Raster to Vector

Andrea Wong, Isabella Siu

Abstract:

One of the major issues with working with pixel art is that it often scales poorly. An image that has been scaled up and down multiple times tends to have jagged and blurry edges, and the separation between colors isn't as sharp. Vector art, however, can be scaled up and down without loss of sharpness due to the fact that it is stored based on the placement of control points and curves connecting them. The cutting edge paper on converting pixel art to vector art is "Depixelizing Pixel Art" by Johannes Kopf and Dani Lischinski, which is what we have started implementing. Our implementation can change pixel art into an artist editable SVG that can be scaled up and down without problems.

1 Introduction

A major problem with using pixel art is that it does not handle scaling well. Once a pixel image has been scaled up and down several times, the edges become jagged and blurry. Similar colors start to blur as well. An example of this is when you try to scale the sprites from old school games like Super Mario. The sprites are often small and their lines become increasingly blurry the more you try to scale them up (Figure1). This becomes frustrating when trying to create images where you might not be sure of the exact scale that you want.

Figure 1: King Boo scaled from 18x18 to 128x128 pixels

In contrast, vector art can be scaled up and down multiple times without losing its crispness because rather than being based on individual pixels, the image is based on the placement of control points and equations of curves used to connect those points. Another advantage of vector art is that the artist does not have to worry about every single little pixel making up the image. Instead, changing a control point changes a small region of the edge making up the image. If pixel by pixel control is desired, it is easy to export from a vector drawing software such as Inkscape or Adobe Illustrator to a png to edit in a raster drawing software, however the converse is not true. Many raster drawing software do not have the option to export to svg.

We had two motivations for this project. The first was to create a vector image from a pixel image that looked reasonably like the pixel image. The second was to be able to create an editable svg image that an artist could open in Inkscape or Adobe Illustrator and make any desired manual changes to the image.

2 Prior Work

The main technique we looked at was in "Depixelizing Pixel Art" by Johannes Kopf and Dani Lischinski [3], which we used as a reference while creating our program. While we followed their section 3.2, on reshaping the pixel graph, rather faithfully, we've taken liberties with their algorithm after that. In addition, we looked at a few other papers for clarity and guidance.

"Potrace: a polygon-based tracing algorithm" by Peter Selinger [4] goes over

transforming bitmaps into a contour based on Bezier curves using polygons as an intermediate step. First they transform a bitmap into a series of paths dividing white and black pixels. Then polygons are generated for the closed paths and the optimal polygon is found for a path. These polygons become Bezier curves and straight line segments. While this method uses Bezier curves, unlike the paper we implemented, which uses B-splines, this paper is cited in the “Depixelizing Pixel Art” and was a useful resource as many of the concepts presented in it are similar to what “Depixelizing Pixel Art” described, just for bitmaps instead of full color images. This paper was mentioned in our implemented paper as an inferior predecessor.

More directly, we referenced the Github project Pixel-Art by Vemula and Yeddu [6]. They also implemented “Depixelizing Pixel Art” and while we mostly referenced it for a more concrete example of things we were confused about in the original paper, we were also inspired by them to use shapes to create the different colored regions as opposed to the method described in the original paper, which makes B-splines and diffuses color in those regions from the centers of their original cells. Our more complicated test cases also came from their repository.

Also, we used simple-svg by Shavit, a C++ library for writing to SVG files [5]. It is a single file, header only C++ library, and thus was very easy to incorporate into our code. While Shavit has not actually implemented Bezier curves, meaning if we were to add them to our functionality we would have to add them to the library, it was overall very helpful for generating SVG output.

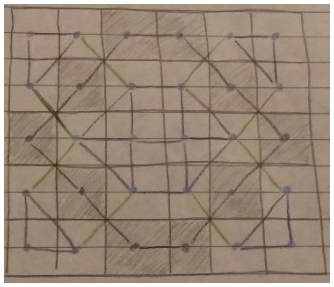

Figure 2: Similarity Graph. Green and purple lines form a cross

Finally, we used code from Barbara Cutler’s Fall 2015 Data Structures homework 9 [2] in order to read in and store the image data from PPM files.

3 Approach

3.1 Similarity Graphs

The first step was to create a similarity graph for the image. This was done by first creating a node for every pixel in the image. Then, all the nodes were given edges to four of their neighbors: up, upper right, right, and lower right. We did this in order to only represent each edge once. Nodes were stored in a width by height vector of vectors, the same dimensions as the original image. Each node object stored its X coordinate, its Y coordinate, and a vector of edges. The edges each had a pointer to their start point node, a pointer to their endpoint node, and their weight.

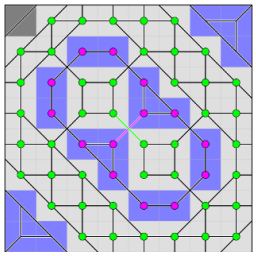

Figure 4: Example of sparse pixels heuristic. Supports keeping the pink diagonal

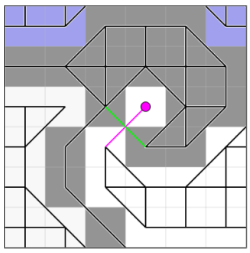

Figure 3: Example of the island heuristic. Supports keeping the pink diagonal because it connects to an island.

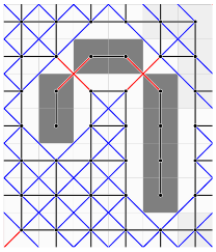

Figure 5: Example of the curve heuristic

Next, we went through and removed connections between nodes with “dissimilar” colors. Kopf and Lischinski considered colors to be dissimilar “if the difference in Y, U, V is larger than 48/255, 7/255, or 6/255 respectively” [3]. We converted our RGB values to YUV to perform the comparison [1]. Then, the nodes were only connected to ones of similar colors. Diagonals between fully connected squares of 2x2 nodes were disconnected in order to simplify the graph.

However, there was still a problem, where the edges of two dissimilar regions form a cross shape (Figure 2). In that case, we had to decide which edge takes priority. There were three heuristics for weighting these edges and determining which should stay.

In the island heuristic, if either of the nodes on an edge had only that edge touching it, then a weight of 5 should be added on that edge (Figure 4). This heuristic prevented us from having too many floating islands on our final image.

In the sparse pixels heuristic, we went over the pixels in an 8x8 square surrounding the cross and counted up the number of pixels in the color of each edge in the cross (Figure 3). The number of pixel of each color were added to the weights of each edge respectively. This was due to the fact that humans tend to view a less represented color as the foreground and a more represented color as a background. Thus when humans creates pixel art, they tend to follow this same rule.

In the curve heuristic, we counted up the number of nodes with valance 2 that this node is connected to (Figure 5). This was to maintain the shape of a curved line, as the longer a curved line was, the more likely it would be important to the final image. We did recursively calculated the length of the curve and this length was the weight given to the edge.

After all of these heuristics were run for all the crosses, we went over them again and removed the edge with less weight from each of the crosses.

3.2 Voronoi Diagrams

The next step of the algorithm was to take the similarity graph and create a Voronoi diagram from it. The Voronoi diagram used in our implementation is a bit different from the normal Voronoi diagram however. A normal Voronoi diagram maps points to their closest center. Our Voronoi diagram mapped area to edges, based on the placement of edges in the similarity graph.

In our Voronoi Diagram, the node at the center of each pixel was assigned a

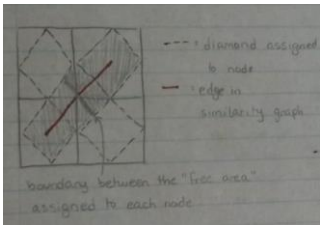

Figure 8: Assignation of eight areas if there is a diagonal edge between two nodes

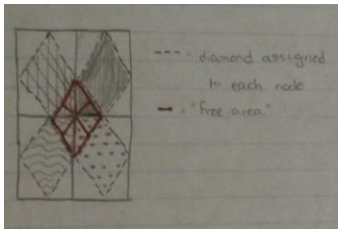

Figure 6: Assignation of eight areas if four surrounding nodes each have different colors

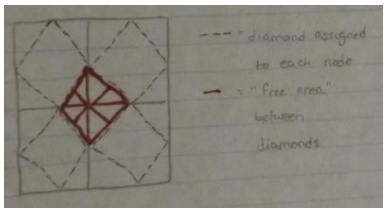

Figure 7: Diamonds around each node and the eight resulting free areas

diamond shaped region around it. This left diamond shaped regions of free area in-between the regions assigned to the nodes (Figure 8). These free areas were then divided equally into eight smaller areas, that were assigned to one of the four surrounding nodes, based on the placement of edges in the similarity graph. For example, if there were no edges between the nodes surrounding a free area, then each of the nodes got assigned the two closest smaller areas of the eight (Figure 7). If there was an edge diagonally connecting two edges, then the two nodes that were connected would each get four of the eight smaller areas (Figure 6).

In our implementation, a Voronoi region was represented as a vector of points: the four points of the diamond around a node as well as points that define what part of a free region that a node has been assigned.

After implementing the Voronoi diagram, we simplified it. Kopf and Lischinski mention collapsing all nodes that have a valence of two. We went through all the points of each Voronoi region and if a point was only shared between two Voronoi regions then it was considered as having a valence of two and was removed (Figure 9).

3.3 Shapes

After we made the simplified Voronoi Diagram, we use it to make shapes. This was where our implementation differed from the original Kopf and Lischinski method. What they do is make the B-splines by separating the different similarity regions first and then make any adjustments they need to make on them to smooth them out. Finally, they color them by placing color at the center of each of the cells and diffusing it out until it hits the B-splines. We instead collected similar voronoi cells into shapes for our next step.

The reason for this was that by collecting them into shapes, we could write closed paths into the SVG file we output at the end. This meant that an artist can go into our output SVG and adjust the shape manually, giving them more control. We were not sure that they could do the same with the original method. However, we did lose the nice gradient shading look from the original implementation by doing this, as our method fills the whole path with one color.

In order to collect our shapes, we created a vector of vectors of Voronoi regions, where each inner vector represents a shape with all the regions contained in it. In order to collect the shapes, we iterated over all of the regions. For each shape currently in the outer vector, we checked the similarity graph to determine if the current region was touching any of the regions making up that shape. If it was, it was added to the shape. Otherwise, we added a new shape for it. We have realized that this was why something that should be a single shape ends up being divided into multiple smaller ones: if the region connecting the

current region to the rest of the shape was not currently in the shape, it would start its own.

After we collected the shapes, we removed the edges between the regions of the shape. For each edge in each region, we checked the edges of the regions prior to it in the regions making up the shape to see if there was a match, either in the same direction or in the opposite. If there was, then the matching edges were removed from both regions. Only the edges of the regions prior to it in the vector of regions were checked in order to make it a bit more efficient.

The next step was to extract some objects we called ShapePaths from this vector of vectors of regions. ShapePaths contains a color, which itself was stored in r,g,b and represents the fill color, and a vector of vectors of points. Each of the inner vectors represents a subcurve of the shape, as it was possible for some shapes to contain multiple independent curves. These curves were collected by first collecting all of the remaining edges of the Voronoi regions making up a shape. Then, while there were still remaining edges, we took the first edge, stored its startpoint as the original point, put its startpoint into a vector storing the current curve, and deleted this edge from the remaining edges. We then took the endpoint of this edge and while the endpoint of the current edge did not equal the original point we continued collecting edges in this manner. When the endpoint of an edge was the same as the original point, we put this curve into ShapePaths as a subcurve. This was repeated until there were no edges left.

This gave us a vector of several unconnected curves that make up a figure and we had to make sure that images with holes in them were drawn correctly as an SVG. In order to do this, while there was more than one subcurve, we took the last subcurve and for each of its points, we calculated its distance to every point in all of the other subcurves, in order to find the pair with the shortest distance apart. We then cut the shape along those points, by inserting the last subcurve (reordered to have its shortest distance point at the front) into the other subcurve after the other shortest distance point, followed by the point from the last subcurve and the point from the curve we inserted into. The last subcurve was popped off, and the process was repeated until there was one curve.

3.4 Output

We found an line svg library to help us output our images [5] to an svg file. It has predefined classes for lines and polygons and will fill in polygons with a specified color. We used this library to produce our final output as well as our similarity graphs and Voronoi diagrams. For the similarity graph, we took the start and end points of an edge and created a Line object with them, which was then passed to a Document object via its operator<<. For the Voronoi diagrams and the final output we defined a polygon object with a color and stroke. The polygon class in the library has an overloaded operator<< that we used to pass it the points of each Voronoi region or ShapePaths object. The polygon was added to a Document object in the same manner that a line was.

In order to create a reasonably large output image that we could see clearly without having to scale up, we multiplied the points by the ratio of the original document size to the output document size, using the

larger dimension in the case of the original images that were not square.

4 Results

In most cases, our program was able to produce a reasonable svg image. We started with simple images, such as a circle and a square with a black outline to them. The square was the most basic case, just for testing if the basic algorithm was implemented correctly and dissimilar colors were disconnected (Figure 10). The circle tested the curve and the sparse pixels heuristics. In both these cases the output looked reasonably like the expected shapes. Since we did not implement smoothing of lines, the outlines of the shapes look rather bumpy. We also tested a couple of basic shapes together in an image, putting a cross inside of a circle (Figure 11).

The next set of tests we did was with a set of invader pictures that we found here [6]. These tested all three heuristics, and some of them were not square images. We tested the first, third, and fifth invader to get a spread of different invaders. The third invader's similarity graph, Voronoi diagram, and shapes output look like what we would expect (Figure 12). The first and fifth invaders both seemed to be missing a piece (Figures 13, 14), probably due to the edges being weighed incorrectly in either the curve or the sparse pixels heuristics.

Boo from from Super Mario was our next test case since it was also used in the paper. We were able to produce an svg that looked similar to the pixel image, however there was the same problem as the invaders where two edges were being weighed incorrectly, resulting in the wrong edge being removed (Figure 15).

The final set of test images were the most complex: game characters. We tested Yoshi and Mario from Super Mario. These images were both larger than the other test images and were also more complex, having several colors and shapes in them. In both cases the output produced from the shapes actually looked like the game characters (Figures 16, 17, 18), if a bit jittery. There was also a transparent space on the right side of the images that was due to our output dimensions being square, whereas the original input images were rectangular. There was a noticeable pause of about 4-5 seconds when removing edges that are between ShapePaths of similar colors. However, this was expected since efficiency was not our first concern.

The svg images were also editable in a vector drawing software (Figure 19). Though Yoshi looked a bit jagged, when modified in Inkscape it only took one step to fix it: selecting all the points of the image and using the smooth point option.

5 Challenges

One of the challenges we had was understanding the Voronoi diagram - what it was, how it was different from a normal Voronoi diagram, and how to implement it. Kopf and Lischinski provide a brief but inadequate description of the Voronoi diagram, so we looked mainly to [7] to get an idea of how the Voronoi diagram would be implemented. Kopf and Lischinski mention that they were able to create just the simplified diagram but did not fully explain how, and it did not seem like other implementations we looked at [6], [7] understood how to skip straight to the simplified diagram either.

Implementing the island and sparse pixels heuristics also presented some challenges. Part of our problem with implementing the island heuristic was correlating the algorithm from the paper with our implementation, which was a little strange because each node only stored four edges, instead of all eight that could possibly lead to it. So in the beginning there was a little trouble with Yoshi due to incorrectly checking islands (figure #). There was a bit of misunderstanding with sparse pixels, where we thought that only one of the edges of a cross would be weighed but after looking at [7] it turns out that both edges of the cross should be weighed. This fixed a small error in the third invader where the right antenna was disconnected from the body.

6 Conclusion

While we did not ultimately end up completing the algorithm we wanted to implement, in the end we were able to get an output that would be helpful to an artist. Due to our implementation choices our result is an SVG with shapes that could be manually adjusted by an artist.

The svg images that are created by our program can be improved with manual adjustment in Inkscape. Yoshi took less than a minute to edit because we simply smoothed out the points in the image . Boo took about a half hour to edit because we smoothed out the shapes as well as deleted extraneous points along the curve and made slight edits to the placement of some of the points. These problems could be solved by implementing quadratic Bezier curves, rather than turning the ShapePaths directly into svg images.

7 Future Work

If we continued this project, along with general bug fixing, we would like to work on implement the quadratic B-splines and their conversion to Bezier curves. While we know that we would have to use De Boor's algorithm in order to change our paths into B-splines, our code isn't currently setup to handle the required knot vector and repeated points in a curve that it would require, though we could do that by having another vector going alongside the path vector store how many times a point is repeated. Then, we would need to use conversion matrices to convert our points into Bezier curves. If we had another week, we could do this, however, due to the challenges we had along the way there wasn't any time. Also, we would like to implement the smoothing to get rid of staircasing on diagonals, which is dependent on the existence of the B-splines. In addition, dealing with sharp corners and places where two edges meet would be a nice addition.

8 Division of Labor

We largely worked in tandem over the course of this project, joining forces for the initial codebase setup for similarity graphs, for drawing debugging diagrams, for debugging our challenges, and for setting up Voronoi regions. In addition, Isabella worked on the curve heuristic and setting up the shapes and ShapePaths parts of the code, while Andrea did the island heuristic, sparse pixel heuristic, and the code for the SVG output as well as testing artist edits to the svgs in Inkscape.

9 References

[1] 2015. Color Conversion. *Equasys*. Web, http://www.equasys.de/colorconversion.html

[2] Cutler, B/ 2015. Homework 9. *Cs.rpi.edu*. Web.

[3] Kopf, J. & Lischinski, D., 2011. Depixeling Pixel Art. *ACM Transactions on Graphics (Procedings of SIGGRAPH 2011),30*, 4, 99:1 - 99:8.

[4] Selinger, P., 2003. Potrace: a polygon-based tracing algorithm. http://potrace.sourceforge.net

[5] Turney, M., 2010. simple-svg. Github repository, https://github.com/adishavit/simple-svg

[6] Vemula, A. & Yeddu, V., 2014. Pixel Art. Github repository, https://github.com/vvanirudh/Pixel-Art

[7] Wala, R.J., 2016. depixelize. Github repository, https://github.com/rjalfa/depixelize

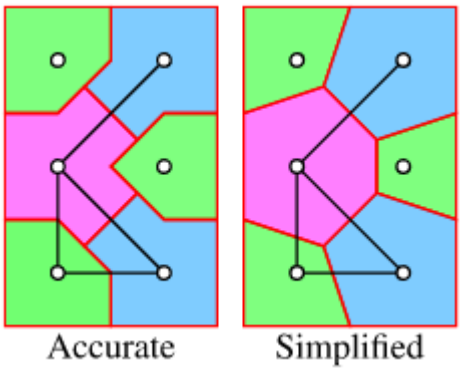

Figure 9: Voroni Diagram. Complex and simplified version from [3]

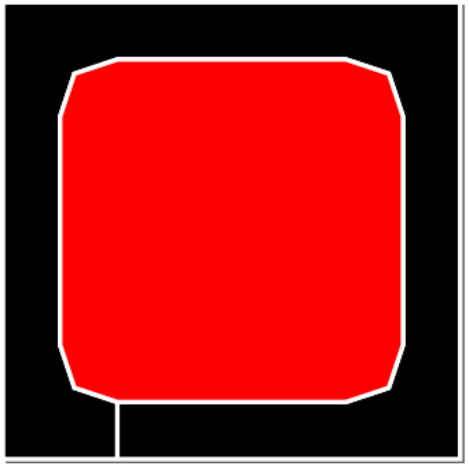

Figure 10: Square output

Figure 11: Cross inside circle

Figure 12: Invader 3

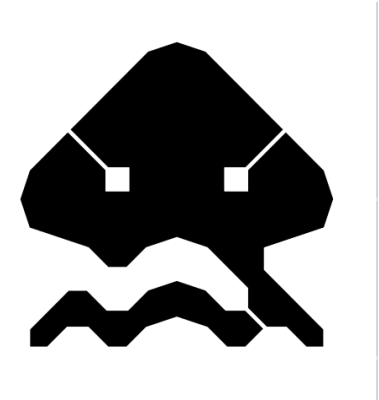

Figure 13: Invader 1

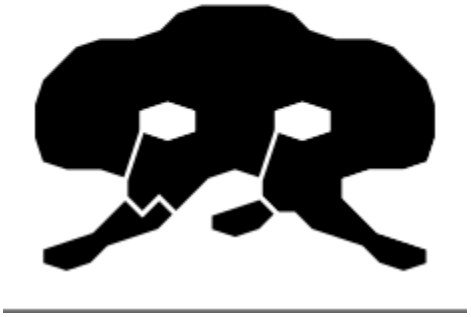

Figure 14: Invader 5

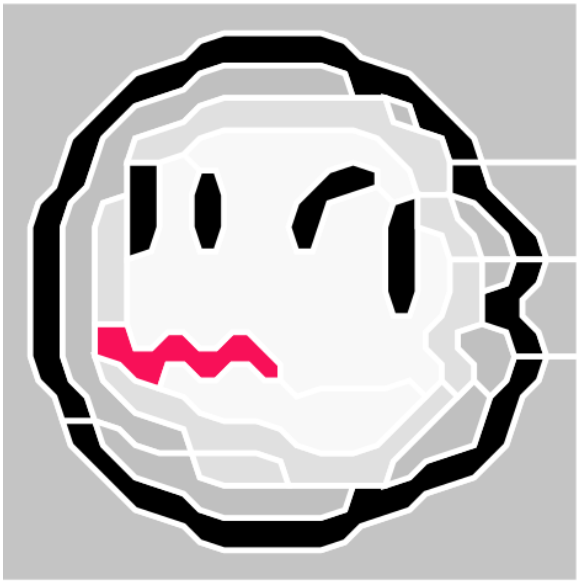

Figure 15: King Boo

Figure 16: Mario

Figure 17: Yoshi with no stroke

Figure 18: Yoshi with stroke

Figure 19: Yoshi edited