Raster to Vector Conversion in Pixel Art Images

Team 02

SHREYAS GUPTA

1 INTRODUCTION

Many arcade machines and mainstream video game consoles manufactured from the 1980s until the late 2000s had highly constrained memory and computational speed, due to which it was infeasible to store and render detailed images while ensuring smooth gameplay experience. Game designers were, therefore, incentivised to design scenes and characters with a highly constrained set of pixels and colours, which is what led to the origin of pixel art.

While modern games are able to render high-quality images, they often lack the aesthetic of pixel art from the earlier games, that many players might prefer. Using some cleverly designed algorithms, we can convert pixel art to a vectorised image, which can be rendered at an arbitrarily high resolution. If done right, this can greatly improve the graphics of a game while preserving its aesthetics and details.

While numerous algorithms exist today to convert a pixelated image to a higher resolution one, their goal is generally to do so with 'realistic' images. In contrast, pixel art can often have important details stored in very few pixels (for instance, the eye of the character is often a single pixel!) that can be lost with such algorithms. This project aims to implement such an algorithm designed specifically to vectorise pixel art raster images.

2 LITERATURE REVIEW

This project will be closely referencing the paper 'Depixelizing Pixel Art' by Johannes Kopf and Dani Lischinski [1]. In it, the authors compare the images obtained by some existing algorithms for depixelizing pixel art, and show their shortcomings. For instance, Adobe Live Trace vectorisation technique can cause loss of detail in pixel art, which algorithms like Photozoom 4 and hq4x can leave 'staircasing' artifacts in their final result. The algorithm proposed in the paper aims to preserve detail from pixel art, while eliminating staircasing for a smoother final result.

2.1 Planar Graph Representation of Similar Adjacent Pixels

We first need to identify connected segments of the image. Two pixels are likely to be 'connected' if they have similar colours and share either an edge or a corner. Note that in a connection at a corner, upon zooming in, the pixels can always appear disconnected. Thus, it is important to ensure that these pixels appear connected during vectorisation.

However, there are multiple ways to decide the vertices that should be connected, and not all of them will lead to a good final result. How do we decide which pixels need to be connected in the final image, and which are not? We create a graph G where the vertices correspond to the raster pixels, and the edges are drawn between adjacent pixels which need to be connected in the final image.

We initialise graph $G = (V, E)$ to the following:

- V = set of all pixels in the raster image.
- $E = \{(u, v) | u, v \in V$ and pixels u, v share a pixel edge or pixel vertex $\}$

Author's address: Shreyas Gupta, shreyas20131@iiitd.ac.in.

Note that in order to form meaningful connections between the pixels in the final image, **the similarity graph must be planar**. We iteratively remove edges from G until we have a planar graph.

The first step in edge removal is the simplest; simply remove edges between pixels having 'dissimilar' colour values. We can check for similarity by comparing the euclidean distance between the colours in the **YUV colour space**.

Next, we resolve all overlapping edges. Note that overlaps only occur between diagonal edges in any 2×2 grid in the raster. This can be of the following 2 types:

(1) Suppose in a 2×2 grid, each pixel shares an edge with every other. This means each pixel is similar to each other, and we can remove both the overlapping diagonals.
(2) Else, the 2×2 grid contains only the 2 overlapping edges, and either diagonal can be connected, but for best results, we select the diagonal that preserves curves (is part of a sequence of valence 2 vertices), contains **sparse pixels** (if some pixles are sparse, they likely have important detail), or does not lead to the formation of **islands** (a segment containing just one pixel)

Performing this for all 2×2 grids gives us a similarity graph G that is planar. We can use this to decide the segments the final image must be partitioned in

2.2 Creating a Voronoi Diagram from the Similarity Graph

After we obtain the similarity graph G above, we use it to derive a Voronoi diagram from the image. This is obtained as follows: Divide the graph into 'segments', by cutting each edge in half. FOr every point in the image, it is placed in a group corresponding to the closest segment in this image.

The above gives us the 'accurate' Voronoi diagram of the graph. This consists of somne vertices having a valence of 2, and some having a valence of 3. We obtained a 'simplified' Voronoi diagram by removing all vertices of valence 2, and connecting their adjacent vertices instead.

This simplified Voronoi diagram gives a good representation of the image segments that will be connected. However, this image is not 'smooth'. We do this by adding quadratic spline curves between visible edges of the edges.

2.3 Adding Spline Curves to Smoothen the Vectorised Image

Note that some edges of the edges are 'visible', i.e., they separate two segments having dissimilar colours. We are only interested in drawing spline curves around these edges.

In the visible edges, we can sometimes have 3-way junctions, that is, 3 edges meeting at a common vertex. We need to choose to connect 2 of them.

If 2 of the edges are contour edges (separating segments with very dissimilar colours) and 1 is a shading edge (separating segments with similar edges), we connect the contour edges. Else, we connect any of the two edges having angle closest to $180 degrees$.

Following this, we add spline curves to the visible edges.

2.4 Optimising the Spline Curves

While the spline curves obtained above are smooth, they are likely to have staircasing artifacts. The final step before rendering the image is to optimise the curves to eliminate such artifacts.

For the i-th node, we defien its **energy** as follows:

$$E^{(i)} = E_s^{(i)} + E_p^{(i)}$$

where

$$E_s^{(i)} = \int_{s \in r(i)} |\chi(s)| ds$$
$$E_p^{(i)} = ||p_i - \hat{p}_i||^4$$

We want to minimise the total energy of all the nodes. $E_s^{(i)}$ is minimised with decrease in curvature of the curve at that point, and $E_p^{(i)}$ is minimised with how close the displaced point is to the original point. Minimising $E^{(i)}$ we are smoothening the curves while ensuring the curves are not too different from the original image, so as to preserve the shape.

2.5 Rendering

Once we have the spline curves from the above algorithm, we can use these to render the image using standard rendering techniques provided by OpenGL.

For splines having very dissimilar colours on both sides, it is preferable to have sharp colour contrast between them. If the colour is slightly similar, we can choose to diffuse the colours together for a smoother textured appearance.

3 MILESTONES

Based on the literature review above, the following project milestones have been identified.

S. No.	Milestone	Member	Status
	Mid evaluation		
1	Initialise similarity graph for raster image pixels	Shreyas	✓
2	Reduce similarity graph to optimal planar graph	Shreyas	✓
3	Obtain actual Voronoi diagram for similarity graph	Shreyas	✓
4	Simplify Voronoi Diagram and Render Simplified Diagram	Shreyas	✓
	Final evaluation		
5	Resolve T-junctions in Simplified Voronoi Diagram	Shreyas	✓
6	Add quadratic spline curves to visible edges	Shreyas	✓
7	Optimise spline curves to add smoothness	Shreyas	✓
8	Render vectorised image from smoothened spline curves	Shreyas	✓

4 APPROACH

4.1 Overview

In order to depixelise a given raster image, the program first needs an input of an image. This is done using the stb_image library, to read a PNG image and translate it into colour coordinate values for each pixel.

Notice that in case of pixel art, some pixels may not share an edge For instance, the mouth of a character is often composed of diagonally connected pixels. How do we know which such pixels need to be connected in the final image? We construct a **similarity graph**, a graph where the vertex set is the set of pixels, and edges between 2 pixels represents that the 2 pixels need to be connected in the final image.

We can infer that the similarity graph **must be planar**, as any edge overlap makes it impossible to connect both pairs of pixels. To make this similarity graph, we take an edge between adjacent similar-coloured pixels, and iteratively eliminate edges until the graph becomes planar.

Once we have a similarity graph, we draw a **Voronoi Diagram** of the raster image. A Voronoi diagram divides the plane of the graph into segments, where each segment has points closest to itself as well as its half edge. This diagram shows that any 2 pixels adjaent in the similarity graph would appear connected in the Voronoi diagram.

Finally, we obtain a **simplified Voronoi diagram** by deleting all intersections of 2 edges. These connections will be unnecessary when we interpolate the intersections with spline curves. The simplified Voronoi diagram has a "smoother" appearance that closely resembles the expected final result.

4.2 Constructing a Similarity Graph

After reading the input image, we have the pixel art image stored in an $m \times n$ 2D array, where m and n are the height and width of the image respectively. The similarity graph is a graph that has a vertex for each pixel, and edges that may be connected to neighboring pixels only. Since the maximum degree of a vertex is 8, it is best to store the graph as an adjacency matrix of $mn \times 8$ size. The row represents the corresponding pixel and the column number represents which neighboring pixel may be connected to it (top-left, top, top-right, etc.). This allows us to modify, search, and store the edges using linear time and space.

When initializing the graph, we set all the edges as 'true' (that is, we connect an edge between every pair of neighboring pixels), after which we eliminate unneeded vertices. The first step of elimination is to simply remove all edges between pixels with "dissimilar" colors. There are many ways to define dissimilarity between colors. Here, we consider 2 colors as dissimilar if their Euclidean distance in the RBG color space is greater than a threshold value.

The figure above shows an example of this method. After this, the graph is left with some edges overlapping which need to be resolved. These overlappings occur in 2×2 sub-grids of the image and can be of 2 types: Those where the 4 pixels are all similar colors, and those where the 2×2 image creates a "checkerboard" pattern, that is, the diagonal colors are similar but dissimilar to the other diagonal.

The former is easier to resolve; we may simply remove both overlapping edges. This works since the horizontal and vertical edges are connected in such a case, and will thus appear connected in the final image.

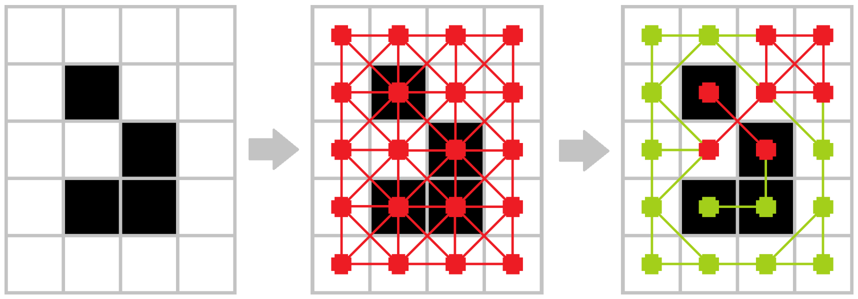
Fig. 1. Demonstration of Similarity graph. (a) Original raster input. (b) Initialisation of Similarity Graph. (c) Elimination of edges between dissimilar pixels.

The latter is tricky to resolve. Removing either of the two edges would create an image that is very different from the other, and it is hard for a computer to decipher the vertices that need to be connected without the use of some computer vision techniques. Here, we use 3 heuristics, in decreasing order of priority, to determine the edge that needs to be eliminated.

(1) Firstly, if either of the edges is part of a chain of vertices of degree 2, that edge is given greater priority. We can check if that is the case by performing a graph traversal starting from the nodes of the diagonal. A series of nodes in a chain is likely to carry important detail such as a character's mouth, clothing, or border.
(2) Next, we check if either of the edge colors is "sparse" compared to the other. This means, in close proximity, is one of the colors predominant over the other? In our code, we check this over a 6×6 area around the diagonal and check if either of the diagonal colors occurs much more frequently than the other. Sparse pixels usually carry important detail that should not be broken.
(3) Lastly, we check if removing an overlapping edge breaks the graph into more connected components. In general, we wish to minimize the number of connected components in our graph, as multiple components may make our image appear broken. In our code, we perform graph traversal (BFS) to determine if an edge can break the image into multiple components.

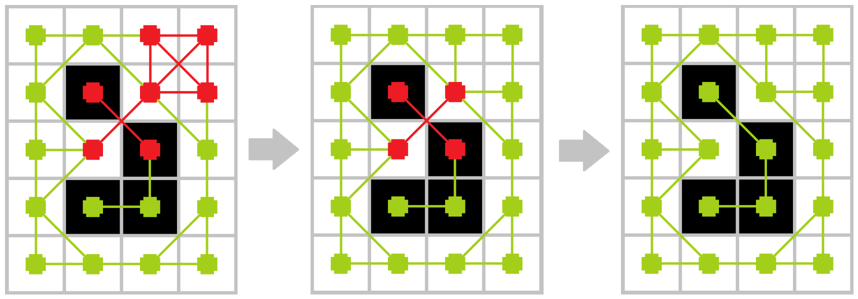
Fig. 2. Making the similarity graph planar. (a) Graph with similar pixels connected. (b) Graph with similar pixel overlaps resolved. (c) Graph with checkerboarding resolved.

Notice that in the above example, the algorithm resolved checkerboarding by connecting the black pixels together and eliminating the edge between the white pixels. This is in accordance with the heuristics defined above, and ensures that the black pixels will all be connected in the final image.

4.3 Constructing a Voronoi Diagram

A Voronoi diagram tells us the parts of the image that will be connected in the final output. The diagram is defined for a planar graph as follows: Divide each edge into 2 halves, and allocate each half to the vertex it is connected to. Then, divide the plane into segments such that each segment consists of just one graph vertex, and all points closest to this vertex.

In our case, we color each segment the same color as the pixel corresponding to the vertex. This way, each segment pair where the vertices are connected by an edge, are connected visually. This is because the 2 segments sharing an edge have the same color, and would appear to be connected in the diagram.

The figure below shows the Voronoi diagram for the figure shown above.

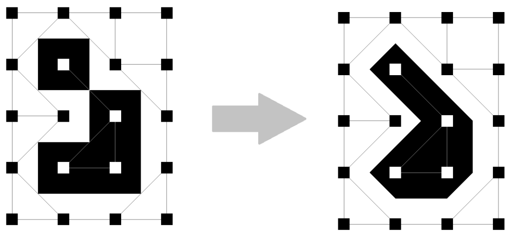

Fig. 3. Voronoi Diagram based on a given similarity graph.

Here, our planar graph always has vertices arranged in a grid pattern. For each 2×2 subgrid, we have 3 cases that can arise, based on the location of the diagonal in the similarity graph. If there is a diagonal, the Voronoi subgraph has an 'I' shape. In case of no diagonal, the Voronoi subgraph has a '+' shape. The figure below demonstrates these cases more clearly.

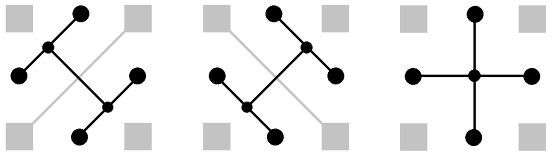

Fig. 4. Voronoi Subgraphs for the corresponding similarity subgraphs. (a) Diagonal from top-left to bottom-right. (b) Diagomal from top-right to bottom-left. (c) No diagonal

The implementation of the Voronoi graph involves the use of a **Half Edge data structure**. We first place vertices between each pair of side-adjacent pixels. For each 2×2 subgrid, we determine if a diagonal is present, and set half-edges accordingly. Within each sub-grid, we may place either 1 or 2 extra vertices, depending on the presence of a diagonal.

Doing this alone will leave all the **border vertices** acyclic, while they must be in a cycle in order to be rendered properly. To resolve this, we add vertices and corresponding edges at the borders, that connect to these vertices and connect the cycle for each of these.

All half edges are arranged in counterclockwise order for each cycle. The faces of the data structure are corresponding to the value of each pixel. For rendering the graph, all the faces are **triangulated using one-ring traversal** about each pixel vertex.

In the code, the class `VoronoiGraph` is used to store the half-edge data structure, based on the similarity graph associated with it. The class composes of several `VoronoiVertex`, `HalfEdge`, and `PixelFace` objects. It supports functionalities for creating the Voronoi graph of a given similarity graph and returning the triangulations that can be rendered by any `ObjectRenderer` object.

4.4 Simplifying the Voronoi Diagram

The Voronoi diagram generated has more vertices than we require. In particular, there are some Voronoi vertices having a degree of 2, that is, just 2 outgoing half-edges from them. For further rendering, it is preferred as we will be replacing all edges with spline curves. Having these vertices in will add unnecessary computational complexity.

In our case, the 2-degree Voronoi vertices are simply the ones we initialized at first, that is, between each edge-sharing pixel position. We can delete these iteratively, by redirecting the incoming half-edges to their respective next vertices.

The simplified Voronoi graph for the example raster image is as follows. The simplified vectorized image has a close resemblance to the intended shape of the image.

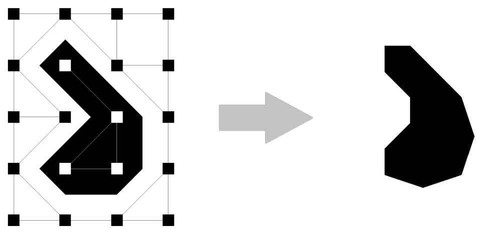

Fig. 5. Simplification of the given Voronoi diagram.

The `VoronoiGraph` class as defined above, has the functionality of simplifying the generated Voronoi graph as well.

4.5 Identifying Visible Edges and Resolving T Junctions

Until now, the steps defined have divided the image into a number of connected shape, each having a particular colour. Our remaining goal is to reshape these segments in order to obtain a smooth image that eliminates artifacts such as staircasing. Our focus henceforth will be on **visible edges**. These are the boundary lines that divide two segments of different colours from each other.

In our implementation, we can check for each half-edge individually if it is visible or not. If the colours on both sides of the edge are of different colours, the edge is easily visible. The information about connected visible edges is stored in the vertices of the half-edge structure; if a visible edge is incoming to a vertex, another visible edge must be going out. Storing this in the vertices makes traversal along visible edges fast.

In the next step, when we interpolate spline curves to all the visible edges, we would want all the splines on sequential edges to be continuous. This is only possible if every vertex has at most 2 visible edges adjacent to it. However, it is common to see 3 visible edges intersecting at a single vertex. This is commonly known as a **T Junction**, and our goal is to select 2 edges from these that can be connected in the final image.

Here, the heuristic is simple. We connect the 2 edges out of the 3 whose angle is the closest to 180 degrees. Choosing this gives the most likelihood that they were intended to be merged, and leads to the best result that does not distort the original graph much.

The figure below demonstrates visible edges and T-junction resolution.

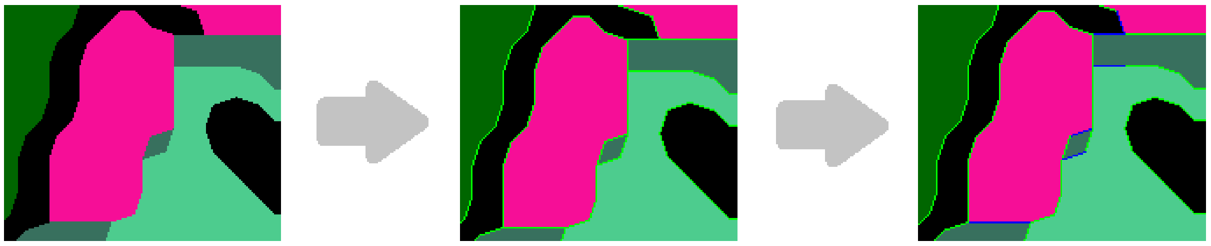

Fig. 6. Resolution of T-junction. (a) Part of Simplified Voronoi diagram. (b) Visible edges highlighted in green. (c) Excluded T-junction edges highlighted in blue.

4.6 Interpolating Edges with Splines

In order to smoothen the image, we replace each visible edge in the Voronoi diagram with a **quadratic Bezier Curve**, with two of the control points of the curve being the ends of the edge. We derive a third (intermediate) control point, keeping in mind that the tangents at each control point must be continuous.

Suppose the two vertices of an edge are v_2 and v_3. Apart from being connected to each other, they are connected to v_1 and v_4 respectively via the adjacent edges.

We can calculate m_1 as the slope of points v_1 and v_3, and m_2 as the slope of points v_2 and v_4. Pass a line through v_2 with slope m_1, and a line through v_3 with slope m_2. The intersection point of these two is our third control point.

This method of calculating control points guarantees that the tangents at a point of intersection are continuous. The figure below shows our example shape's edges interpolated with spline curves.

One challenge with the above is that the interpolation may not be continuous in case multiple parallel edges are sequential, or if the edges follow a strict zigzag pattern. In both cases, the lines of intersection become parallel, and there is no definite solution.

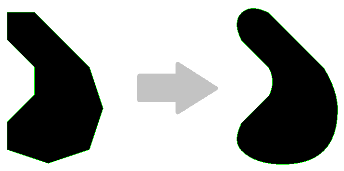

Fig. 7. Interpolation of Edges with spline curves

4.7 Optimisation: Removal of Staircasing

With most pixel art images, it is observed that even after replacing edges with spline curves, there are "zigzag" patterns on certain edges resembling a staircase. It is usually desirable for a generated image to be as smooth as possible. To do this, we can move the vertices of the figure to some other point on the image, to remove the staircasing effect.

There are 2 contradictory qualities that we want to maximize. We wach the curve to be **as smooth as possible**, and at the same time, we want each vertex **not to deflect too much** from its starting position.

To ensure this, we introduce 2 **loss functions**. $E_s^{(i)} = \int_{s\in r(i)} |\chi(s)|ds$
$E_p^{(i)} = ||p_i - \hat{p}_i||^4$

Our goal is to **minimise the sum** of these two values. This can be done by selecting a random set of vertices, and testing a random set of new points to calculate which of then will reeduce in value by assuming a new position. This cycle can be repeated iteratively.

Note that to generate a good quality raster image, the program must run for a long time, running multiple iterative steps sequentially. Since this is a randomisation algorithm, the results need not be optimal. Note that our algorithm computes the integral over a finite range. While the algorithm will attempt to smoothen over this range, it is possible that a local optimisations causes greater curvature to appear elaewhere. This is a major challenge of the randomisation approach.

4.8 Optimisation: Preseving Sharp Aspects

A significant challenge in the above approach is to separate sharpness generated by staircasing, and desired sharpness that was intended by the artist. As a result, the above algorithm can have undesired reshaping of critical objects such as the eyes, or intended shapes such as icons.

To identify whether a pixel is sharp, Koph and Lischinski point out certain configurations on vertices. They are shown as follows:
If a pixel is identified as sharp, their certices' loss function is independent of the smoothness of the pixel range. Thus, vertices that are part of sharp pixels do not deviate from their starting position.

We are now able to preserve the shape of objects having high curvature. For those with low curvature, we are able to flatten them to eliminate staircasing. Shapes having **intermediate curvature are hard to optimise**, as

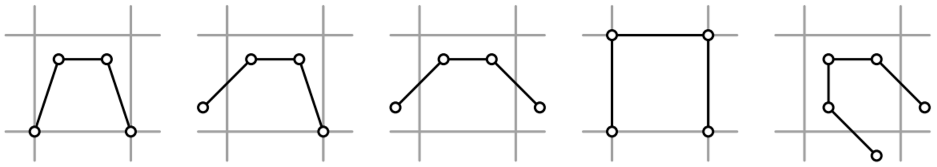
Fig. 8. sharp pixels detected by the program. Source: [1]

it is unclear just by looking at the shape, whether it requires sharpening or smoothening. This may require the use of computer vision concepts, which is outside the scope of this project.

5 RESULTS

The mid-evaluation milestones involved creating a similarity graph, making the similarity graph planar, creating a Voronoi diagram, and simplifying the Voronoi diagram by removing all degree-2 vertices in the Voronoi graph. The images generated had a few shortcomings. The edges were not smooth, and images generated had staircasing artifacts.

The final evaluation milestones covered replacement of edges with splines, and optimising splines for smoothness, while preserving the shape of critical parts of the image. The images generated, as a result, are much smoother, and have far fewer staircasing artifacts.

Below is a set of 4 pictorial outputs generated by our algorithm.

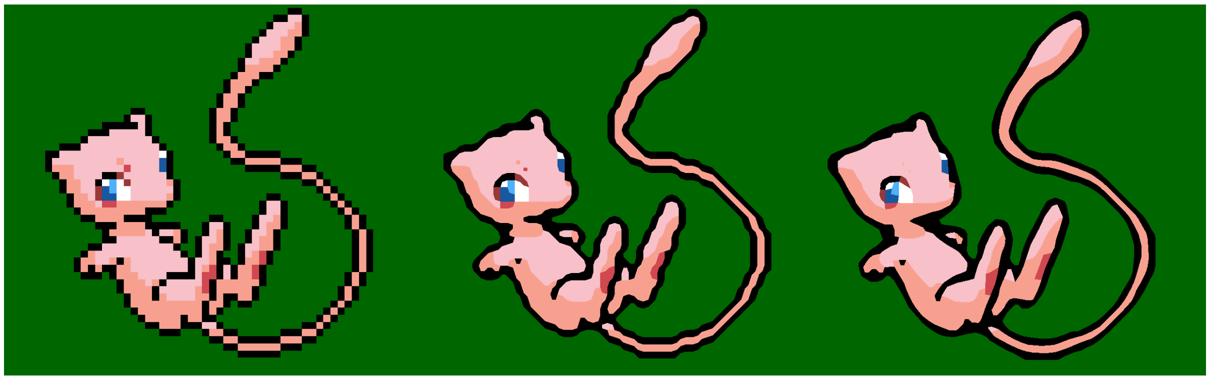
Fig. 9. Depixelisation of a character. (a) Input raster image. (b) Simplified Similarity Graph of the Raster (Mid Evaluation) (c) Optimised Spline-interpolated image (Final Evaluation). Character: Mew, from Pokemon

6 CONCLUSION

The raster to vector converter created in this project works well for pixel art images with **limited and contrasting colour pallets**, such as those in early video games. The quality of the images is an improvement over the original raster image when it comes to graphics.

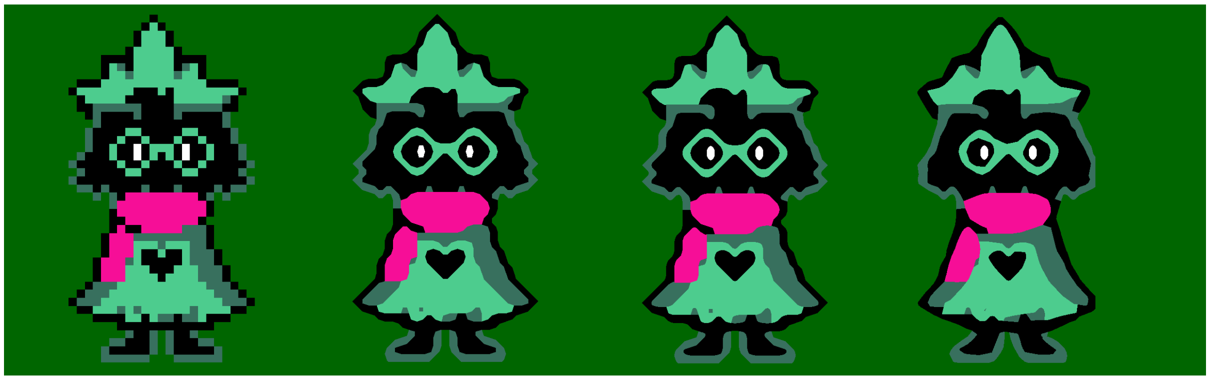

Fig. 10. Depixelisation of a character. (a) Input raster image. (b) Simplified Similarity Graph of the Raster (Mid Evaluation) (c) Spline-interpolated image (d) Optimised Spline-interpolated image (Final Evaluation). Character: Ralsei from Delarune, created by Toby Fox.

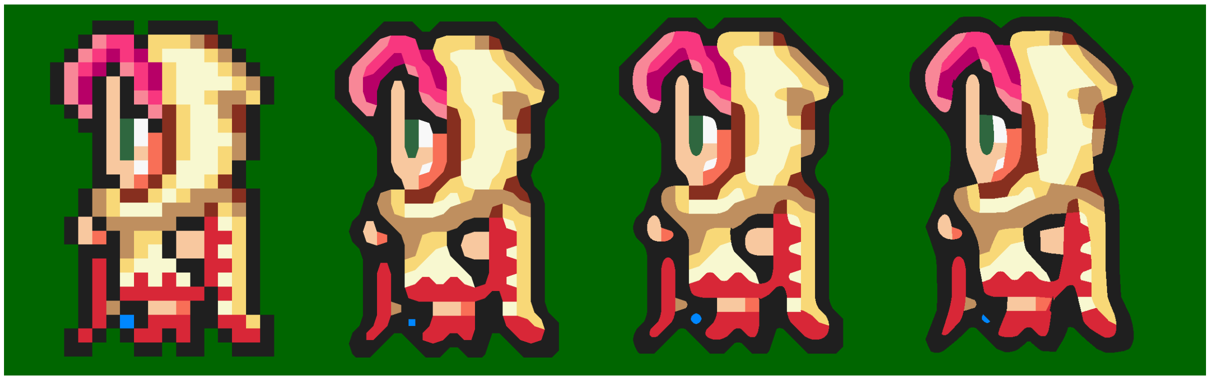

Fig. 11. Depixelisation of a character. (a) Input raster image. (b) Simplified Similarity Graph of the Raster (Mid Evaluation) (c) Spline-interpolated image (d) Optimised Spline-interpolated image (Final Evaluation). Character: Lenna from Final Fantasy 5, developed by Square Enix.

On the other hand, certain **challenges** were faced, which are as follows:

(1) The program does not work for more ‘realistic’ pixel art images, that have a large amount of different colours. Having many different colours in an image makes it difficult for the program to understand the segments of the image or connect/simplify them.

(2) The output after optimisation of the image is not always clean, and **may have rough/sharp edges**. This is because the algorithm is randomised, and the point takes the exact value of that sampled coordinates. To prevent this issue, randomised input may be replaced, or use in addition with, some deterministic heuristics.

(3) For many shapes with an ‘intermediate’ curvature, the shape may be optimised to **higher sharpness** instead of lower. It is hard for a program to tell just by looking at curvature, if the vertex has staircasing

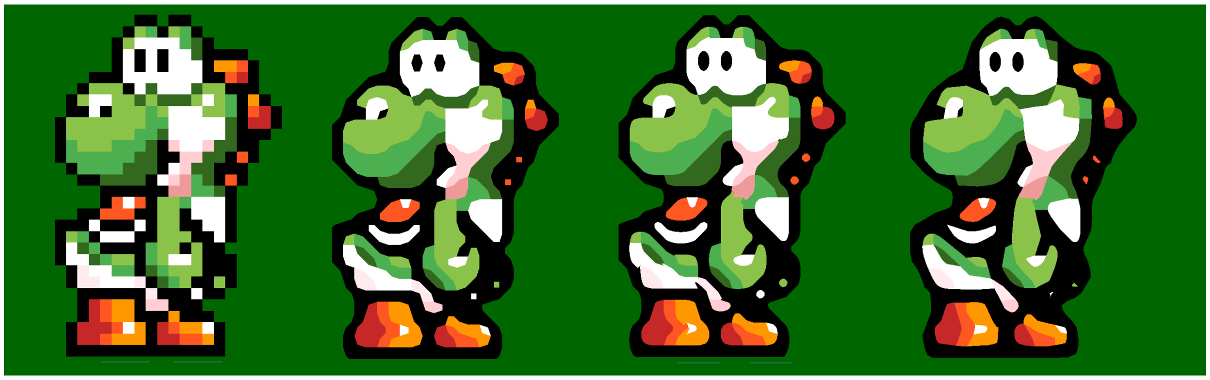

Fig. 12. Depixelisation of a character. (a) Input raster image. (b) Similarity Graph of the Raster. (c) Voronoi Diagram of the graph. (d) Simplified Voronoi diagram. Character: Yoshi from Super Mario World, by Nintendo.

that needs t0 be reduced, or a particular shape that needs to be preserved.

Any future improvement of this model would likely focus on one of these three areas.

The speed of the algorithm indicates that, without optimisation, it may be run on top of a game having pixel art graphics, to improve the game's graphics without changing its aesthetics.

REFERENCES

[1] Johannes Kopf, J. and Lischinski, D., 2011 *Depixelizing pixel art.* In ACM SIGGRAPH 2011 papers (pp. 1-8). https://johanneskopf.de/publications/pixelart/