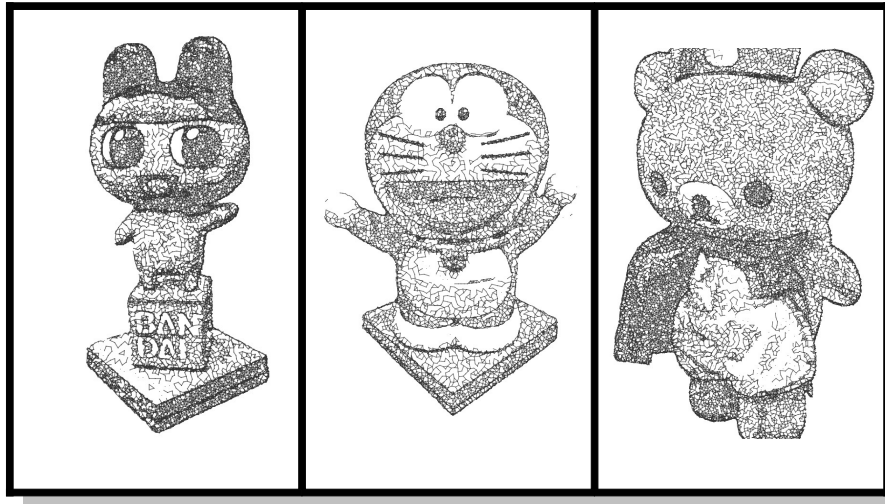


Computer Generated String Art

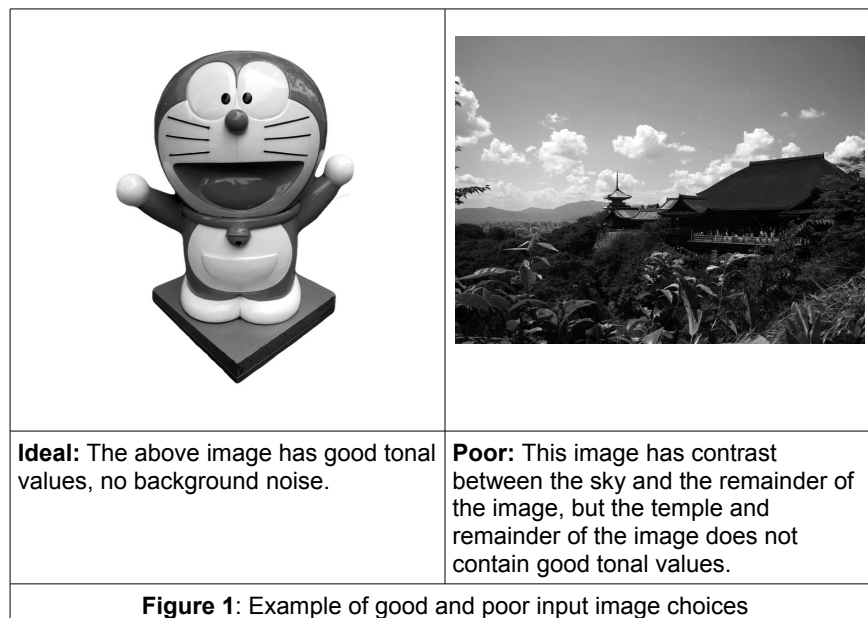


Amanda Manarin

<http://g6mandicsc490.wordpress.com/>

Artists are constantly using interesting mediums to create innovative art. Kim Kamensⁱ, created a series of portraits consisting of nails and strings; nails are densely connected in areas of low intensity and loosely connected in areas of high intensity. The goal of this project was to create a computer program that can create images similar to Kamen's art. The question is quite simple, given a grayscale image, how can we simulate the intensity of the image using points and lines?

Before we input our image into the program, we must ensure that the image is a good candidate. Firstly, the input image should be converted to grayscale. Secondly the image must contain good tonal values, there should be contrast between areas/features of the image, rather than the image being one consistent tone. Figure 1 shows an example of a good input image, and a poor input image.



Once an ideal image has been selected, the user can begin using the StringArt program. Figure 2 shows the main UI of the StringArt application. Input images must be stored within the directory Models/Images under the directory where StringArt.exe is stored. To load an image into the application, the

user enters the name of the image, e.g.: "image.png" into the "File Name" box. The user may also set the desired size of the output image. For instance by entering an image size of 800, the larger dimension of the input image will assume the length of 800 pixels.

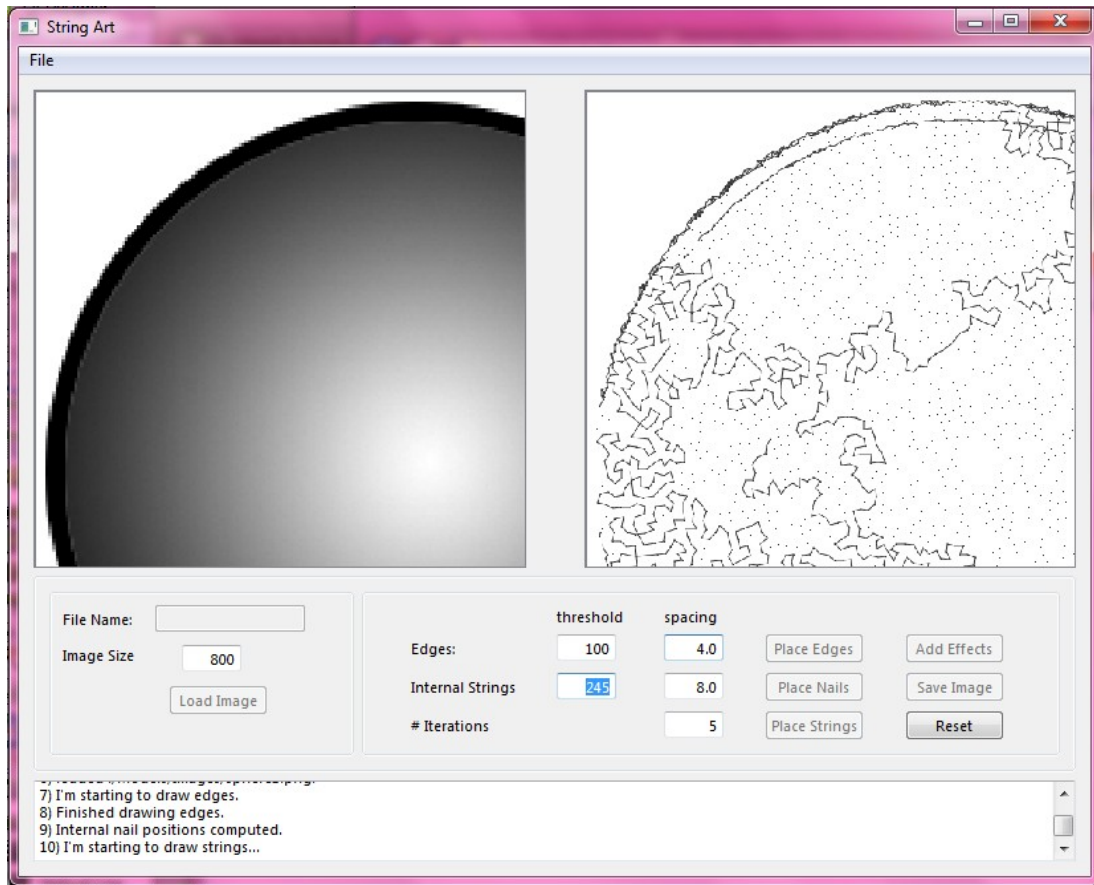


Figure 2: The main UI of the StringArt application

Now the process of converting our image into String Art can begin. Edge detection must be performed first. This helps identify the feature lines in the image and creates a more appealing result. After examining Kamens' art, I noticed that nails were placed densely along the feature lines of her images. The edge detection algorithm used was simply the Sobel algorithmⁱⁱ. The user may specify the threshold for edge detection, a higher threshold will display more feature lines, while a lower threshold will only display the most prominent feature lines. The user must also specify the desired distance between nails along the edges. Once these two parameters have been set, clicking on "Place Edges" will perform the following steps

- 1) Perform Sobel edge detection on the input image
- 2) Create a grayscale image where the background is white and edges are black-gray
- 3) Create a pure white image, this will be our "output" image
- 4) Place nails along the edges ("spacing" units apart) in the grayscale image, whose intensity values are less than the threshold specified by the user. Store the results in the output image. (Details of the nail placement algorithm will be described shortly)
- 5) Display the results to the user.

Once the edges have been drawn it is time to place the nails on the remainder of the image. The user may specify a threshold for nail placement in the box next to "Internal Strings" and under "Threshold". This causes the algorithm to only place nails in regions where the intensity is less than threshold. The user may also specify the spacing of the nails, this causes the algorithm to place nails with even spacing throughout the image. Initially, I thought that placing nails evenly throughout the image was a valid approach. Upon examining Kamens' art again, I noticed that nail spacing is not consistent; Nails are densely placed in low-intensity areas and sparsely placed in high-intensity areas (otherwise known as

stippling). I first used the uniform spacing approach to generate imagery, but later implemented stippling. The later produced imagery that more closely resembled the input image.

Once the user hits "Place Nails" the program does the following (in the case of even spacing):

- 1) Using the ANN Library, creates a kdTree where each pixel (with intensity less than threshold) is a node in the kdTree tree.
- 2) Create a vector of all the nodes in the kdTree called `_nail_positions`, shuffle the order of these nodes.
- 3) For each node in `_nail_positions`:
 1. If it is not marked for deletion
 1. Perform nearest neighbour search with a radius of `s` units (where `s` is the spacing entered by the user)
 2. Mark each of the nearest neighbours for deletion
 3. Store the current node in a vector `_sparse_positions`
 2. If it is marked for deletion, just skip it and move on to the next node.
- 4) Let `_nail_positions = sparse_positions`
- 5) Place a nail on each position in `_nail_positions`.

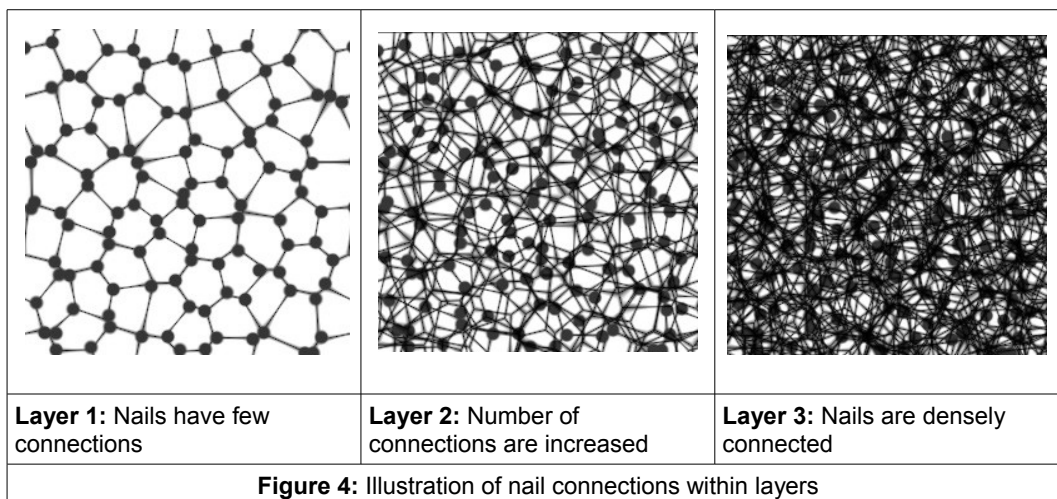
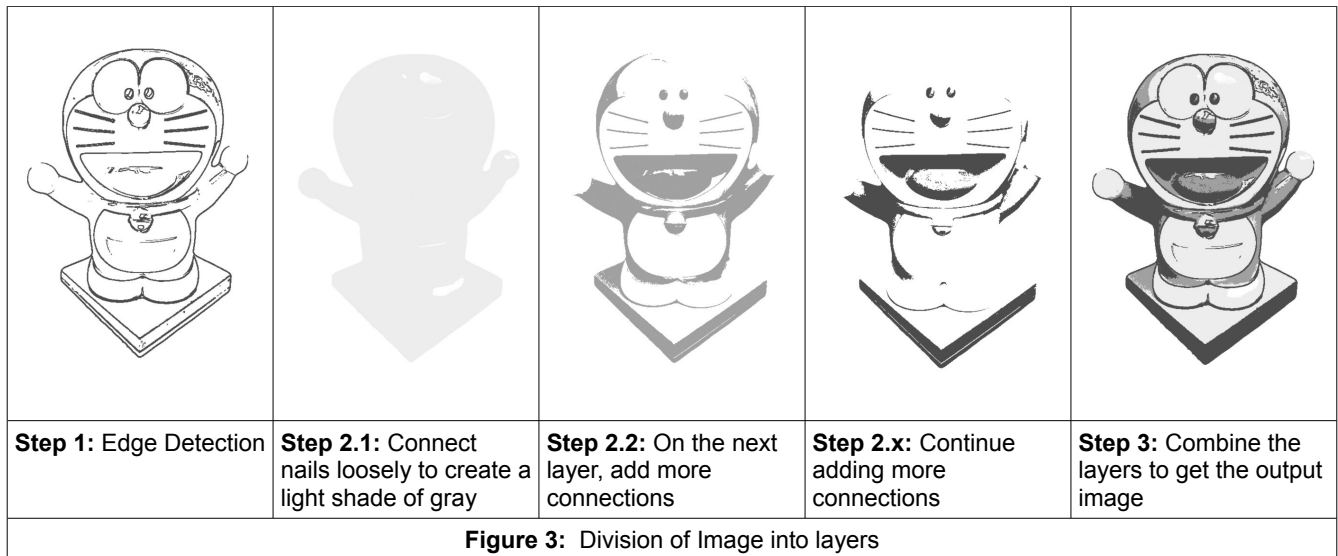
In the case of using a stippling approach we need the user to input one more value, this is the "# of Iterations". This essentially breaks the images into `x` layers, where each layer only contains the pixels whose intensities lie within the range of the layer. For instance, if the user specified threshold is 245 and the user specifies 3 iterations, then we split the image into 3 layers, where layer 1 contains pixels with intensities between 0 and 82, layer 2 contains pixels with intensities between 82 and 164, and lastly layer 3 contains pixels with intensities between 165 and 245. For each layer, we create a separate temporary image, containing the layer's pixels, and we perform the above algorithm on each layer, except we decrease our search radius (by 1 unit) each time.

The last step in the String Art creation process is to connect the nails. Here we also use the field "# of Iterations" to determine the number of layers we need to create. In this part of the algorithm, when we mention layer we consider all pixels below a certain threshold. Using the above 3 layer example, layer 1 would contain all pixels with intensity less than 82, layer 2 contains all pixels with intensities less than 164 and layer 3 contains all pixels with intensity less than 245. The algorithm first starts with the lightest layer, for instance all pixels with intensity less than 245. All nails within this layer are only allowed 2 connections, creating the illusion of very high intensity. Then we look at the next layer, and allow all nodes within this layer to have 3 connections, creating the illusion of a darker intensity. We repeat this process for each layer, increasing the number of connections allowed per node each time. This works since nails in low-intensity areas will have many connections creating the illusion of darkness, while nails in high-intensity areas will have few connections, creating the illusion of lightly shaded areas. The process for connecting nails is as follows:

1. Start with the lightest layer
 2. `numCon = 2` (max number of connections allowed per node for current layer)
 3. For each layer:
 1. Iterate through the nails in `_nail_positions`
 1. If nail `n` is in the current layer and `n`'s number of connections `< numCon`
 1. `grow_string(n, numCon)`
 2. `numCon++` // The number of connections allowed per node increase per layer
- `grow_string(Nail n, int numCon)` // Form a string starting at nail `n`
1. Perform nearest neighbour search for `n` (with a radius of `2*spacing` to return many neighbours)
 2. For each neighbour `m`:
 1. If `m` is in the current layer, `m`'s number of connections `< numCon` and `edge(n,m)` does not exist:

1. Create an edge (n,m)
2. Add edge (n,m) to the output image
3. Call grow_string(m) (continue forming a string from nail m).
2. If m is not in the current layer, break and move on to the next neighbour

This process is mainly a brute force algorithm, to improve run-time, in each layer I only look at about half of the nails. I observed that once half the nails have been examined, the majority of the current layer has been explored. However run-time is still extremely long and takes around 5 minutes for the entire process using uniform nail spacing and 20 minutes using non-uniform nail spacing (tested on a MacBook Pro with Intel Core 2 Duo 2.2ghz processor, 4GB RAM). A more visual illustration of the process is depicted in figures 3 and 4



Results



Figure 5: Input Image

Figure 6 depicts the results obtained with uniform and non-uniform nail placement. Figure 7, displays the image that is created after the user clicks on “Add Effects” in the UI. A gradient is placed on the background, and nails are placed at each nail position. Nails (in figure 8) were created in Maya, one nail was rendered from 6 different angles. Lastly, a blurring effect is performed on the strings (using Gaussianⁱⁱⁱ blur) and placed on top of the gradient (but under the strings and nails) to create the illusion of a shadow. The goal here was to simulate Kamens' original art.

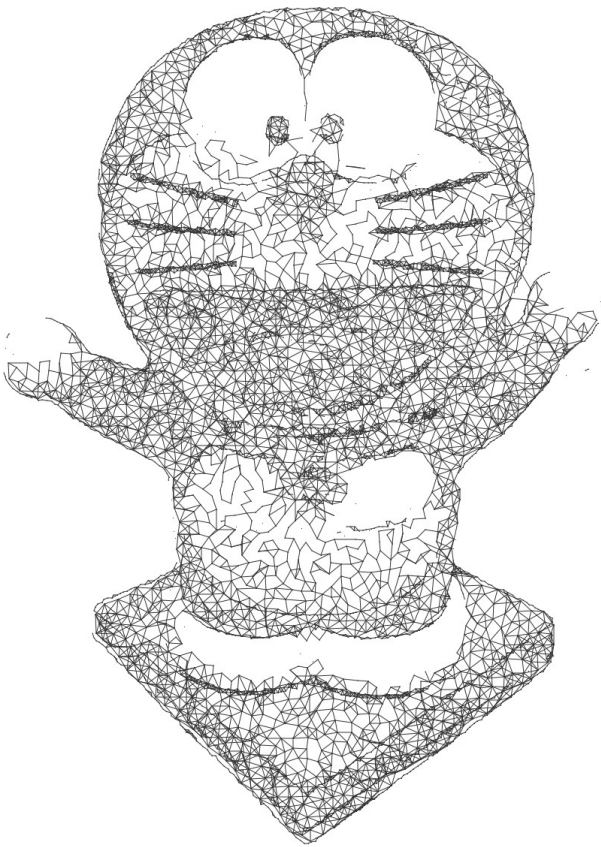
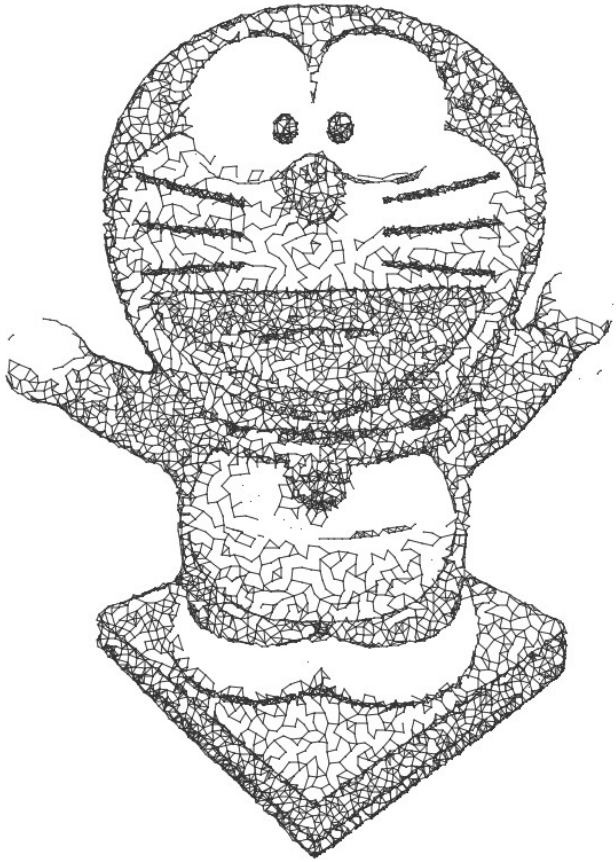
 A wireframe representation of the Doraemon character. The character is composed of a dense mesh of small, uniform triangles. The overall appearance is somewhat flat and lacks depth or shading.	 A wireframe representation of the Doraemon character, similar to the left image but with a different mesh structure. The triangles are larger and more irregularly shaped, particularly around the face and body, giving it a more textured and three-dimensional appearance.
<p>Nails in this image were uniformly placed</p>	<p>Nails in this image were non-uniformly placed, noticed how this image has good contrast and good tonal values. It is an overall better representation of the input image.</p>
<p>Figure 6: Example Output Images</p>	



Figure 9: Input #2

Figure 10 shows the result of our second input image. Many more (some impressive) examples are available online at <http://g6mandicsc490.wordpress.com/>



Limitations/Future Directions

The main limitation right now is processing time. As described above it takes around 5 minutes on an image of 1000 pixels in width using uniform nail placement, while up to 20 minutes with non-uniform nail placement. The large jump in processing time is due to the exponential increase in the number of nails, therefore a more efficient algorithm for connecting nails needs to be devised.

The program does not also handle complex images such as landscapes well. I believe this is largely do the “layer” approach I am using to connect nails. The layer approach causes some small details in landscapes (leaves, grass, flowers) to be lost, especially if some of these details get lumped together into the same layer.

Source

Source code is also available online. `StringArt_1.1.zip` contains the code that performs non-uniform nail placement (use at your own risk), while `StringArt_1.0.zip` is a more stable version and contains the code that performs uniform nail placement.

- i Kim Kamens' Art: <http://www.kimkamens.com/Thread-Series.html>
- ii Sobel Edge Detection: <http://www.pages.drexel.edu/~weg22/edge.html>
- iii Gaussian Blur: <http://www.gamedev.net/reference/programming/features/imageproc/page2.asp>