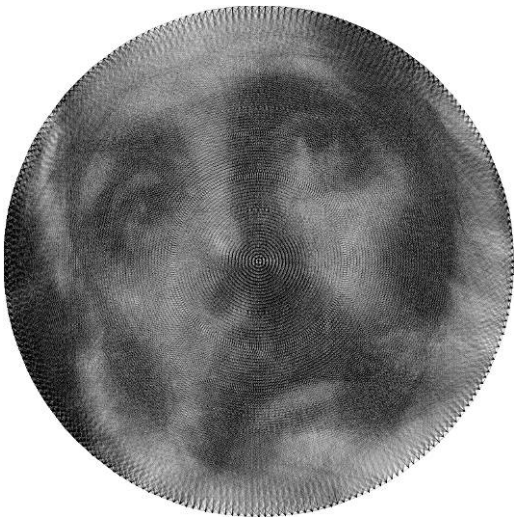
4.1.2 Evaluation

We see in figure 4b than the Moiré effect was indeed removed by our method. Furthermore, this method works on every pins arrangements.

The trial to have a more intense grid at center is not concluding. As you can see in figure 5b, the added pins are quite visible and ugly and do not improve the images in this area. In fact, this trial was based on a wrong idea. Having a more intense grid in a area not necessary induce a more precise picture in this area. The opposite seems more truthful. If we have a lot of lines converging to a region, this region becomes dark regardless of the original color of this area. If we look at the extreme case really close from pins, we observe that theses regions are quite dark and do not respect the original image. So, we do not found a better pins arrangement, but this experiment allow use to tries other grid shapes that we will later use for artistic reasons.

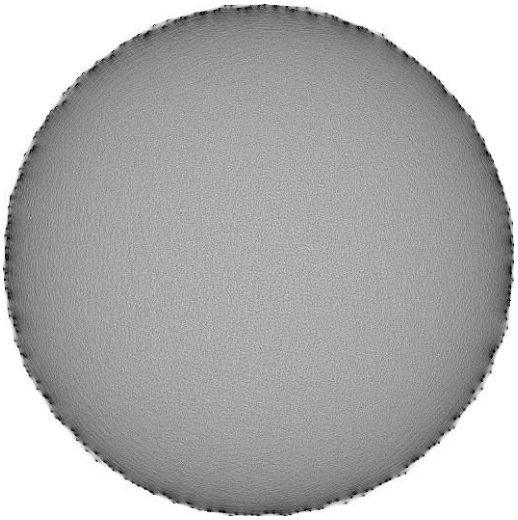


(a) symmetric circle grid of 240 pins

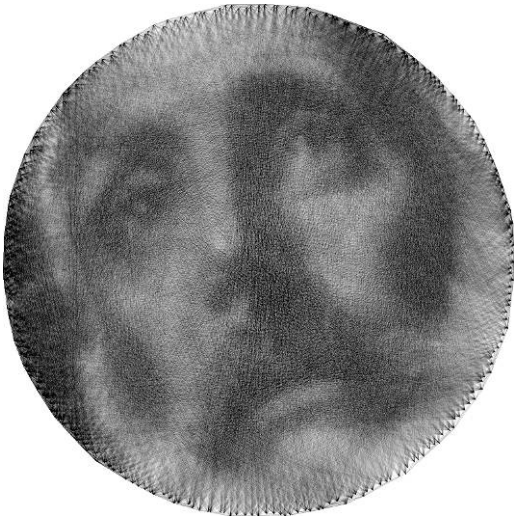


(b) El Greco's painting knitted in a symmetric circle grid of 240 pins

Figure 3: Standard circle grid

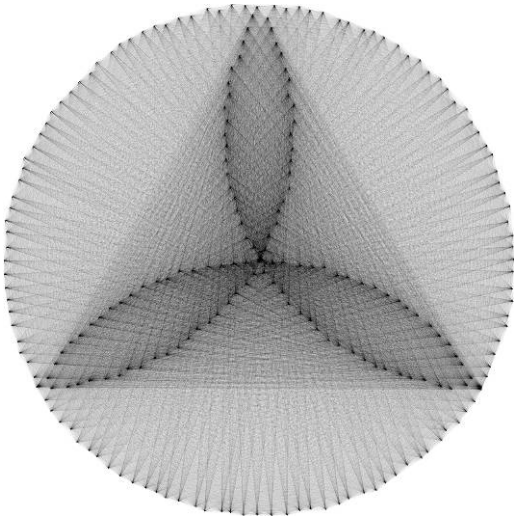


(a) randomized circle grid of 240 pins

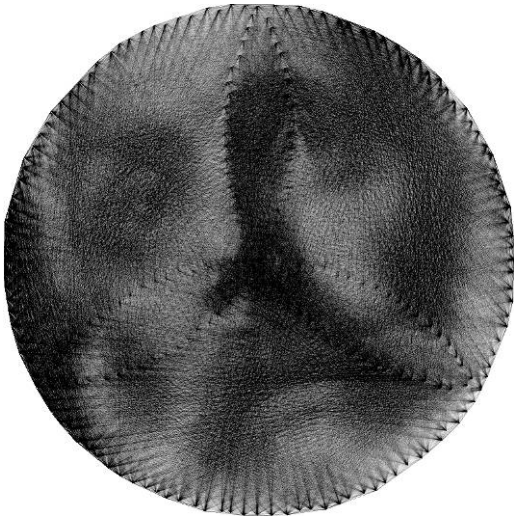


(b) El Greco's painting knitted in a randomized circle grid of 240 pins

Figure 4: Randomized circle grid



(a) randomized circle grid of 240 pins



(b) El Greco's painting knitted in a randomized circle grid of 240 pins

Figure 5: Grid with added pins

4.2 Line score function

4.2.1 Experiment

At each step of the algorithm described in section 3, the next pin is determine directly and only by the score function. So, the core of the heuristic is based on this score function. In this experiment, we want to try out other score functions and improve the existing ones. We will first, define some objects to have a more clear and concise pseudo-code. Then we will describe the explored score functions using pseudo-code. Finally, in the next subsection we will present the results.

Def resultImage: The image that hold the actual result of the algorithm. At begin this image is white, then at each step the new gray line is drawn.

Def diffImage: The virtual image produce by subtracting the actual $resultImage$ from the $originalImage$. For example: if a pixel has a darkness value of 30 in the $originalImage$ and has a darkness value of 45 in the $resulImage$, then it has a darkness value of -15 in the $diffImage$. This image is said virtual because, we do not know how to display a color with a negative darkness value. (We assume that white has a darkness value of 0). However, in our context, a negative darkness value means that we should add some white to this pixel to reach the exact darkness of the original pixel.

Def sketchImage: The image is produce by subtracting the actual $resultImage$ from the $originalImage$ but with a minimal darkness value of 0. For example: if a pixel has a darkness value of 30 in the $originalImage$ and has a darkness value of 45 in the $resulImage$, then it has a darkness value of 0 in the $diffImage$.

Let's describe the score functions. The first one, "LineScore", was discovered in the code of Siegel and it is also describe in section3. The other functions was completely develop by us. The names are the same as in the code on github[5], but the implementation differs slightly. In particular, the $diffImage$ doesn't exist as an object, but is computed inside the function.

function LineScore($sketchImage, line$)
  $score = 0$
  for all $pixel \in line$ **do**
    $color = sketchImage.GetColor(pixel)$
    $score = score + color.GetDarkness()$
  end for
  $score = score/line.Length()$
  return $score$
end function

Idea behind this score function: If we choose the darker line in the sketch image. This line will be added in result image and the difference between the original image and the result image will be smaller.

function LineScoreSignedDifferenceBetweenOriginalAndResult($diffImage, line$)
  $score = 0$
  for all $pixel \in line$ **do**
    $color = diffImage.GetColor(pixel)$ \\ we change the input image!
    $score = score + color.GetDarkness()$ \\ the darkness is virtual and can be negative
  end for
  $score = score/line.Length()$
  return $score$
end function

Idea behind this score function: The idea was to penalize, also, the pixel that get darker than the original image.

function LineScoreWeightedExtremity($diffImage, line$)
  $score = 0$
  $maxFactor = 3$
  for all $pixel \in line$ **do**
    $color = diffImage.GetColor(pixel)$
    $darkness = score + color.GetDarkness()$ \\ the darkness is virtual and can be negative
    \\ darkness.Limit() is a function return the higher possible value for darkness
    $factor = 1/darkness.Limit() * absoluteValue(darkness)$
    \\ create an exponential function from 1 to maxFactor
    $factor = Power(maxFactor, factor)$
    $score = score + (factor * darkness)$
  end for

$score = score/line.Length()$
return *score*
end function

Idea behind this score function: The idea was to penalize or advantage more values that are far of the original value.

function LINESCOREDELTA($sketchImage, line$)
\\ In this function we assume that the pixels in the line are ordered
$score = 0$
\\ We do not take the last pixel of the list because, it has not a next pixel
for all $pixel \in line\{lastPixel\}$ **do**
$nextPixel = line.GetNext(pixel)$ \\ return the next pixel on the line without changing pixel
$darkness1 = (diffImage.GetColor(pixel)).GetDarkness()$
$darkness2 = (diffImage.GetColor(nextPixel)).GetDarkness()$
$diff = AbsoluteValue(darkness1 - darkness2)$
$score = score + diff$
end for
$score = score/line.Length()$
return *score*
end function

Idea behind this score function and remarks: The human eyes are more aware of contrast than lightness. The idea was to use this fact and search for the line with the less contrast. But use alone, always the same single-colored lines are chosen.

4.2.2 Evaluation

Metrics

We have used two metrics to evaluate the different score functions. The first one is the absolute error of the image. The second one is human preference. Absolute error metric (AE) is simply the sum of the absolute difference of darkness between the resultImage and the originalImage. This can be interpreted as the distance of the result image from the original image. We may, also, use a mean square error or an error function based on neighbor contrast. But, we decide to save this time to explore other direction. To evaluate human preference, we ask to six persons to choose between different knitted versions of the same input image. Each of the knitted version was produce using the same algorithm and the same parameters, only the score function changed. We like the idea to have two really different metrics. On the one hand, absolute value error gives us a really easy and fair way to evaluate the result. On the other hand, our images are design for human being, so human preferences is a metric that reflects the final audience tendencies.

Data used for evaluation

For score functions, "LineScore" (L), "LineScoreSignedDifferenceBetweenOriginalAndResult" (D), "LineScoreWeightedExtremity" (W), we run our algorithm up to 30002 lines on 7 images. At each 1000 strings, we save the result image and compute the absolute error. The 7 input images chosen for this evaluation are in the appendix A.1. We have tried to cover a large range of images, portrait, painting, abstract art, landscape. We use a square randomize grid to not have the effluence of the border and to not have the Moiré effect. Remark that we save the randomize square pins arrangement to have for each image exactly the same grid. We do not evaluate "lineScoreDelta" because it needs to be combined to produce viewable output.

This process produces 630 result images and 630 absolute error computations. To be able to interpret the result, we produce one chart for each image. On the x axis of the chart, we have the number of lines and one the y axis the absolute error value for each score function. These charts

are in appendix A.3. For the human preference test, we chose the three versions of each image at the lowest point of the chart. They are in appendix A.2. At some point, it will be relevant to compare result images produce with the same score function and the same input image that differs around one percent in their AE metrics. Remark that these images differs because one have less lines than the other. You can find such images in appendix A.4. Finally, we want to compare the difference between the images used for the human preference test so we compute their differentiation in appendix A.5

Analysis

Comparison between LineScoreSignedDifferenceBetweenOriginalAndResult (D) and weightedExtremity (W)

First, we like to highlight the fact that D and W function are really close in the resulting images and in the underline scoring procedure. In fact, W is a try to tune the D function. We, just, give more weight to the extreme values. Human preference for W or D is not clear. Both have 19 likes, see table 1. The curve of the charts have the same shape, see appendix A.3. Differentiation images are quite homogeneous (sees images in section "Difference between scoring function at the minimal absolute value point"). So, we can state that W and D belong to the same category of score function and the comparison between these two score functions are not so relevant. Thus, for the rest of the analysis, we will focus on the comparison between L and D function.

Comparison between LineScore (L) and DiffOriRsult (D)

If we look at the global shape of the chart, we see that process using D function are stabilize earlier than process using L function. This come from the fact, that all lines in D become bad earlier and the process starts to loop in a deterministic fashion. At opposite, L tries still to ameliorate the picture until the sketchImage is totally white.

Let's now focus on images at the lowest point of the charts. The human preference test is quite clear. The W and D functions get 38 votes together and the function L gets only 4 votes, see table 1. So, we have a strong human tendency for W and D versions over L. However, the absolute error difference between functions D and L, at the minimal point, is less than 1 percent in every cases. Can this 1 percent alone explain the human tendency? If we take a look at an absolute error difference around 1 percent between two images using the same scoring function, see appendix A.4, we, just, observe a slightly lightness difference. The difference of 1 percent between the different versions used for the test seems more deeper. In fact, if we make the differentiation of the D images and the L image at the minimal point, see appendix A.5, we observe that the differences are concentrate around white areas and the borders of the shapes. So, it seems that our new score function produce more contrasted and precise images. This makes sense, because D function cares about white areas, whereas L function are more only focused on filling black areas. This clarity and contrast property is a reasonable way to explain the human preference for the D version. Finally, aware of this strong human tendency, this contrast property and clarity property, we can claim that our new "LineScoreSignedDifferenceBetweenOriginalAndResult" function improve the existing algorithm.

IMAGE	abstract	city	dance	elgreco	starik	tree3	tree4
Person1	W	D	D	W	D	D	L
Person2	D	D	W	D	W	L	D
Person3	W	D	W	W	W	D	W
Person4	D	W	D	W	L	L	D
Person5	W	W	W	D	W	D	W
Person6	W	W	D	D	W	D	D

Table 1: Human preference

4.3 Color

4.3.1 Experiment

Until now, we always work with grayscale images. The goal in this experiment is to generalize our algorithm to produce colored images. The idea is to have three threads of primary color and combine them to obtain the desired color.

Additive color model extension

RGB color model is an additive color model. It's mean that the ground color is black and then we add the red (R), green (G) and blue (B) components as light colors. Until now, we manage the RGB encoding using lightness and darkness concept. So, we were a little perturbed by this encoding. In fact, if we start with a white image, this means that all values are set to RGB(255,255,255) and if we subtract a red color (RGB(255,0,0)), we obtain a cyan color (RGB(0,255,255)). In our model, it's seems quite unrealistic to draw with a red thread and obtain a cyan color. We decide in a first time to have a black background image and to add the color of our thread. So, if we draw with a red thread (RGB(255,0,0)) on a black background (RGB(0,0,0)), we obtain a red line (RGB(255,0,0)). We modify our algorithm to have a red thread, a green thread and a blue thread. Each thread tries to optimize is color component independently. The main while loop in base algorithm, see section 3.2 line 1 will be transform into the while describe in algorithm 1. The score function would be the same as the "LineScoreSignedDifferenceBetweenOriginalAndResult" function of section 4.2 except that we would use the color component of RGB instead of the "GetDarkness" function.

Algorithm 1 Main loop for RGB extension

while $step \leq maxSteps$ **do**
  $nextRedPin$ =FinNextBestRedPin($sketchImage$, $grid$, $currentRedPin$)
  $nextGreenPin$ =FinNextBestGreenPin($sketchImage$, $grid$, $currentGreenPin$)
  $nextBluePin$ =FinNextBestBluePin($sketchImage$, $grid$, $currentBluePin$)
  IncreaseRed($resultImage$, $grid.lines[currentRedPin][nextRedPin]$)
  IncreaseGreen($sketchImage$, $grid.lines[currentGreenPin][nextGreenPin]$)
  IncreaseBlue($sketchImage$, $grid.lines[currentBluePin][nextBluePin]$)
  $currentRedPin = nextRedPin$
  $currentGreenPin = nextGreePin$
  $currentBluePin = nextBluePin$
  $step++$
end while

Subtractive color model extension

We remembered us the existence of other color models, in particular the cyan (C), magenta (M), yellow (Y) color model used for printers. In this model, we start with all the visible light spectrum and we add filters to produce the wanted color. For example, if you want to produce a red color you should filter every wavelengths except the red ones (around 650nm). This can be done by filtering the magenta wavelengths (around 510 nm) and the yellow wavelengths (around 580 nm). In term of encoding this mean that RGB(255,0,0) will be transform into CMY(0,1,1). Remark that in the CMY encoding, it is standard to have a maximal value 1. There is quite simple bijection between this two models describe bellow.

$$C = 1 - R/255, \quad M = 1 - G/255, \quad Y = 1 - B/255 \tag{1}$$

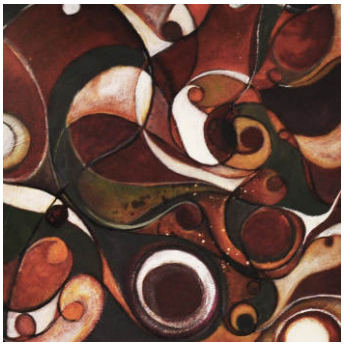
So, we came up with the idea to switch into to CMY color model and to have cyan, magenta and yellow threads instead red, green and blue threads. The rest of the algorithm is the same as for the additive color extension.

4.3.2 Evaluation

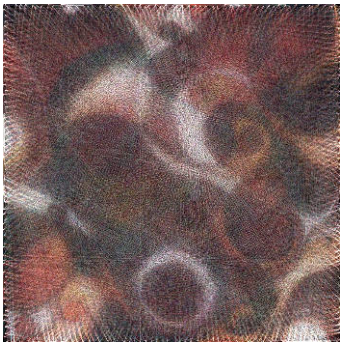
First of all, the images produced by our both new algorithms, see figure 6, seems to be well colored and representative of the original images. So, we can claim that our generalized algorithm works for our generalized underlying model. This means two things: one, our underlying model can express a reasonable representation of the image; two, our heuristic can find a god representation of it.

But, let's look if our generalized underlying model is still reasonable. In fact, the only change is the color model. Before we used a gray color model, now we use the RGB color model or the CMY color model. The RGB color model, as describe before, assumes that the color acts as light. In fact, if we have a wool red thread and a wool blue thread, we will never get a yellow color as expected by the RGB model. Even, if the threads are translucent, we would have some kind of dark brown. This is because if we use translucent threads we are filtering the light and thus be more close of a subtractive color model. We would said that if we want to use the RGB color model, we need some kind of amazing setup with optical fiber illuminated by leds. The CMY model seems more realistic. In fact, if we have translucent threads we can expect that the color would be indeed combine if they overlap. If we set the thread color at CMY(2,0,0), CMY(0,2,0), CMY(0,0,2), we obtain the image 6g, if we set the thread color at CMY(26,0,0), CMY(0,26,0), CMY(0,0,26), we obtain the image 6h and finally if we set the thread color at CMY(110,0,0), CMY(0,110,0),CMY(0,0,110) we obtain the image 6i. I like the image 6h and it uses colors of 26 intensity, so we can imagine that it would be doable with threads that has a width of one pixel and filter the light at 10%. But, this is highly hypothetical and some physical experiences seem needed to go further.

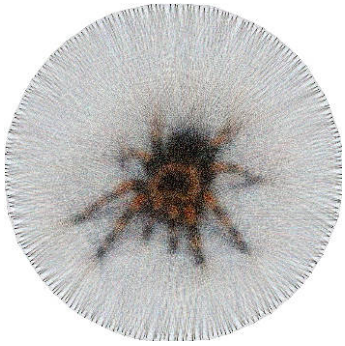
(a) Result image produced using subtractive color algorithm on 6b image, (3300 lines)



(b) Input image "Abstract2"

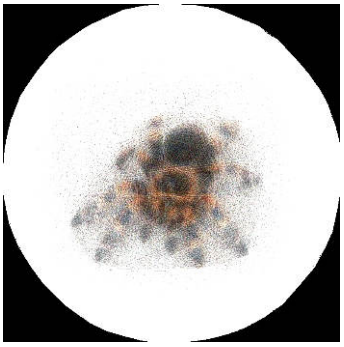


(c) Result image produced using additive color algorithm on 6b image, (2700 lines)



(d) Result image produced using subtractive color algorithm on 6e, (8001 lines)

(e) Input image "Tarencula2"



(f) Result image produced using additive color algorithm on 6e, (51000 lines)

(g) Result image produced using subtractive color algorithm and threads of intensity 2/255, (111000 lines)

(h) Result image produced using subtractive color algorithm and threads of intensity 26/255, (10000lines)

(i) Result image produced using subtractive color algorithm or the threads of intensity 110/255 (3000 lines)

Figure 6: Images produced using the subtractive or additive color extension

4.4 Interaction

4.4.1 Experiment

The idea behind this experiment, is to view the algorithm as an art tool and do not try to strictly replicate the input image. When a photographer is in the front of a scene, he can setup different parameters as the focal length, the aperture, the shutter speed and so on. This parameters allow the photographer to introduce some effect on the captured image. In this experiment, we had developed a graphical user interface to interact with the algorithm. We would like that an artist can play with our software, a little bit as with a camera. In our GUI, you first decide if you want to use subtractive colored threads, additive colored threads or a single gray thread. If you chose the single thread you came up on a interface where you can set the grid, the line score function,

the thread density and one extra feature call "brushing mode", see the screenshot 28 in appendix. If you chose the subtractive colored threads or the additive colored threads you came up on a interface where you can set the grid and the density of threads, see the screenshot 29 in appendix. The color thread interface is less developed due to time constraint and not for feasibility reasons.

The grid parameter is explain in section 4.1. The line density refers to color intensity of the thread, more precisely at the decrement value of the "decreaseLightness" function (see: start algorithm pseudo-code, line 49). The score line function is explain in section 4.2. What is interesting to notice is that this parameters do not need to stay constant during the process. So, we allow these parameters to change during the process!

Let's explain the new "brushing mode" feature. Basically, this feature enable to have different levels of importance in the image. The idea is to have for each pixel a factor that weight the importance of the pixel. To do so, we add to our implementation a 2D array of the image size. This 2D array acts as mask and multiplies the darkness value of the "diffImage" pixel in the line score function, see the new line score function described in algorithm 2. The mask can change during the process and can be setup through our user interface. The user simply needs to select the "brushing mode" and chose the weight of the brush and he can draw on the image using the mouse, see screenshot 30 in appendix.

Algorithm 2 Line score function that enable the "brushing" feature

```
1: function LineScoreWeightedByMaskFactor(diffImage, line)
2:     score = 0
3:     for all pixel ∈ line do
4:         color = diffImage.GetColor(pixel)
5:         tempScore = color.GetDarkness() * mask.GetFactor(pixel)
6:         score = score + tempScore
7:     end for
8:     score = score/line.Length()
9:     return score
10: end function
```

4.4.2 Evaluation

If we run the software using the setup shown at screenshot 30. This means having a factor 5 in the eyes areas and every where else a factor 1. We get the image 7b. If we compare it with the image 7a produced using the same parameters but without a brushed area, we see that the eyes image are more precise in the brushed image and the rest of the face is more precise in the none brushed image. So, the "brushing mode" feature allows the user to have a kind of focus! Remark, that this focus work because, we use the "diffImage" in the scoring function. If we had use the "sketchImage" as in the original "LineScore" function, the area would simply becomes completely dark.

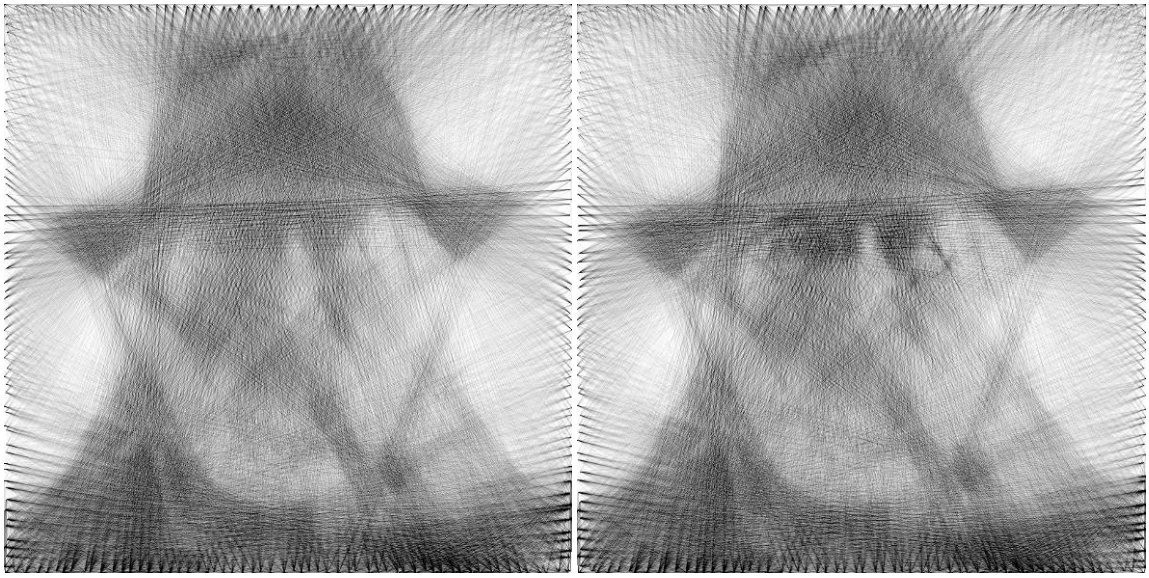
The line score score function parameters was already explore in section 4.2. Moreover, we add a score function that return a random number. These allow the user to have some random steps.

The intensity of the color of the thread has as effect on the visibility of the lines and on the range of color.

The grid parameter allow to set pins everywhere. We had noticed the fact that the pins region becomes darker and the lines converge to it, in section 4.1. This is a unwanted property when we want to reproduce faithfully the image. But, in this context we can hijack this property to get some interesting art effect, see image 8.

Of course, the interface is more a proof of concept that a finish product. In fact, the effect of the parameters are hard to predict for an unexperienced user. Furthermore, the process is relatively time expensive (around 2 hours for a image of 20000 lines on a home laptop). So, it can be quite fastidious to find a nice knitted image. It would be very useful to have an unpick feature that allows to roll back the path.

Finally, we would said that an artist can have a relatively large range of expression combining this feature and parameters before and during the process. It is certainly possible to add parameters and features to have an even more artistic opportunity.



(a) Standard Starik (12000 lines) (b) Starik with brushed eyes (12000 lines)

Figure 7: Starik input images produced with the same parameters except for the brushing area

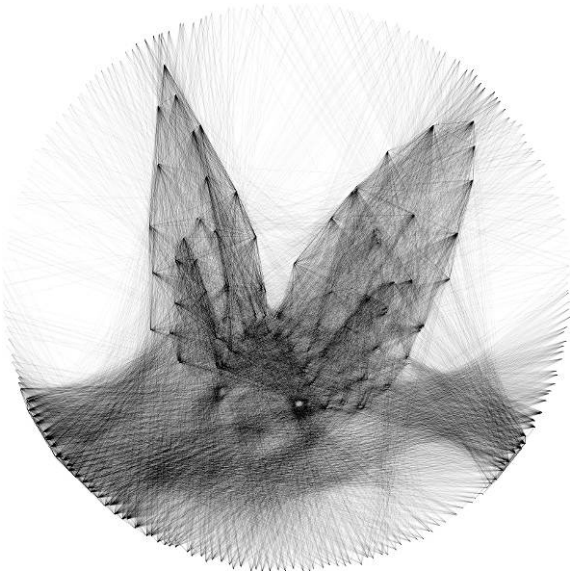


Figure 8: Manually added pins grid on a bat image (13000 lines)

Figure 9: Knitted bat image using brushed mode and extra pins (20000 lines)

5 Conclusion

We have seen many time during the project that the algorithm assumes a rough underlying model. So we suppose that if we physical knit the path produced by our software, we do not obtain exactly the result image. However, if we make abstraction of the model and we look at the algorithm it self. We can say that we have successfully ameliorated the algorithm with new line score functions. We have, also, successfully generalized the algorithm for color. Finlay, we have extended his art potential through a generalization of the grid concept, a new "focus" feature and a more interactive interface.

6 Further work

As said in the conclusion, one missing crucial point is the relation between the physical knitted artwork and our model. So, maybe, the first step to continue this project is to make some experiences to validate or to disprove the model and find its limitations. We have started to work at a new model based on percentage and probability, but unfortunately we have not finished yet. Another direction is to stick to the model, but to search another solution to solve this optimization problem. It will be, also, interesting to try to extend this single thread drawing concept to 3D. Of course, this will become drastically more complex due to path constraints.

7 Acknowledgments

I would like to sincerely thank Mina Aleksandra Konakovic that has supervised this project and that shown me the importance of writing the report during the project.

I would like to sincerely thank Christos Kotsalos that has proposed to explore this unexplored topic.

I, also, would to express my gratefulness to the Computer Graphics and Geometry Laboratory of the EPFL, especially to his Professor, Doctor Mark Pauly, that allowed this project.

Finally, I would thank Christian Siegel that shares his knowledge with the world.

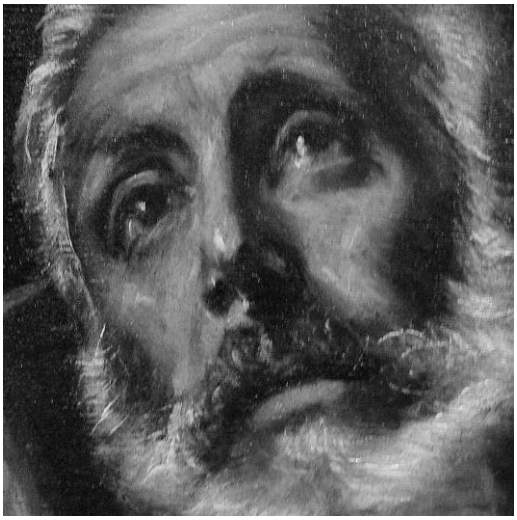
A Lines score figures

A.1 Input images

(a) image input: abstract1

(b) image input: city1

(a) image input: dance1



(b) image input: elgreco1

(a) image input: starik1

(b) image input: tree3

(a) image input: tree4

A.2 Human preference test images

Left Images: compute with "LineScore" function
Middle Images: compute with "LineScoreSignedDifferenceBetweenOriginalAndResult" function
Right Images: compute with "WeightedExtremity" function

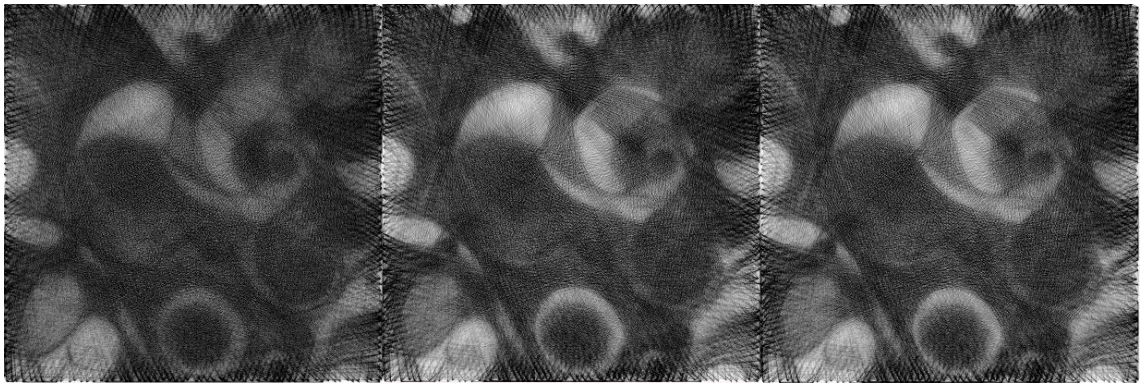


Figure 14: Human preference test image for "abstract1" input

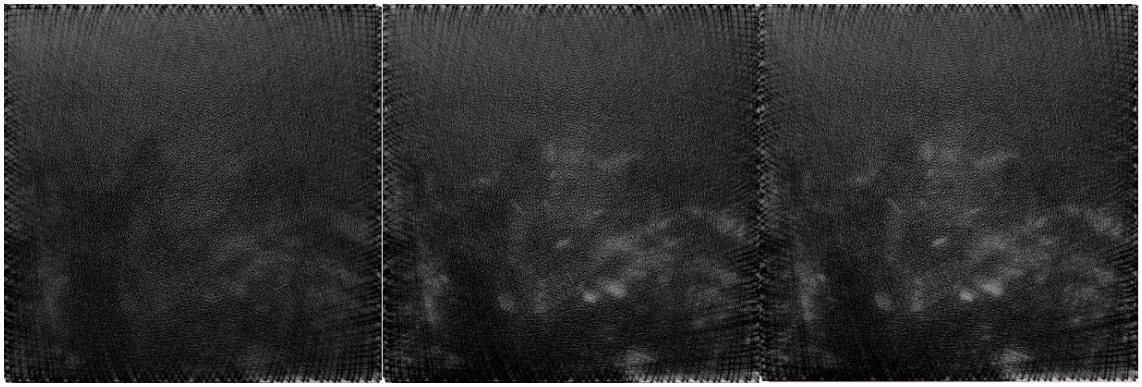


Figure 15: Human preference test image for "city1" input

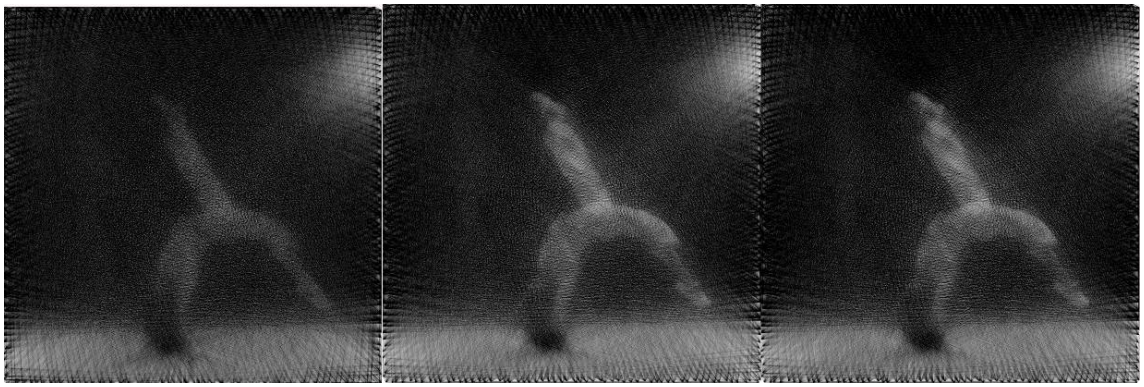


Figure 16: Human preference test image for "dance1" input

Figure 17: Human preference test image for "elgreco1" input

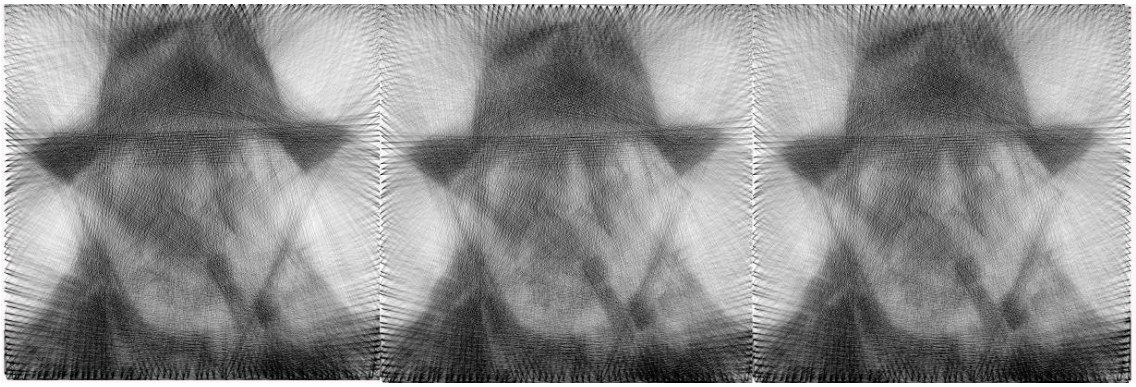


Figure 18: Human preference test image for "starik" input

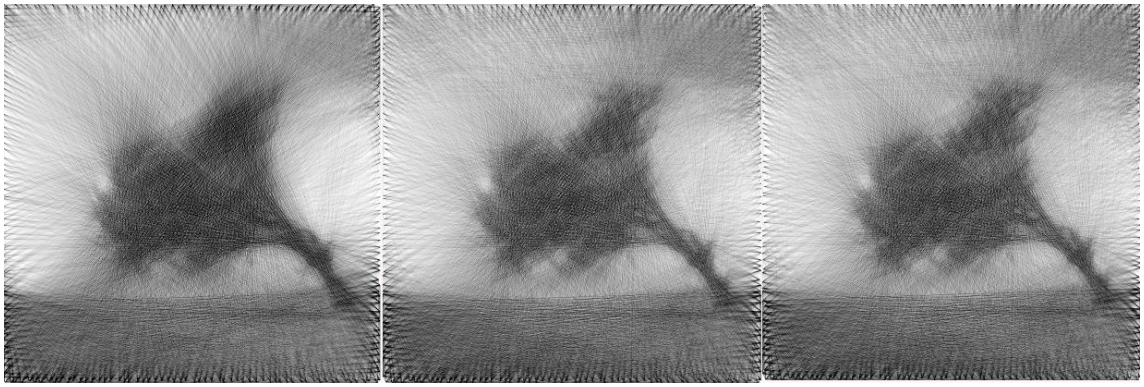


Figure 19: Human preference test image for "tree3" input

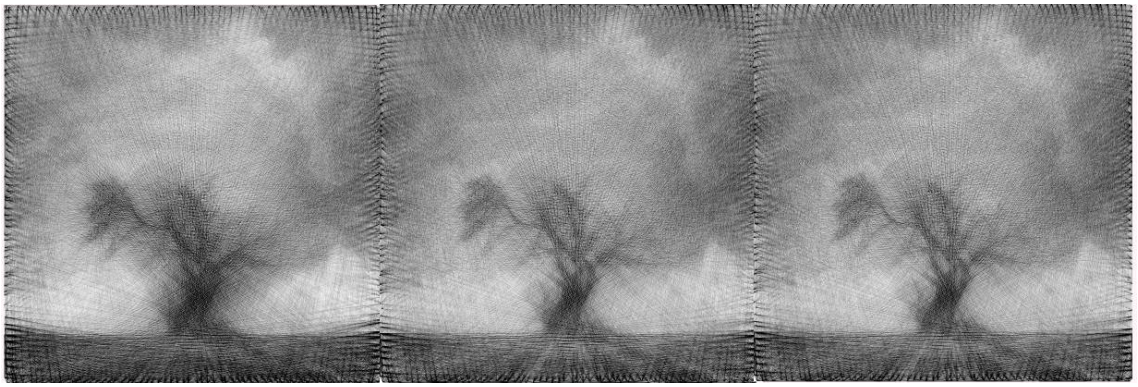
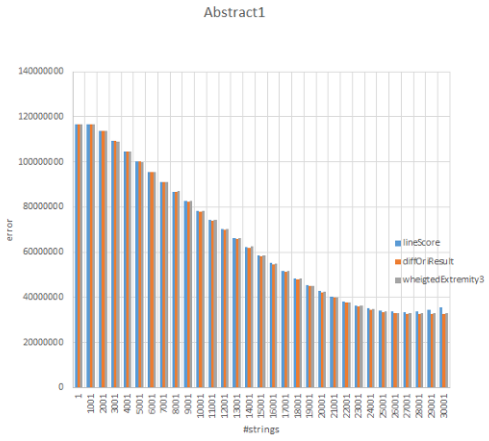
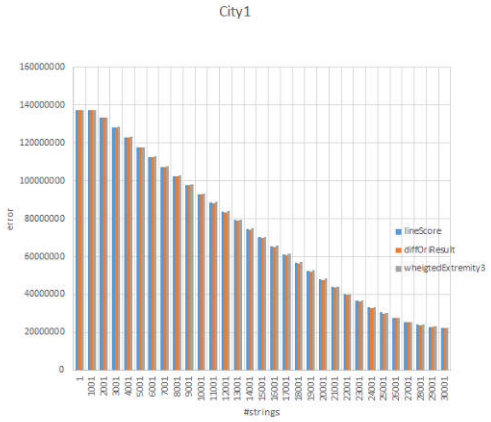


Figure 20: Human preference test image for "tree4" input

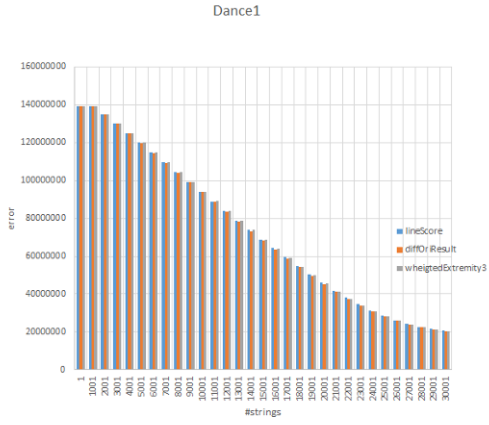
A.3 Absolute error charts



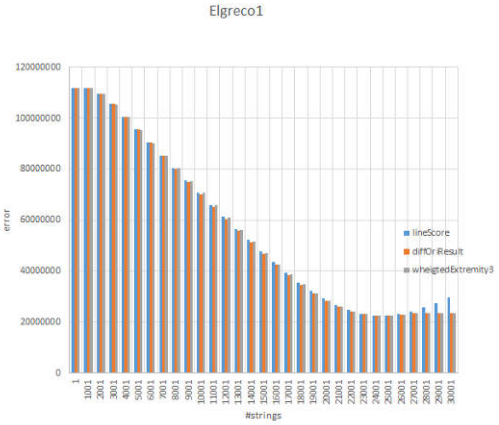
(a) Absolute error chart: abstract1



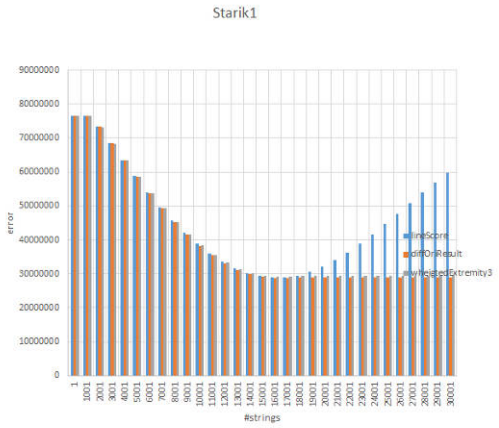
(b) Absolute error chart: city1



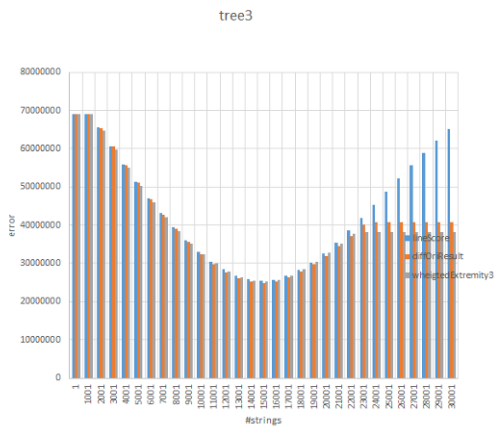
(a) Absolute error chart: dance1



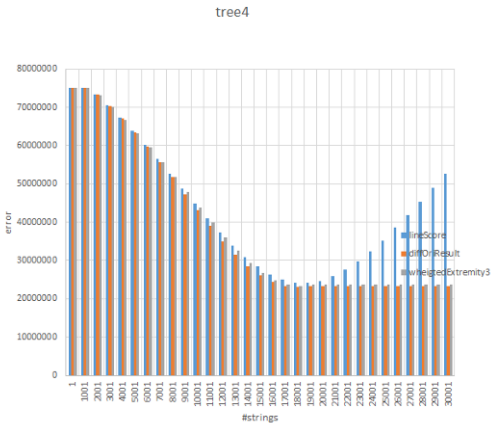
(b) Absolute error chart: elgreco1



(a) Absolute error chart: starik1

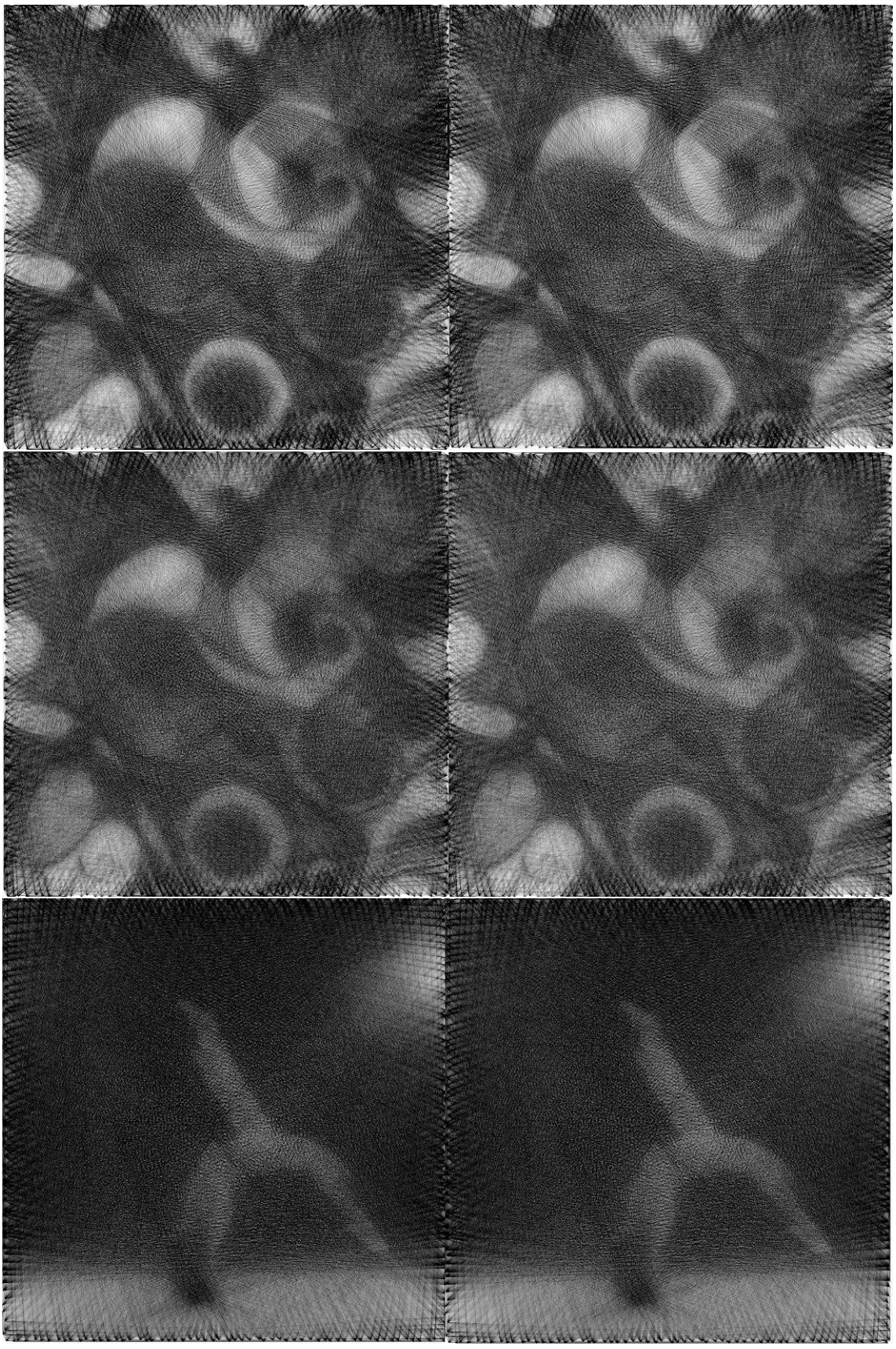


(b) Absolute error chart: tree3



(a) Absolute error chart: tree4

A.4 Tuple of result image that has AE metric difference of one percent

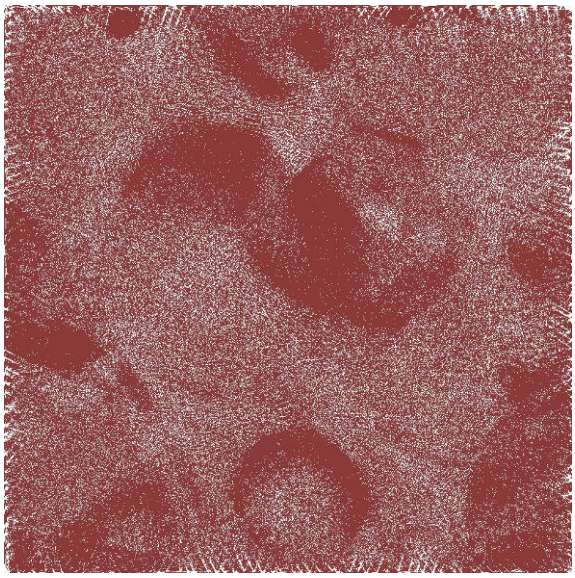


A.5 Differentiation between D and L image used in the human preference test

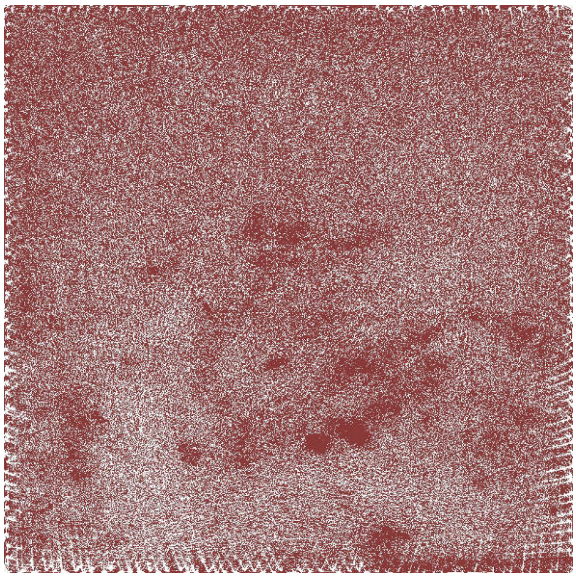
The following images are the absolute error by pixels between L images and D images used in the human preference test. They was compute using compare of "imagemagik". More precisely, they was compute using the following command line:

»> compare "$file1" "$file2" -metric mae -fuzz 1% -compose src -quality 100 -highlight-color IndianRed4 -lowlight-color White $outname

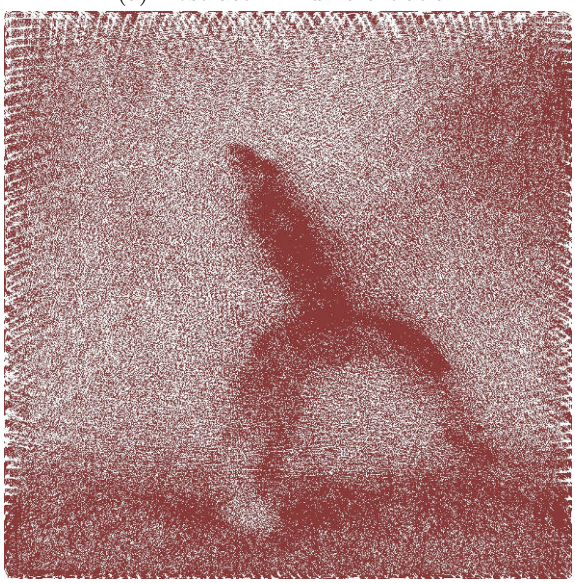
where $file1 = linescore image , $file2 = diffOriResult image



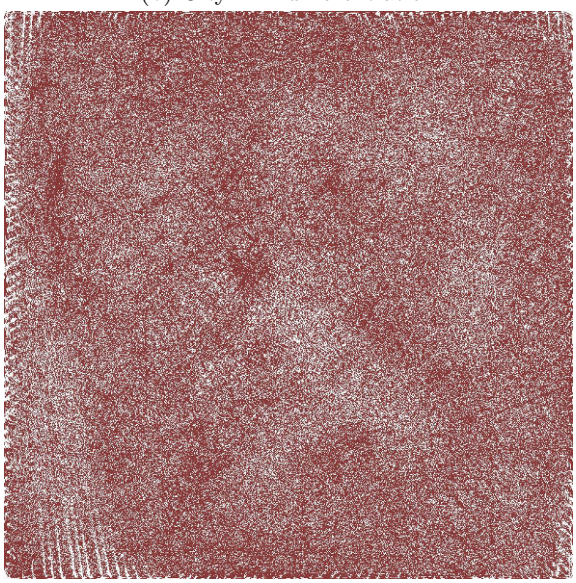
(a) Abstract D L differentiation



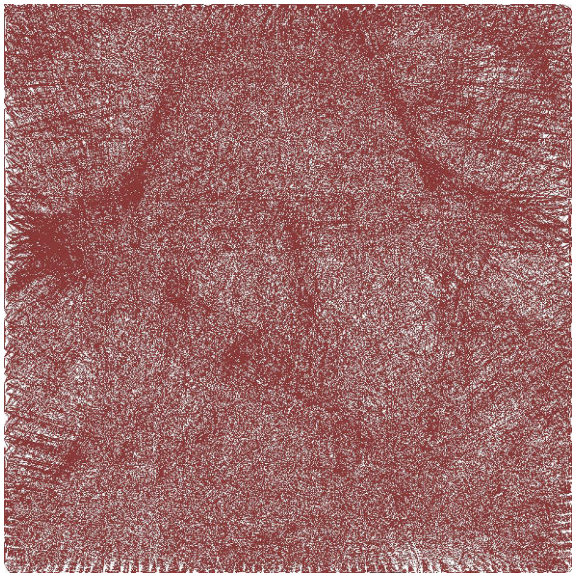
(b) City D L differentiation



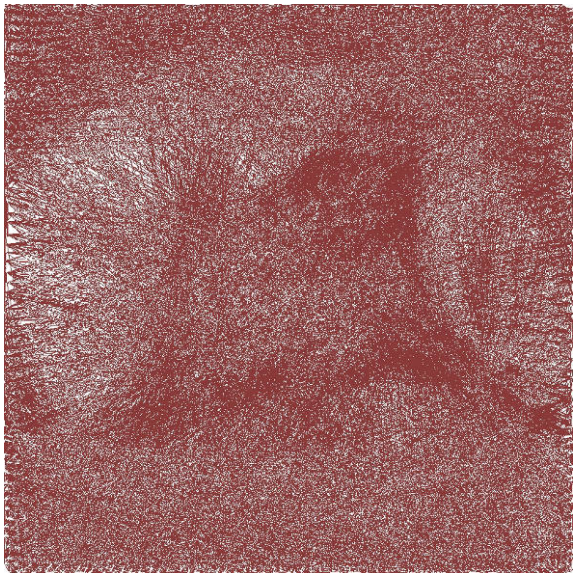
(c) Dance D L differentiation



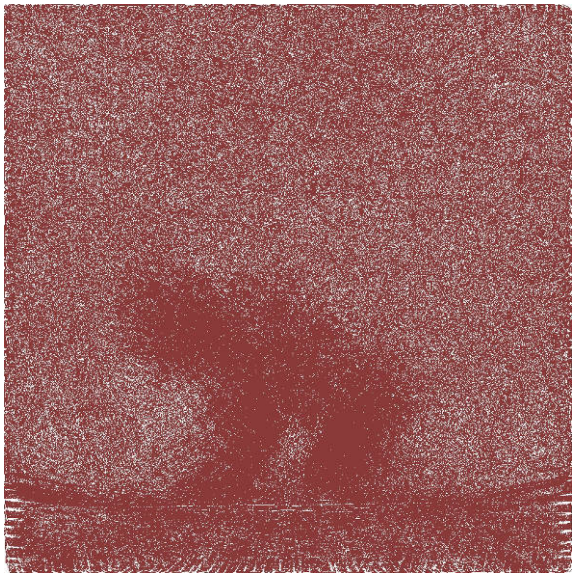
(d) ElGreco D L differentiation



(a) Starik D L differentiation



(b) Tree3 D L differentiation



(c) Tree4 D L differentiation

B Interaction Figures

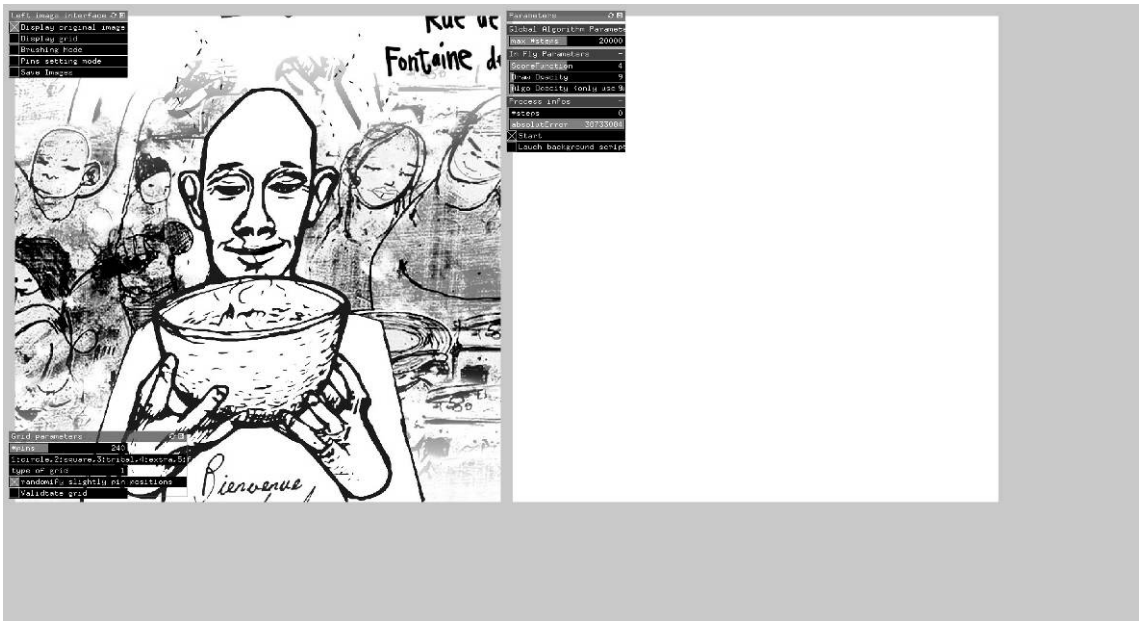


Figure 28: Screenshot: gray interface

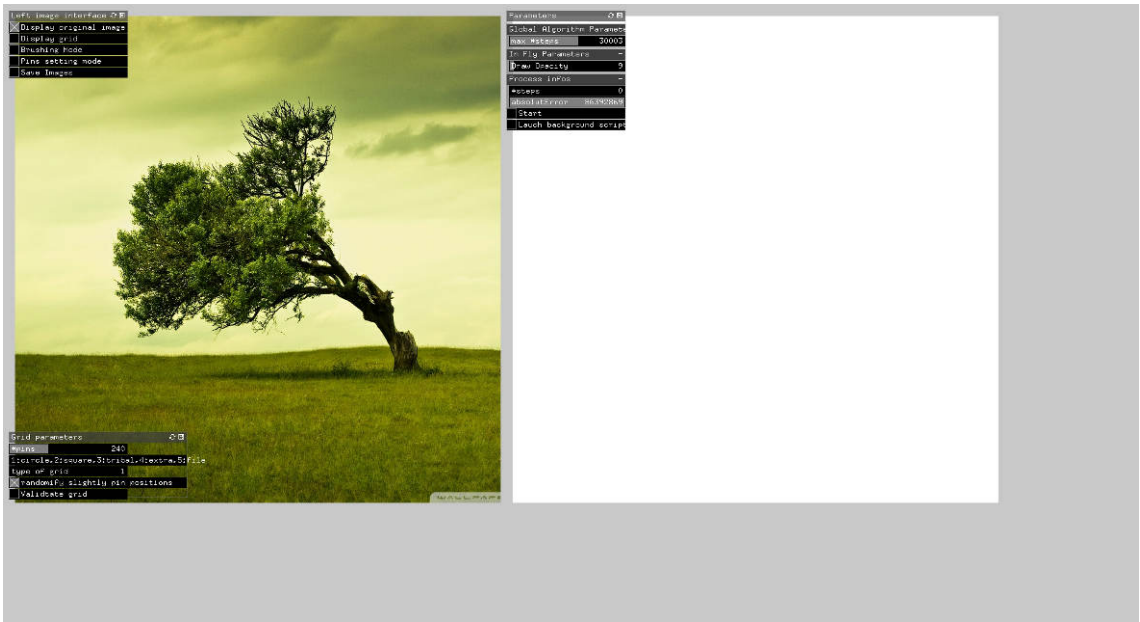


Figure 29: Screenshot: color interface

Figure 30: Screenshot: "brushing mode" interface, (The sunglasses shape is the brushed area)

References

[1] Christian Siegel, *https://github.com/christiansiegel/knitter*

[2] Petros Vrellis, *http://artof01.com/vrellis/works/knit.html*

[3] DDA Algorithm, $https://en.wikipedia.org/wiki/Digital_differential_analyzer_(graphics_algorithm)$

[4] Real knitted art work produce by Christian Siegel Algorithm, *http://imgur.com/gallery/pN5T9*

[5] Code repository, *https://github.com/MaloDrougard/knit*