CSE 559A - Final Project

Depixelizing Pixel Art

Isaias Suarez

December 2016

1 Introduction

Pixel art is a form of digital art where the details in the image are represented at the pixel level. Graphics in computer and video games in the 1990s consist mostly of pixel art, where artists were forced to work with a limited color palette and had to arrange every pixel by hand due to hardware constraints.

As a result, pixel sprites had the task of representing objects using only a small amount of pixels, meaning pixel art carries a maximum of expression and meaning per pixel. Many classic games like Mario or Space Invaders are ingrained in the cultural memory due thanks to charming pixel art renditions. People can continue enjoying these classic games using emulators built to run these old games.

Most vectorization techniques are designed for natural images are based on segmentation and edge detection, which are not well suited for the tiny features present in pixel art. I have attempted to implement an algorithm by Johannes Kopf and Dani Lischinski for depixelizing pixel art [3]. This algorithm is designed to best represent all the features present in pixel art when computing a vectorized output for the original image.

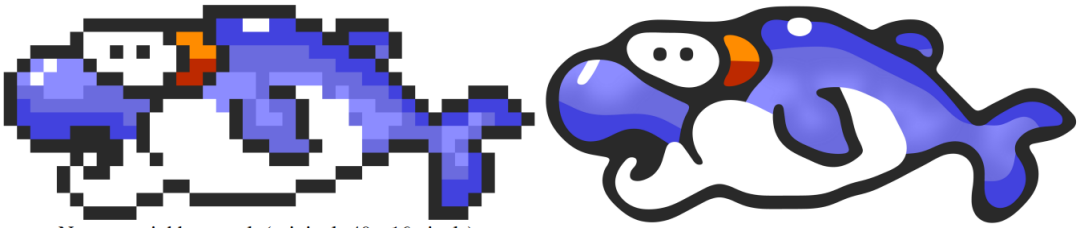

Figure 1: Left: The original pixel art image. Right: The algorithm's intended output.

2 Related Work

Previous work related to this algorithm can be classified into three categories. I will go over some techniques in each category, but for more thorough information please refer to the original paper. As full disclosure, all the information in the section is pretty much lifted from the original paper [3].

2.1 General Image Upscaling

The standard approach to image upscaling is to apply linear filters such as Nearest-Neighbor, Bicubic, and Lancosz [5]. These filters make no assumptions about the data aside from it being band-limited. Images upsampled in this manner typically suffer from blurring of sharp edges and ringing artifacts.

PhotoZoom 4 (General Image Upsampling) hq4x (Specialized Pixel Art Upscaling) Adobe Live Trace (Vectorization)

Figure 2: Results from the different categories of algorithms.

2.2 Pixel Art Upscaling Techniques

A number of specialized pixel art upscaling algorithms have been developed over the years within the emulation community. None has been published in a scientific venue; however, open source implementations are available for most. All of these algorithms are pixel-based and upscale the image by a fixed integer factor.

The latest and most sophisticated evolution of this type of algorithms is the hqx family [4]. This algorithm examines 3x3 pixel blocks at a time and compares the center pixel to its 8 neighbors. Each neighbor is classified as being either similar or dissimilar in color, which leads to 256 possible combinations. The algorithm uses a lookup table to apply a custom interpolation scheme for each combination. This enables it to produce various shapes, such as sharp corners, etc. The quality of the results is high. However, due to its strictly local nature, the algorithm cannot resolve certain ambiguous patterns and is still prone to produce staircasing artifacts. Lookup tables exist only for 2x, 3x, and 4x magnification factors.

2.3 Image Vectorization

Various commercial tools exist such as Adobe Live Trace [2] for performing automatic vectorization of raster images. Because these are proprietary tools, the underlying implementation are not disclosed.

3 Technical Description

My implementation of Kopf's algorithm closely follows what is detailed in the paper, but within the paper there are a few areas that were left vague and I had to write code based on my own interpretations of the instructions. I believe there might be a few places where I interpreted the algorithm incorrectly, particularly when it comes to fitting spline curves and I will discuss these sections in detail.

The algorithm can be subdivided into four main parts: reshaping the pixel cells, extracting spline curves, optimizing the spline curves, and rendering the vector output. I managed to implement reshaping pixel cells fully and rendering the vector output, but I had a lot of trouble with extracting spline curves, and subsequently I was unable to do the optimization step. In this section I will specifically discuss what I have written code for and will discuss the unimplemented parts of the algorithm in the future work section.

I wrote all my code using Python 2.7 on a Linux system. I use a some preexisting modules for the more general parts of the algorithm: PyPNG for reading in the input file, NetworkX to represent the underlying graph structures, Mathplotlib.pyplot to output debug graphs, and svgwrite to output the vector file.

3.1 Reshaping the pixel cells

Before we can actually reshape the pixel cells, we need to read in the file. I used PyPNG to read a png image and extract the RGB data. Each pixel holds data for its location, RGB, and YUV values. For each pixel I created a node in the graph that mapped to the pixel data, then set up initial connections for each node to the eight adjacent pixel nodes surrounding it.

Once our graph is initialized, we remove edges between dissimilar pixels. Pixels are considered dissimilar if the difference in their Y, U, V values are $\frac{48}{255}$, $\frac{7}{255}$, or $\frac{6}{255}$ respectively. This gives us an initial similarity graph with

crossing edges. The next step is to resolve these crossing edges to figure out which diagonal connections should be preserved.

To resolve these crossing edges, we examine all 2x2 windows of pixels and add up the weights of three heuristics to figure out which edge to keep, keeping the heavier weighted component (if a 2x2 window is fully connected we remove both edges). If the weight of two edges is the same we remove both of them. The heuristics are as follows:

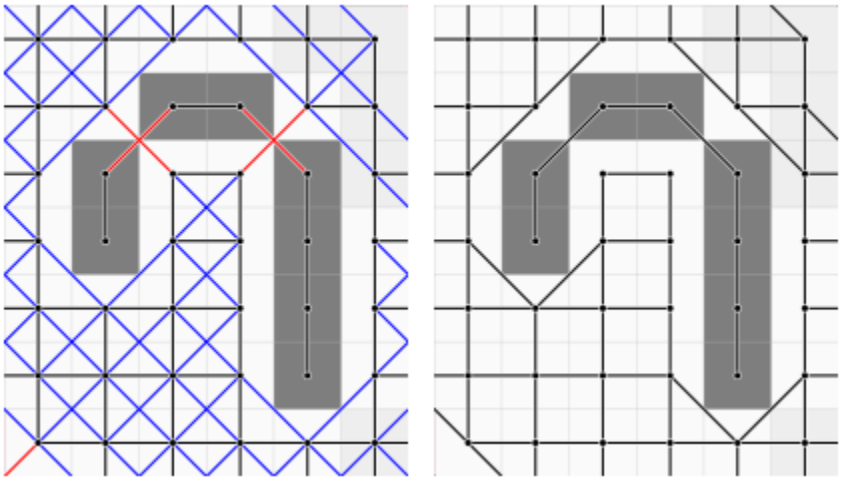

Figure 3: An example from the original paper illustrating crossing edges and the result after resolving them.

Curves If pixels are a part of a long curve feature, we want to connect them. We compute a weight equal to the length of connected valence-2 nodes. I use a simple DFS to walk through the valence-2 nodes connected to a diagonal and just count up the number of nodes visited.

Sparse pixels In two-colored drawings, humans tend to perceive the sparser color as the foreground and the more abundant color as the background. We compute a weight by placing an 8x8 windows around the diagonals and counting up the number of connected components of each diagonal. I use another DFS to count up the nodes connected to each diagonal. Once the connectivity is determined, the actual weight is the size difference between the two components.

Islands We want to minimize the number of small disconnected islands, so if we have a valence-1 nodes connected to a diagonal, this heuristic contributes a fixed weight of 5 to the diagonal.

Once we resolve all the crossing edges, we now have a planar graph. Using this planar graph, we can construct a Voronoi diagram around the union of each node and half the edges connected to them. The generation of the Voronoi diagram can be quickly computed due to the limited possible arrangements of each node in the graph, allowing each cell to be computed in constant time since we only need to look at the neighbors surrounding a node (3x3 window).

The generalization of the Voronoi diagram is left a bit vague in the original paper, so I did my best to come up with a decent constant time generalization for a Voronoi cell. My solution places points at quarter pixel locations if it is connected to a given neighbor, and at half points if there is no connection. Once these points are computed, we take the convex hull as the Voronoi cell. Our end result is a graph of Voronoi cells that can be interpreted as the new color boundaries for each pixel.

The original paper states that we can simplify the graph by collapsing all valence-2 nodes. I found this statement a bit vague and was unsure how to exactly collapse the valence-2 nodes, so I proceeded with the algorithm without collapsing valence-2 nodes. I try to collapse valence-2 nodes later when computing visible edges (see section 3.2)

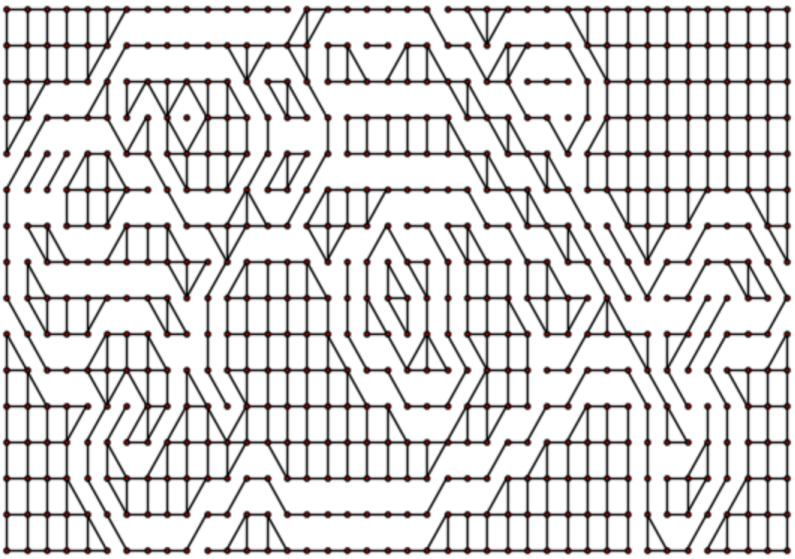

Figure 4: The debug graph from my implementation. This shows the planar graph with all crossing edges resolved.

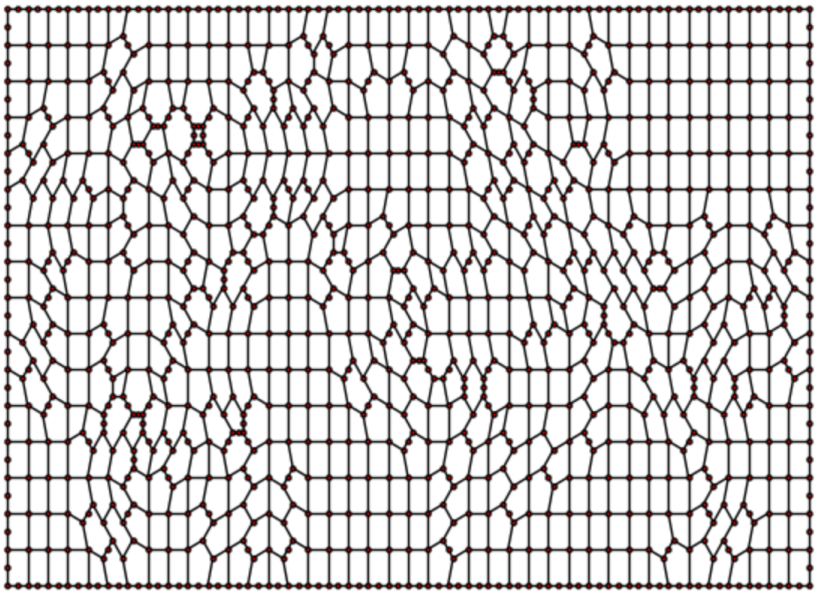

Figure 5: The resulting Voronoi cell graph from my implementation.

but that didn't really seem to help with the end result. As a result, this part of the algorithm may have to be revisited if I continue work on this in the future.

3.2 Extracting Spline Curves

The reshaped pixel cell graph resolves all connectivity issues and encodes the rough shape of the object, but the rough shape contains sharp corners and may look a bit blocky. To get around this, we identify visible edges where significantly different colors meet (using the same conditions for the YUV values from earlier).

To identify these visible edges, I walk through each Voronoi node and compare it to the adjacent Voronoi node neighbors. If the pixel data between the node and neighbor are different, then we follow the path of valence-2 nodes connected to that neighbor. This gives us a visible edge (we ignore neighbors that are already part of an existing visible edge to prevent retreading already computed paths).

If a node has two visible edges, I go ahead and combine them. If a node has three visible edges, we combine two of them into one to resolve the "T-junction". To resolve the T-junction, we categorize each of the three edges as either a shading edge or a contour edge. An edge is a shading edge if two pixel cells have a YUV distance of $\frac{100}{255}$. If a T-junction has two contour edges and one shading edge, we combine the contour edges. Otherwise, we connect the pair that has the angle closest to 180 degrees.

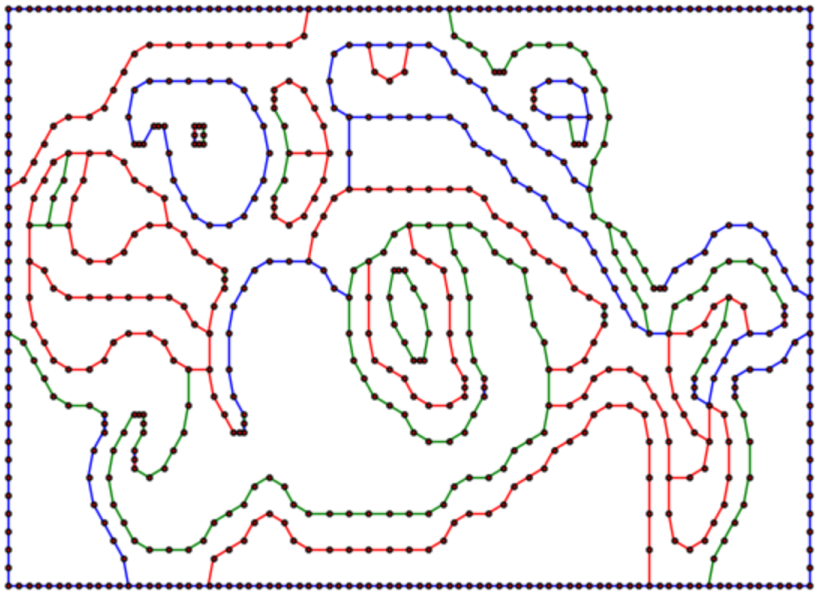

Figure 6: The visible edges that we extract from the Voronoi graph. I cycled through red, green, and blue to arbitrarily color the edges to provide a rough view of how the edges are divided.

Once we have our visible edges, the paper says to fit quadratic B-spline curves [1] to them with the control points initialized to the node locations. Most of my trouble resulted from not understanding what these node locations actually were. I tried to place the control points on the center of the Voronoi cell for the visible edge, but that did not produce good results. As of now I still don't know where the control points should be, and I was unable to properly fit quadratic B-spline curves on the visible edges.

3.3 Rendering

I used the svgwrite Python module to output a vector file. My program supports three different kinds of svg vector outputs; pixel, line, and spline. The pixel output just takes the Voronoi graph and outputs that with the colors set to the data stored in the pixel node. The line output renders the visible edges as black on a white canvas. The spline output shows the attempt at initializing the control points to the Voronoi node centers and using the color value stored there. In trying to output the spline curves, it's worth noting that I used quadratic Bezier curves for a more generalized test output (the svg format natively supports Bezier curves).

In the line and spline outputs, I attempt to collapse valence-2 nodes by removing all the collinear points. I'm not sure if that actually collapses valence-2 nodes.

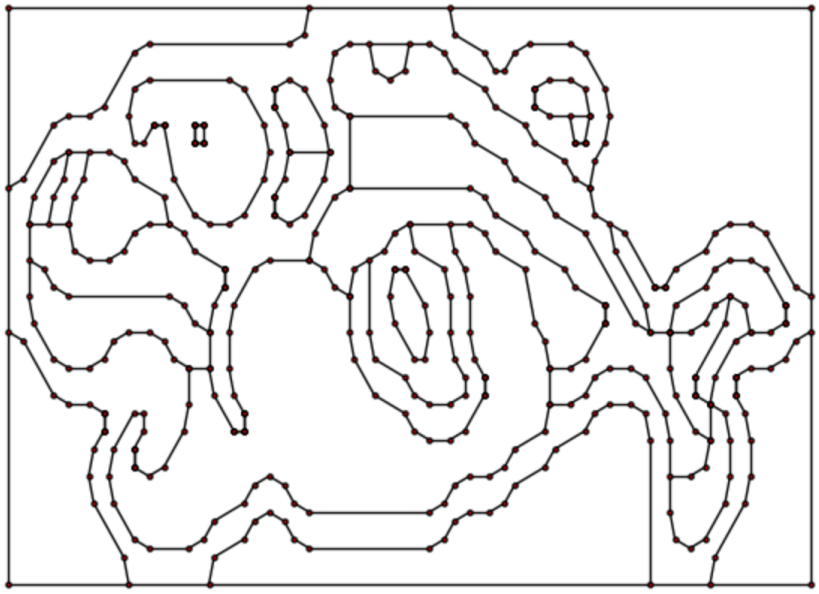

Figure 7: The visible edge graph with collinear points removed.

4 Experimental Results

I ran my program against a set of pixel art sprites pulled from various classic games. I show the original image, the line output, spline output, and pixel output.

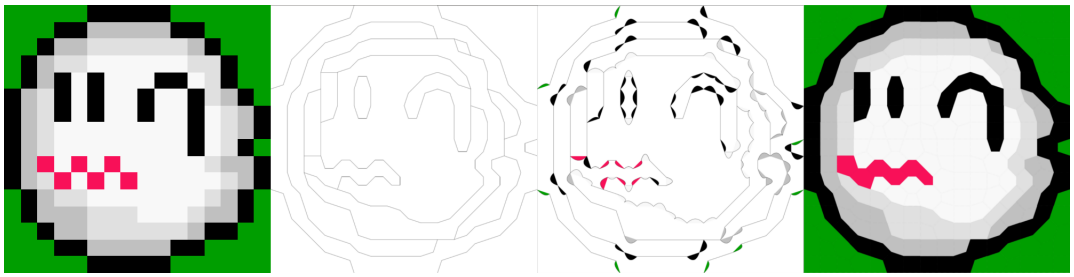

Figure 8: The Boo enemy from Super Mario World (SNES).

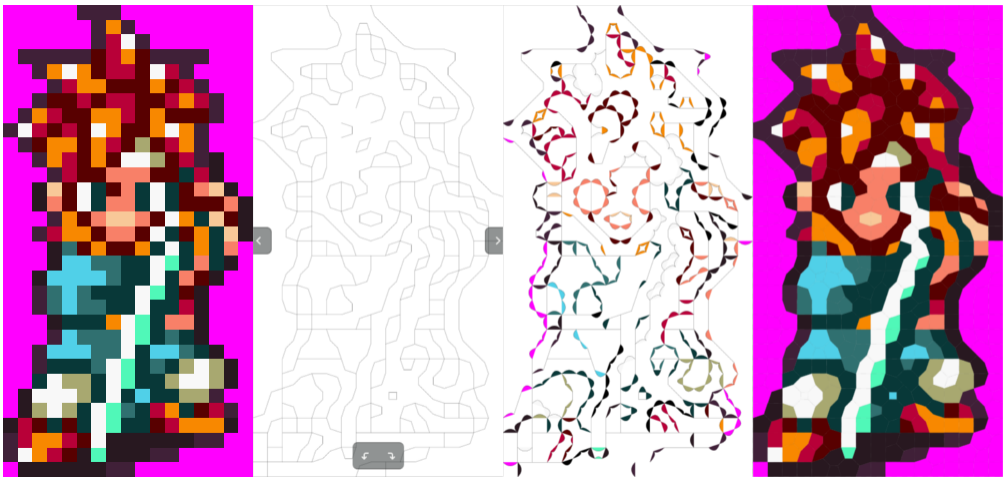

Figure 9: Crono from Chrono Trigger (SNES)

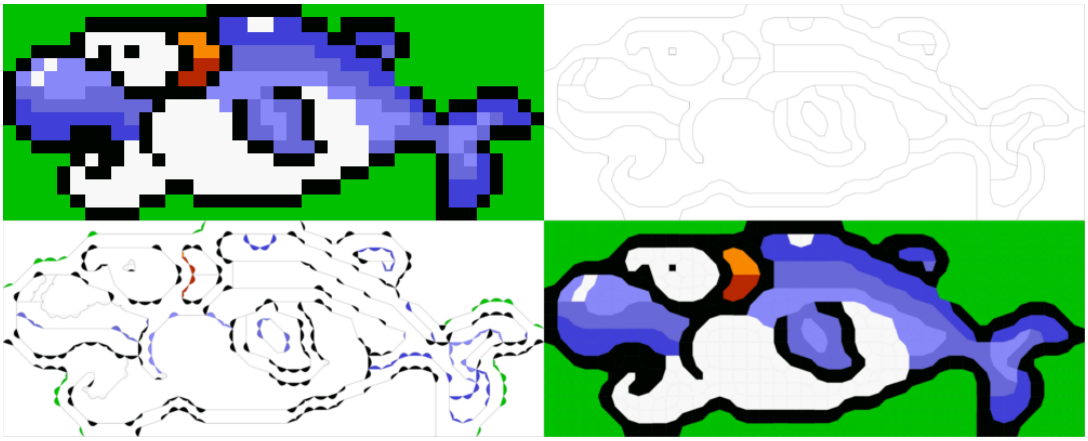

Figure 10: The Dolphin from Super Mario World (SNES).

Figure 11: The Alien from Space Invaders (arcade).

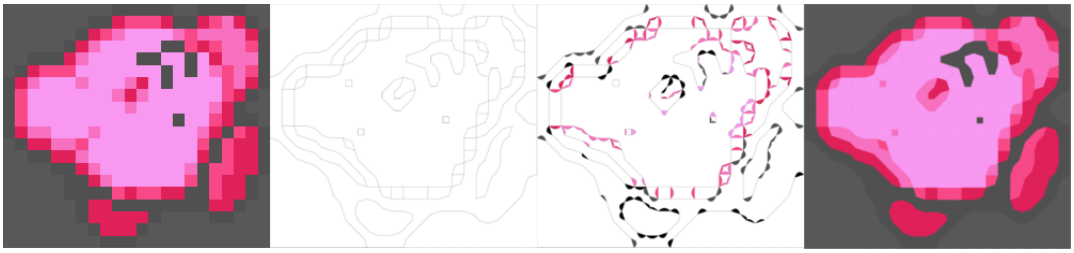

Figure 12: Kirby from Kirby's Dream Course (SNES).

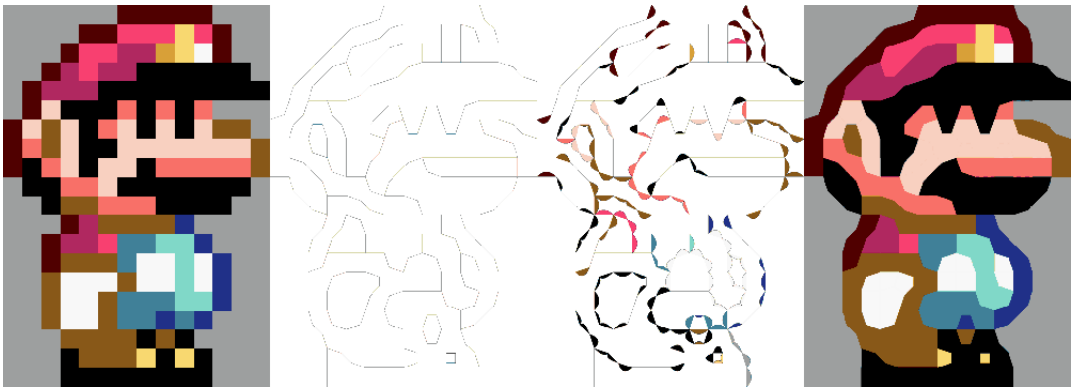

Figure 13: Mario from Super Mario World (SNES).

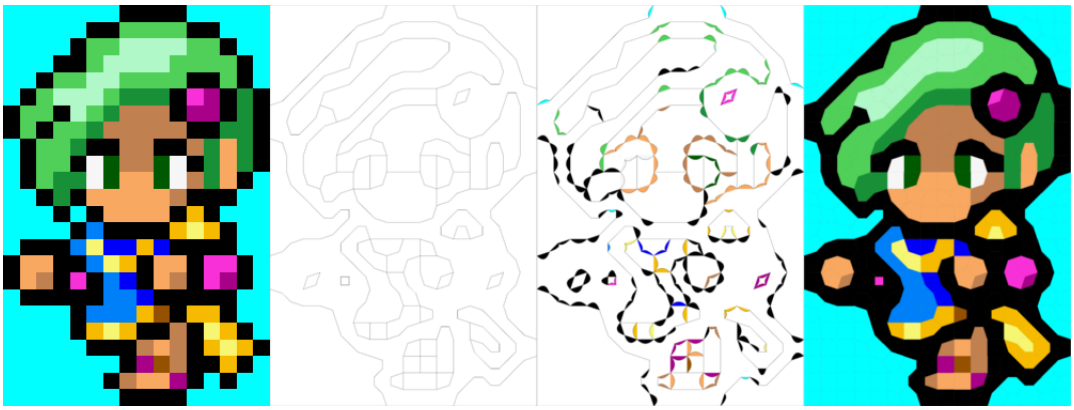

Figure 14: Rydia from Final Fantasy IV (SNES).

5 Discussion

5.1 Results

Despite my implementation being incomplete, I believe the colored in Voronoi diagrams (pixel outputs) still give decent results. The pixel outputs provide a rough approximation of the original pixel art image, and in the case of Mario (Figure 13) and Rydia (Figure 14) the output is reasonably smooth.

The algorithm does a reasonable job on pixel art that is relatively limited in the color palette and does not have complex shading. The Boo (Figure 8) does not look very good because most of its design consists of shaded edges which end up looking blocky in the pixel output. This might be fixed in the future by running a Gaussian blur that does not cross over visible contour edges in the image.

The algorithm runs poorly on pixel art with high level of detail like Crono (Figure 9). I believe that even if the algorithm is fully implemented, the vectorized image of Crono would still look bad as the original pixel art is as dense in detail as the Doomguy output in the original paper (Figure 11 in Kopf).

Kirby (Figure 12) and the Dolphin (Figure 10) could be smoother and would benefit from proper spline fitting and optimization.

Interestingly, the Alien (Figure 11) does not have its antennae or legs attached to its body. This bug is probably a result of the heuristic for sparse pixels is not handling the case where the background has fewer pixels than the foreground. The original paper seems to handle this edge case alright, but does not detail how it got around this problem. A potential fix is to widen the bounding edges and give the connectivity graph dummy nodes to increase the size of the background.

5.2 Complexity Analysis

We can read in our data to get a set of $O(n)$ pixels.

Initializing the connectivity graph only requires a walk through each pixel of the image, and resolving crossing edges only requires a walk through using a sliding window of constant size. Solving the curve heuristic in the worst case may go through all pixels, but this would require every pixel to have valence-2 connectivity, and that is highly unlikely. The expected time to solve the heuristics should be constant. Therefore, creating the connectivity graph takes $O(n)$ time and space.

Since we can generalize our Voronoi diagram to draw the cell using only the immediate neighbors, each cell is generated in constant time and we do this for each cell. Thus, generating the Voronoi diagram also takes $O(n)$ time and space.

In extracting spline curves, we walk through all the pixels to find visible edges, but we do not walk through a single visible edge more than once. Once we generate our visible edges, we walk through each edge a single time to resolve T-junctions. Overall, extracting visible edges takes $O(n)$ time and space.

Rendering the output for the pixel output only requires a walkthrough of each Voronoi cell. Rendering line and spline outputs only requires walking through each visible edge. Therefore, rendering an output takes $O(n)$ time.

Overall, my implementation of the algorithm takes $O(n)$ time and space.

6 Future Work

In the future I would like to figure out how to actually fit the quadratic B-spline curves to the visible edges. Once I manage to figure that out, then I can optimize the control points for those curves to match the results seen in the paper. I would also like to experiment with different Voronoi diagram parameters to see how that would affect the vectorized image.

7 Source Code

All the source code that I wrote can be found at https://github.com/ijsuarez/depixelize.

References

[1] DE BOOR. *A Practical Guide to Splines*. Springer-Verlag, 1978.

[2] ADOBE INC. Adobe illustrator cs5, 2010. http://www.adobe.com/products/illustrator/.

[3] D. KOPF, J; LISCHINSKI. Depixelizing pixel art, 2011. http://johanneskopf.de/publications/pixelart/paper/pixel.pdf.

[4] M. STEPIN. Hqx, 2003. http://web.archive.org/web/20070717064839/www.hiend3d.com/hq4x.html.

[5] G. WOLBERG. *Digital Image Warping, 1st ed.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1990.