Pixel Art Vectorization Report for COMP 4102

Mitchell Blanchard (101000294), Nathan Marshall (101010064), Megan Perera (100998562), and Alexandra Wilson (100998162)

April 15, 2020

Abstract

Pixel art is difficult to upscale using traditional vectorization methods due to its small scale and the fact that each pixel is meaningful, as traditional vectorization methods rely on edge detection that fails in smaller sample sizes. This project seeks to implement and experiment with the algorithm for pixel art vectorization proposed by Kopf and Lischinski in the paper *Depixelizing Pixel Art* (SIGGRAPH 2011). This project will implement their method of vectorizing pixel art, as well as adding to the previous work by addressing issues of anti-aliased pixel art. Anti-aliasing provides a challenge for vectorization because it reduces the meaning of individual pixels, and potentially interferes with the application of the algorithm for creating crisp vectorized images.

Introduction

The issue under consideration is the dynamic upscaling of pixel art images. Pixel art is a traditional graphic style featuring very small images in which most pixels correspond with features and provide meaning. Due to this, pixel art is particularly difficult to upscale with traditional methods, such as nearest-neighbour upscaling, which tends to produce blocky images that reduce meaning and are not particularly useful. The algorithm investigated in this paper provides a way to upscale pixel art images dynamically through vectorization, which allows an image to be rendered at any scale without creating visual artifacts.

The algorithm described by Kopf and Lischinski builds on previous work in the field of dynamic pixel art upscaling, and generally attempts to create a vectorized version of a pixel image by extracting meaningful splines from the art and rendering them **[Kopf & Lischinski, 2011]**. The key feature of this algorithm is its attention to resolving ambiguous connections between pixels that are diagonal to each other. The algorithm begins by creating a connectivity graph for the pixel image, pruning connections en masse followed by a set of heuristics. It then computes a simplified Voronoi diagram of the image at quarter pixel resolution, which helps to reduce blocky appearance in the final render. Koph & Lischinski go on to create splines along edges with significantly different colours, creating a spline graph. They resolve some of these connections in order to simplify the graph, such as by combining similar splines. They smooth the graph in order to reduce stair-casing by reducing the energy of splines, which they calculate in several different ways. After they have selectively smoothed the splines, they render the completed image by creating diffusion points, which could be described as radial gradients that may or may not fade to a different colour, at the centers of pixels, with the smoothed splines serving as boundaries between colours, as described in **[Jescke et al, 2009]**.

This algorithm is important for work in dynamic pixel art upscaling, as it provides a less expensive way to re-render older art styles. It has applications for rendering at higher resolutions, while preserving important details from the original images, and preserving the spirit of the art. It is challenging to implement in that it relies on several computer vision and graphics techniques for image analysis and processing. Many of these techniques were not discussed in the coursework, and leverage both OpenCV techniques and more general image processing techniques to create a polished result. It is important to note that, while we will be implementing the majority of this algorithm, we will not be rendering the final image using the same techniques that the report authors do, as this would substantially add to the scope of the project.

One additional challenge to consider is the application of this algorithm to anti-aliased images. This algorithm assumes that all pixels are meaningful and attempts to preserve them. In anti-aliased images, however, the anti-aliasing would produce unwelcome artifacts, such as strange colours and meaningless lines. Instead of having to manually prune these out of the final vector image, or simply dealing with worse looking vectors, we will be implementing an anti-alias cleanup method to be run prior to the main algorithm. This algorithm will be developed by the team and implement heuristics for colour selection designed to reduce anti-aliasing and produce a final vector image that is extremely similar to one that you would create by using the algorithm on a regular image.

Background

The main algorithm described by this project is a modified implementation of a pixel-art vectorization technique, as implemented in the article *Depixelizing Pixel Art* **[Kopf & Lischinski, 2011]**. Creating an implementation of the algorithm described in this paper is the primary focus, and will take the most time to implement. It provides a structure to approach the issue of rendering pixel art at higher resolutions, and provides us with a guide to determining common heuristics for selecting meaningful connections between pixels. This project also makes use of OpenCV version 3.2 for a variety of image processing and display methods that would be time consuming to implement manually, and is implemented in C++.

We also use several pixel art images for testing taken from classic games, such as *Final Fantasy* and *The Legend of Zelda*.

Approach

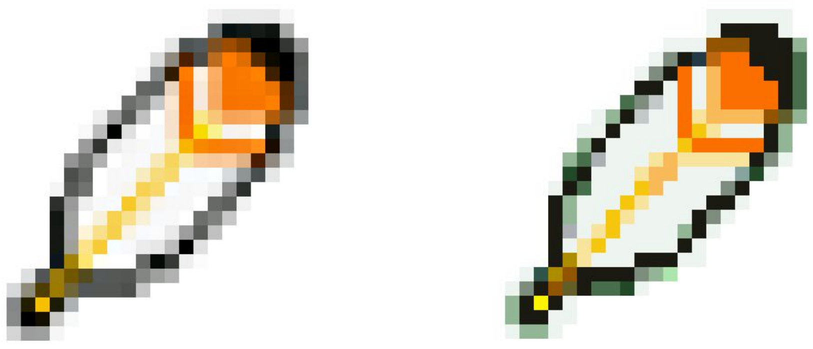
Our approach to implementing this algorithm fell into a few major steps. These steps included creating an algorithm to clean anti-aliased images, generating a connectivity graph for the pixel image, extracting splines from the graph, and rendering. In some cases, we closely followed the method chosen by the report authors, and in other cases, we opted to deviate from their methods in order to achieve results that worked better with the short time frame of the project.

Cleaning Anti-Aliased Images

When an image is anti-aliased, the image looks smoother and its edges are not as sharp. A single colour is split into multiple lighter colours in order to achieve this. Smoother looking transitions between colours are generated through the anti-aliasing process. By looking at an image, one can see how the various colours generated from anti-aliasing an image were reached. This is far harder to tell when looking strictly at colour data. In order to clean up an anti-aliased image to attain a more unified palette, we have constructed the following system.

Since anti-aliasing is the smoothing of edges and the removal of details, we start by sharpening the image. We apply a Gaussian blur and acquire the details of the anti-aliased image removed by the blur. Those details are re-applied to the anti-aliased image, so that the edges blurred by anti-aliasing will be more prominent.

After sharpening the anti-aliased image, the lightest and darkest values within the image will be found. We can be fairly confident that these colours are more representative of the original image's colours rather than anti-alias noise, since the noise that comes from anti-aliasing is muddied and muted.



(a) Feather with Anti-Aliasing (b) Feather after being cleaned

FIGURE 0: Anti-aliased feather before and after cleanup with our algorithm.The white and orange of the feather are more unified, and the black outline is more defined.

With these extremes, now the bulk of the work can begin. The image is to be traversed pixel by pixel. For every pixel in the sprite, we convert its RGB values into HSL values (hue, saturation and luminance/lightness). Hue represents where the colour lands on the colour wheel. Saturation tells us the intensity of the colour, where a high value is a vivid colour, and a low value is a dull and muted colour. The luminance represents how much light the colour has. If a colour's luminance is high, it is lighter and if it is lower, it is darker. Having these values for each colour will allow us to infer useful information about the relationship between the colours, and whether they should be considered the same.

It should be noted that the specifications for our checks were made through rigorous testing and investigation. It is impossible to decisively say which colours should or should not go together for all possible cases, since every piece of art has differently defined palettes and colour usage. Considering this, we reached distinctive rules that we felt generated the best overall results through thorough testing.

For every pixel, we will compare it with all our colour groups to see where it belongs. There are three main cases we look for: can this colour be considered black, can this colour be considered white, or are these colours similar enough?

A colour can be considered black if the sum of its RGB values is less than 76, its luminance is less than 46, or if it has both a luminance less than 81 and a saturation less than 61. Colours that meet these criteria are dark enough that we can call them black, and merge them together. A colour can be considered white if the sum of its RGB values is greater than

709, if its luminance is greater than 214, or if it has both a luminance greater than 194 and saturation less than 51. Colours that fit this criteria are light enough that they can all be placed into the same category.

If we are evaluating a colour that is neither black or white, the comparison is more complicated. Using hue, saturation and luminance, we will make checks to determine whether or not these colours can be considered close enough to combine.

If colours can be considered close by their hue, this means that they are near each other on the colour wheel. We must check to see that their saturation and luminance don't make them look too different. If colours can be considered close by saturation, they are similar in colour intensity. We must make sure that they are similar enough colours with similar enough brightness. Knowing the brightnesses of colours alone is not enough information to know if two colours are similar, so we also must look at their saturation and hues to see that they are close. If we perform all these checks on a single pixel, and find there are no colours we have which are similar to it, we create a new colour bucket to place this new unique colour in.

After we evaluate all of the colours in the image and have a collection of unique colours, we perform a final sweep of our selected averaged colours, to see if any of them are close enough to be combined.

Finally, we have a condensed range of colours made from filtering through the various anti-aliased colours. With this palette, we traverse the anti-aliased image once more. For every pixel we encounter, we find the corresponding colour from our palette that should be used in its place, and paint over it with that new colour. Once we have done this for every pixel in the sprite, we have an image that has been cleared of most of the anti-aliasing artifacts. This image is now ready to be vectorized.

Generating and Refining the Connectivity Graph

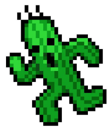
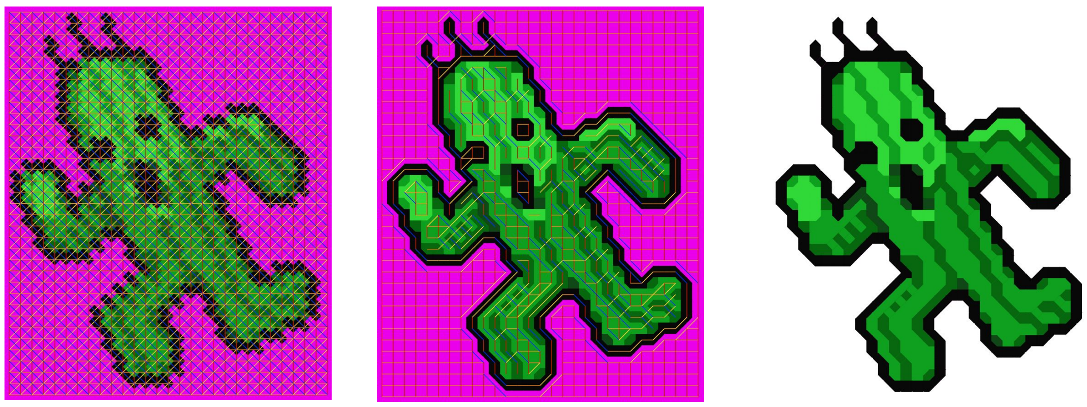


Figure 1: The initial pixelated image.

The first step in vectorizing the pixel images, such as the one in **Fig. 1** was generating a connectivity graph, as depicted in **Fig 2(a)**. In our implementation, the centre of each pixel becomes a node. A pixel may be connected to any of its immediate neighbours diagonally or in the cardinal directions. To begin, each pixel is connected to all of its neighbours. As specified in the original algorithm, we want to minimize these connections to make finding splines and polygons within the image simpler. We begin with pruning. By the end of this step, the goal is to have no connections between different coloured pixels and to have as few diagonal connections as possible.

Pruning

The pruning algorithm is run at a general level to resolve as many incorrect connections as possible. For each pixel, we check the colours of its neighbours in two different ways. Firstly, if a pixel and its neighbour are different colours, we disconnect them. Second, we want to destroy diagonal lines between large areas of the same colour of pixels. For example, if our central pixel is white and the pixels to the top, top-left, and left of it are also white, then we disconnect the top-left and central pixels in order to minimize connections. We do this for each diagonal pixel, reducing the connectivity graph as shown in **Fig 2(b)** and **(c)**.

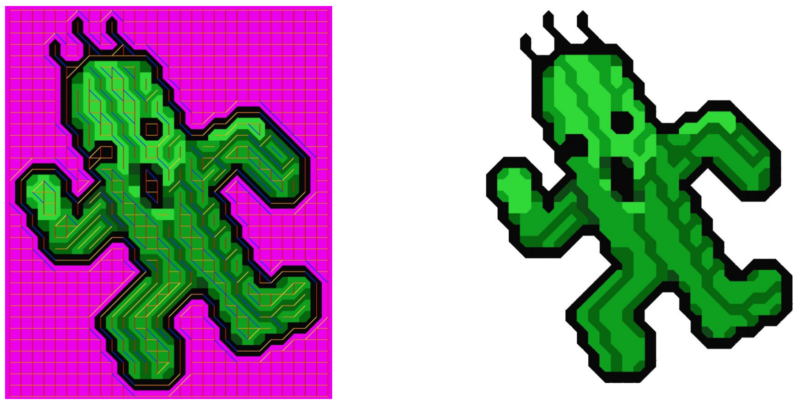


(a) Initial connectivity graph (b) After pruning (c) After pruning (no graph)

Figure 2: (a) The initial connectivity graph which is created as the first step of the algorithm. At this stage, every pixel is connected to each of its neighbours. (b) The connectivity graph after the initial pruning step. The majority of connections are removed during this pruning step. (c) The image as it appears after the pruning step, without the overlaid connectivity graph.

Heuristics

After the initial pruning step is complete, we are left with a number of 2 x 2 regions in the graph with crossing diagonal connections. There are three major heuristics that are used to help select diagonals. The goal of running these heuristics is to select a “most correct” diagonal, so that there are no “x” shaped connections remaining between pixels. The three heuristics that we used are described as the curves heuristic, the sparse pixels heuristic, and the island heuristic **[Kopf & Lischinski, 2011]**. The output of these heuristics form a weighting algorithm for connection selection, in which we sum the weights gathered from these heuristics for each connection, and keep the connection with the largest weight. The other connection is removed. The final output of this step is shown in **Fig. 3**, and the results of the application of individual heuristics are shown in subsequent images.



(a) Final heuristics result (b) Final heuristics result (no graph)

FIGURE 3: (a) The image after the weights from all three heuristics have been combined. The diagonal edges with the higher weights are kept, while the lower weighted connections are removed. (b) The image after all heuristics, without the overlaid graph.

The curves heuristic selectively maintains long curves within a pixel image. For the purpose of this report, curves are defined as lines of valence-2 nodes, ending with a valence-1 node **[Kopf & Lischinski 2011]**. Here, the valence of nodes refers to the number of connections between them. A node is not considered to be part of a curve unless it has two or fewer connections. We iterate over nodes until we find the end of the curves, stopping when we either reach the beginning of the curve (as would be the case in a loop), or when we reach the end. The curve lengths for each diagonal are compared, and we select the longest curve, as shown in **Fig. 4**.

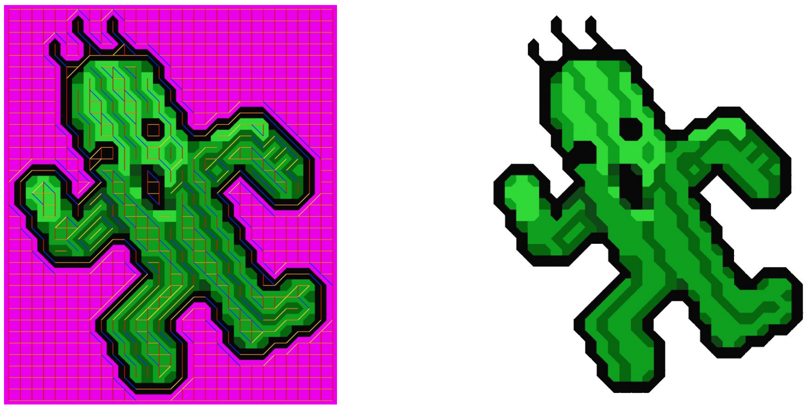

(a) Image after curve heuristic (b) Image after curve heuristic (no graph)

FIGURE 4: (a) The image and the overlaid connectivity graph showing which connections the curve heuristic would vote to keep. (b) Image after curve heuristic, without the overlaid connectivity graph.

The island heuristic opts to maintain connections that, if lost, would leave individual pixels with no connections. In these cases, we compare the connectivity of the pixels surrounding the diagonals. If one of these nodes only has one connection, i.e., to the other side of the diagonal connection, then we opt to keep that connection, as in **Fig. 5**.

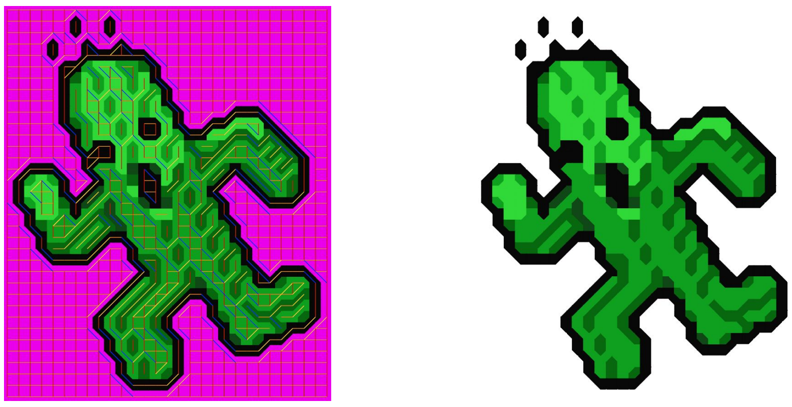
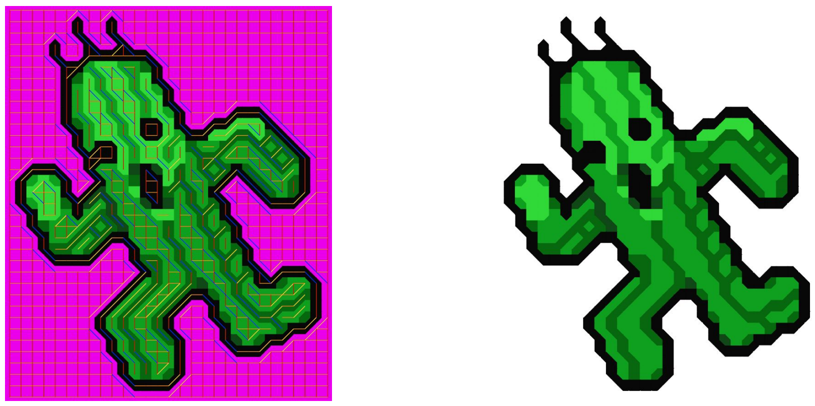

(a) Image after the island heuristic (b) Image after island heuristic (no graph)

FIGURE 5: (a) The image and overlaid connectivity graph showing which connections the island heuristic would vote to keep. (b) Image after island heuristic, without the overlaid connectivity graph.

The sparse pixels heuristic opts to maintain connections between foreground pixels as opposed to background pixels. Within an 8 by 8 pixel neighbourhood, we find all pixels connected to the top-left to bottom-right connection, and separately, all pixels connected to the top-right to bottom-left connection. We count these connections separately by iterating over the neighbourhood and counting connections. After we have counted these connections, we compare them and opt to keep the diagonal connection that has fewer additional pixel connections in the immediate neighbourhood. This heuristic works by assuming that background colours will be more numerous and less important to the final vectorized image. An example of the application of the sparse pixels heuristic can be seen in **Fig. 6**.



(a) Image after the sparse pixel heuristic (b) Image after sparse pixel heuristic (no graph)

FIGURE 6: (a) The image and overlaid connectivity graph showing which connections the sparse pixel heuristic would vote to keep. (b) Image after sparse pixel heuristic, without the overlaid connectivity graph.

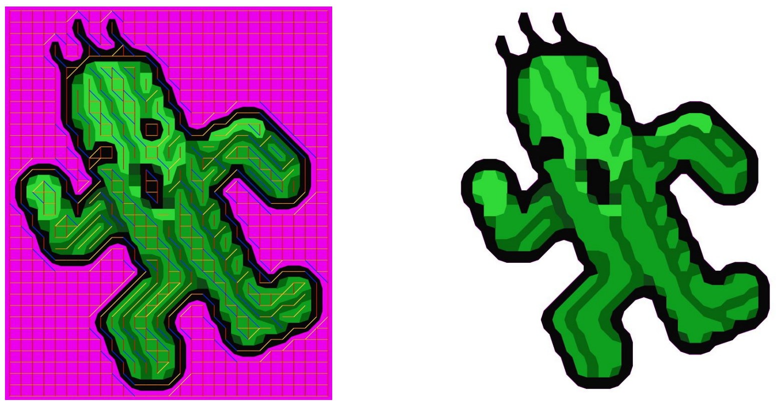
Spline Extraction

Once we created and resolved connections within the connectivity graph, we had to extract B-spline curves from the graph, in order to render paths instead of individual pixels in the SVG output image. At the highest level, our method for extracting these splines involved locating polygons within the connectivity graph based on the edges of individual pixels, and smoothing corners in order to create a less stair-cased output spline.

The first step is to create a lattice grid of intersections between pixels with a width and height equal to the width and height of the image in pixels plus one, respectively. We iterate over these grid intersections and begin to build our contour maps. These contour maps indicate methods of moving between grid intersections. For a given grid intersection, if the top-left pixel is not connected to the other pixels surrounding the intersection, then it is considered a diagonal. If the top-left pixel is diagonal, we then check if the top-right and bottom-left pixels surrounding the grid intersection are connected. If they are, then we adjust the position of the grid intersection point to create more of a diagonal line and therefore reduce stair-casing. We also specify that a bidirectional contour exists between the grid intersections above and to the left of the current intersection. If the diagonal in the top left exists but the top right and bottom left pixels are not connected, then we do not move the corner point, and only specify a contour from the top to the left. We continue to build out the contour map for this intersection by performing similar checks for diagonals in the top-right, bottom-right, and bottom-left corners.

After we have created a full contour map, we must find all of the polygons that exist within the contour graph. For each grid intersection, we must iterate over any contour maps at that intersection. While the contour map at a given intersection is not empty, we create polygon for each contour through the intersection based on the connections between intersections. In each polygon, we consider both the current intersection and the next intersection in our contour. We determine the colour of the polygon based on the direction of the connection. We walk to successive neighbours in this contour, erasing the previous connection, and tracking corner data. We also keep track of the general direction that we have travelled in through an integer that starts at 0, and is incremented each time we travel clockwise. This integer is decremented each time we turn counter clockwise. We traverse this contour until we arrive back at the starting intersection. If we have travelled more in the clockwise direction than the counterclockwise direction (making at least 4 turns clockwise), then we have found a polygon worth keeping. In order to avoid confusion, duplicate polygons, and extra lines, we only keep "clockwise" polygons.

Now that we have polygons with a general shape, we need to create control anchor points for each point in the spline. In order to create these points, we iterate over our list of anchor points in the polygon, taking into account the last and next points in the polygon. We find the tangent line between the next and previous points, reverse the tangent angle by adding PI, smooth the tangent by multiplying it by a constant small factor, and create two new control anchors. We calculate the positions of these new anchor points to be on opposite sides of the circle with a radius of the smoothed tangent length and the tangent angle and opposite tangent angle. These are converted from polar coordinates to Cartesian coordinates and added to the list of control points. **Fig. 7** shows the results of the spline extraction.



(a) Image after polygons are extracted from connected segments

(b) Image after polygon extraction (no graph)

FIGURE 7: (a) Polygons are extracted from the remaining connected segments in the graph, and edges are determined based on color differences. The image shown here is after polygons are extracted. (b) Image after polygons are extracted, without the overlaid connectivity graph.

Rendering

The final step of our process is to convert the polygons into a renderable SVG. Each spline that we have extracted is converted into an SVG path which is composed of a starting point and a series of Bezier curves based on its control anchor data, ending back at the start location. We fill this path with the colour specified for the polygon. These paths are compiled together into an SVG file and saved to disk.

Differences Between Algorithms

There are several differences between the algorithm written by the original authors and the algorithm that we have implemented, particularly with respect to spline extraction, rendering, and anti-aliasing. Since anti-aliasing control was not present in the original paper, we discuss it in its own section. The differences in spline selection are significantly different in the approach that we took and the approach described by the original authors.

One deviation from the original methods developed by the report authors was the different implementations of voronoi diagrams. In the paper by Kopf and Lischinski, a simplified Voronoi diagram is generated at quarter-pixel resolution and is then used as a basis for extracting splines and performing corner detection. We instead integrated this step with our spline selection, creating a simplified diagram while creating our polygons that, had it been rendered in subpixel resolution, would have created a diagram very similar to the one created by the report authors. We experimented with different Voronoi diagram implementations, creating two separate implementations, before switching to our chosen method of creating a simplified version that relied on anchor point positions instead of sub-pixel corner reduction.

Our spline selection algorithm is based on shape selection, as opposed to simply creating partitions between colours. In the original paper, splines are created along contour lines in which there is a significant difference between pixel colours, based on a simplified voronoi diagram **[Kopf & Lischinski, 2011]**. In the original algorithm, these splines are simplified combining splines that form straighter lines and that exist along stronger contour lines. These splines are also manipulated to minimize staircasing and ignore sharp corners [**Kopf & Lischinski, 2011]**. This difference contributes significantly to the differences in rendering. The method of rendering used in the original algorithm makes use of diffusion points and spline boundaries **[Jescke et al, 2009]**. In this method, colours are specified and boxed in by splines, without ever having to draw additional paths **[Jescke et al, 2009]**.

This is in stark contrast to our own method of rendering. We chose to render splines as paths and fill them, creating a different style and requiring different components. Since our splines represent polygons, it was unnecessary for us to perform the same spline control point resolutions at spline junctions and spline simplifications performed in the original algorithm.

It was also simpler for us to compute control points and manage staircasing for our method. This difference in rendering method was chosen for several reasons, including time constraints and focus. Since we wanted to spend the most time on the actual image manipulation and vectorization, we did not want to spend extra time implementing and debugging a complex rendering method.

Results

In this section we have provided a variety of results using different pixel art images as input. The first section of results was generated using the original pixel art, and shows a comparison of the image before and after being run through our algorithm. The second section of our results was generated from anti-aliased versions of pixel art images, and shows the initial anti-aliased image, the cleaned version of that image, and the final SVG result after running through our algorithm. Our first example also provides sample images which show the different stages of our algorithm, and gives a better idea of how each individual step is modifying the pixel image.

While we did not produce exactly the same results as those created by Kopf & Lischinski, we successfully applied heuristics for diagonal-connection selection and extracted splines from the pixel images, allowing us to render the pixel art smoothly at a variety of scales.

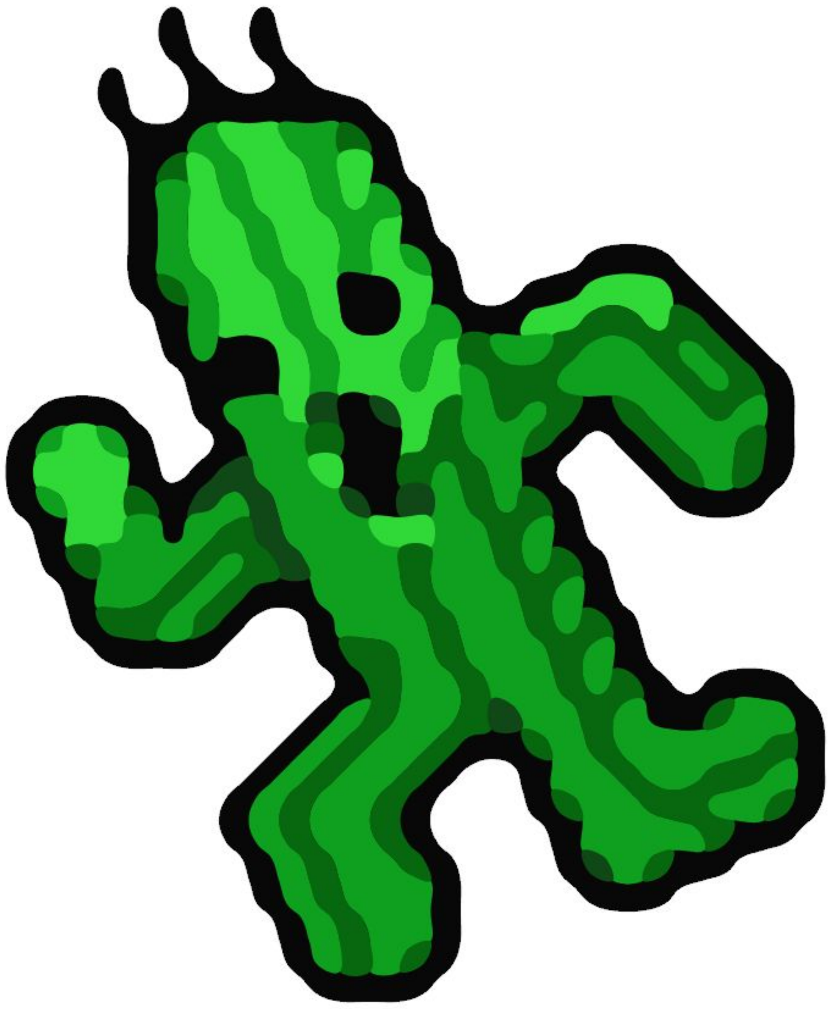
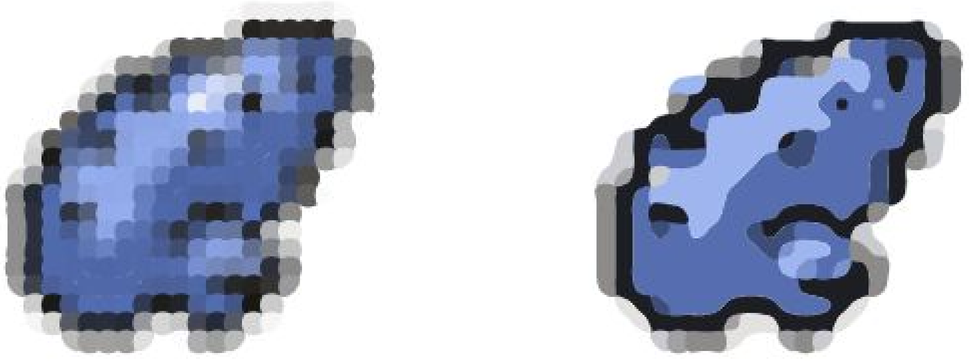


FIGURE 8: The final result after splines are extracted from polygon contours, which form the new shapes for the final image.

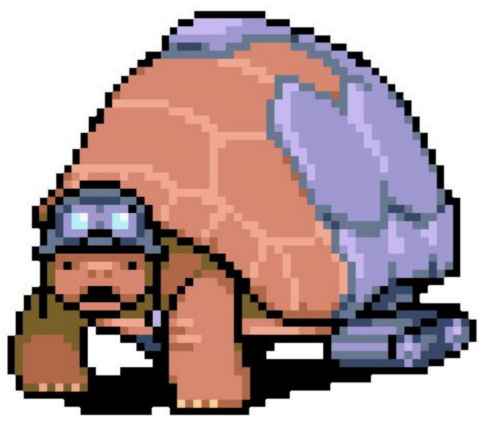
By comparing the vectorized results between an anti-alias image and an anti-alias image we have cleaned, it becomes apparent the importance this step has in producing crisper and more coherent results, as seen in **Fig 9.**

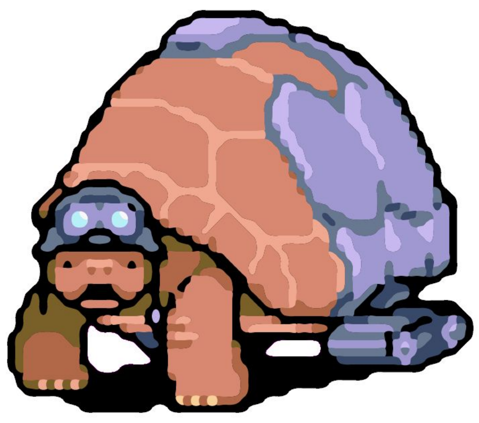


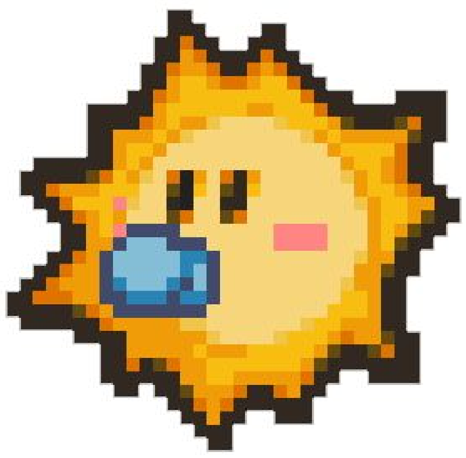
(a) Vector result of anti-alias image (b) Vector result of cleaned anti-alias image

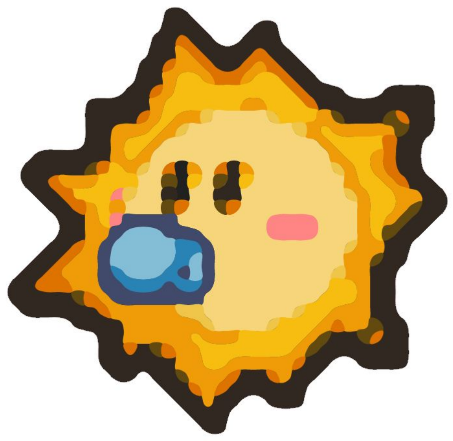
FIGURE 9: (a) Image of the vectorized result of an anti-alias image. Simply vectorizing an anti-aliased image does not provide good results, as anti-aliasing leads to a large amount of colours that are only slightly different from one another. (b) Image of the vectorized result from the cleaned anti-alias image. With unified colours, the result is more solid and clear.

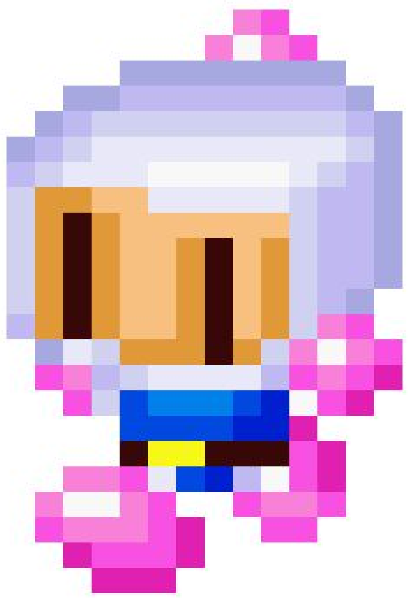
Other results (Showing before and after our algorithm is applied).

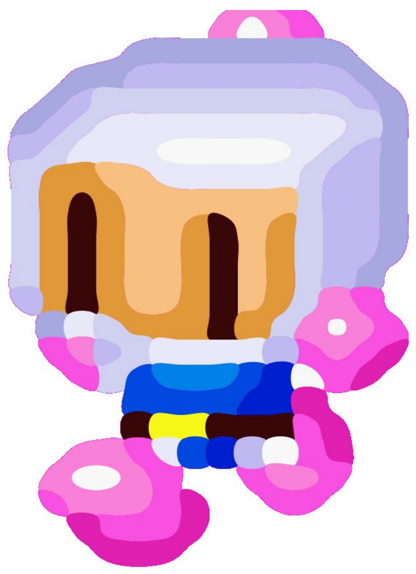




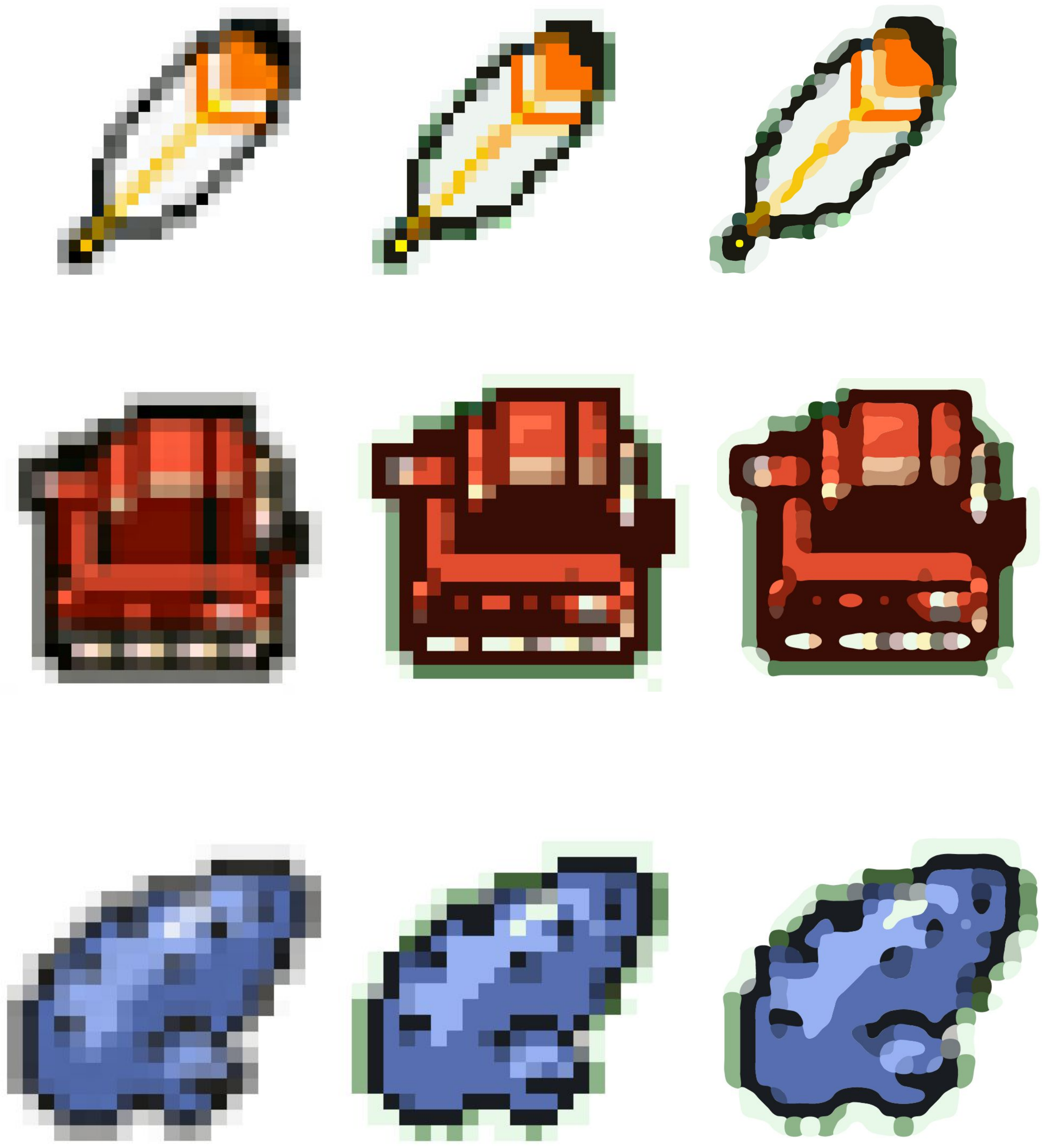








Results using anti-aliased images as input. Images are cleaned by our algorithm, and then the svg is created.



SVGs and further results can be found in the github repository

List of Work

Equal work was performed by all team members.

GitHub Page

Github Project: https://github.com/aqwilson/pixel-vectorization-4102

References

Jescke, S., Cline, D., and Wonka, P. (2009) A GPU Laplacian solver for diffusion curves and Poisson image editing, ACM Transactions on Graphics, Volume 28, Issue 5, pp.1 - 8

Kopf, J., and Lischinski, D. (2011) Depixelizing Pixel Art, ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011), Volume 30, Number 4, pp. 99-1 – 99-8.

Image References

Bomberman: *Bomberman Max 2: Blue Advance / Red Advance*
Cactuar: *Mario Hoops 3-on-3*
Feather: *Super Mario World*
Mecha-Turtle: *Mother 3*
Ocarina: *The Legend of Zelda: A Link to the Past*
Seel: *Pokemon FireRed / LeafGreen*
Sprout: *Mario & Luigi: Superstar Saga*
Tank: *Super Famicom Wars*
Watt: *Paper Mario*