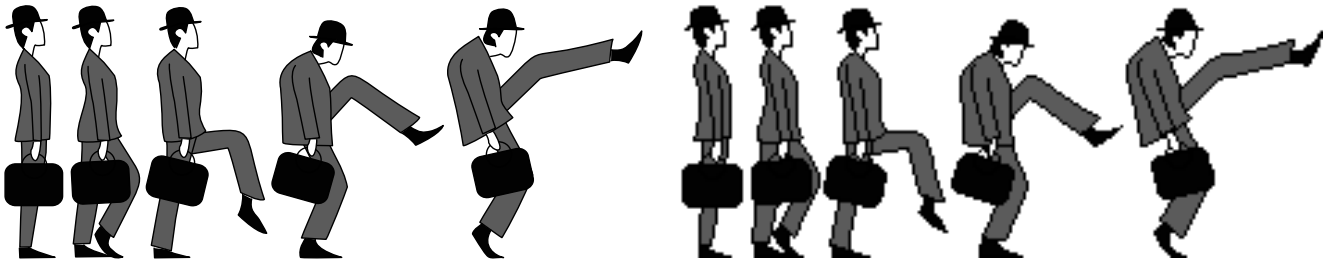
Rasterizing and antialiasing vector line art in the pixel art style

Tiffany C. Inglis*
Computer Graphics Lab
University of Waterloo

Daniel Vogel
Computer Graphics Lab
University of Waterloo

Craig S. Kaplan
Computer Graphics Lab
University of Waterloo



(a) *Vector input* **(b)** *Pixelated output*

Figure 1: *A 5-frame animation in vector form, converted into pixel art by our Superpixelator algorithm.*

Abstract

Pixel artists rasterize vector shapes by hand to minimize artifacts at low resolutions and emphasize the aesthetics of visible pixels. We describe Superpixelator, an algorithm that automates this process by rasterizing vector line art at a low resolution pixel art style. Our technique successfully eliminates most rasterization artifacts and draws smoother curves. To draw shapes more effectively, we use optimization techniques to preserve shape properties such as symmetry, aspect ratio, and sharp angles. Our algorithm also supports "manual antialiasing," the style of antialiasing used in pixel art. Professional pixel artists report that Superpixelator's results are as good, or better, than hand-rasterized drawings by artists.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Bitmap and framebuffer operations;

Keywords: rasterization, pixel art, antialiasing

1 Introduction

The conventional purpose of pixels is to provide small discrete samples of a continuous signal, ideally small enough for seamless reconstruction by the human visual system. In contrast, *pixel art* is a style of digital art that celebrates the aesthetics of visible pixels. The style developed out of necessity when early 8-bit graphics hardware had limited resolutions and colour palettes. It remains popular today in games, mobile applications, and graphic design – even though these limitations are imposed artificially.

Nearly all pixel art is constructed painstakingly pixel-by-pixel, with very little automation beyond flood fills. Automation is difficult because pixel art focuses on low resolution details which are not handled well by automatic drawing tools. By using pixel-based editors, artists are forced to work with individual pixels instead of addressing higher-level problems of shape, outlines, and composition.

A natural alternative is to create pixel art using vector-based illustration software. This approach enables artists to operate at a higher level of abstraction, making tasks like animating sprites for games much easier. But it fails to address the problem of representing vector art on a coarse pixel grid. Many standard rasterization algorithms are available, but these are optimized for infinitesimal pixels and produce unacceptable artifacts at low resolutions (see Section 3).

Our research focuses on *pixelation* [Inglis and Kaplan 2012], a special class of rasterization algorithms designed to respect the aesthetic conventions of pixel art at low resolutions. Every pixel counts in this context, and a pixelation algorithm must therefore consider the colour of every pixel carefully, taking into account its effect on its neighbours. Ultimately, pixelation should mimic the pixels that would be chosen by a human artist.

Pixel art is supported by a large online community where individuals regularly share their work, critique the work of others, and create tutorials. For example, a popular tutorial by Yu [2013] (reproduced in Figure 2) covers the cleaning, colouring, and shading steps involved in developing a finished piece of pixel artwork. By studying the work of pixel artists, and interacting with them directly, we can articulate the conventions they follow and ideally devise algorithms that embody those conventions.

In this paper we present *Superpixelator*, an algorithm for converting vector line art to pixel art. Our algorithm offers substantial improvements over the previous *Pixelator* algorithm [Inglis and Kaplan 2012] As a superset of Pixelator, Superpixelator generates smoother pixelated curves that preserve symmetry, yielding more faithful representations of geometric primitives. It also supports *manual antialiasing*, a form of antialiasing unique to pixel art that uses limited colours to draw clean, smooth outlines. We compare our results to hand-drawn pixel art and line art rasterized using existing software, and report on feedback from professional pixel artists. They report that shapes rasterized by Superpixelator are as good, or better, than those done hand rasterized by artists.

*e-mail: piffany@gmail.com

NPAR 2013, July 19 – 21, 2013, Anaheim, California.

Figure 2: *Selected steps from Yu's pixel art tutorial [2013], showing the evolution of a design from initial pixel line art to a finished work.*

2 Related work

Rasterization is a fundamental problem in computer graphics. Bresenham's algorithm is commonly used to rasterize lines and circles without antialiasing, and has been extended to handle ellipses and spline curves [Hearn and Baker 1986]. Recent advances in rasterization are focused more on improving efficiency by simplifying calculations [Boyer and Bourdin 1999] or exploiting graphics hardware [Liu et al. 2011]. Antialiasing can be used to lend rasterized line art a smoother look. There are many antialiasing algorithms, including supersampling, multisampling [Akeley 1993], fast approximate antialiasing [Lottes 2011], etc. Recently, Jimenez et al. [2012] developed a technique combining morphological antialiasing with multi/supersampling strategies to generate high quality antialiasing with fast execution time. Lines and circles can be rasterized more efficiently using Wu's antialiasing algorithm [1991a; 1991b]. Font hinting can improve the quality of font rasterization at the level of individual pixels, and there are automatic methods for creating and transferring hints between TrueType fonts [Stamm 1998]. Subpixel rendering, such as Microsoft's ClearType, takes advantage of the colour subpixel layout in a liquid crystal display to increase the apparent resolution available for antialiasing.

With the recent resurgence of retro pixel art games, the computer graphics community has approached pixel art as a research topic from various perspectives. Kopf and Lischinski [2011] introduced an algorithm for extracting a resolution-independent vector drawing from a pixel art image. Gerstner et al. [2012] presented a method for abstracting high-resolution images into low-resolution, restricted colour pixel art output. Inglis and Kaplan [2012] developed a real-time algorithm called Pixelator for pixelating vector line art. In a comparison with commercial software, Pixelator produced low-resolution curves that are more visually appealing and have a greater similarity to original vector curves. However, their algorithm does not support antialiasing and it has limitations related to smoothness and symmetry (discussed in depth in Section 3.1) that we address in this paper.

3 Pixelating line art

Creating good pixel line art is challenging because there are many artifacts to avoid. In raster graphics editors, any drawn shape is immediately converted to pixels, and attempting to transform the shape will significantly degrade its quality. For example, most raster graphics editors draw axes-aligned geometric primitives symmetrically by copying pixels from one part of the shape to another, but undesired artifacts are introduced when primitives are rotated (Figure 3a). For vector editors, rasterizing a variety of rotated and scaled ellipses is a good way to test for artifacts. We did this with Adobe Illustrator, CorelDRAW, and Java 2D, and found many cases of asymmetric rasterized ellipses and various artifacts (Figure 4), including: blips, missing pixels, extra pixels, and jaggies.

Blips are single pixels that stick out of smooth curves. A blip occurs when a vector curve lightly grazes a column or row of pixels, causing the pixelated curve to contain a single pixel in that column or row. By repositioning the vector curve as shown in Figure 5a, the blip can be eliminated without significantly changing the perceived shape of the pixelated curve.

Missing pixels are gaps in a curve which should be connected, and *extra pixels* are places where the curve is more than one pixel thick. Extra pixels are the cause of three problems, blip-like pixels sticking out of smooth curves, L-shaped corners where rounded corners are preferred, and pixel clusters that make sections of the curve look too thick and too dark.

Jaggies is a term used by the pixel art community to describe places where a smooth curve looks jagged (not the computer graphic meaning of staircasing). The concepts of *slope order* and *pixel span* are required to explain Jaggies. A smooth curve has a positive slope order if its curvature is positive everywhere, and negative slope order if its curvature is negative. If the slope is not changing monotonically, then its slope order is ill-defined. Each pixelated curve contains contiguous rows or columns of pixels that we call pixel spans. We define the slope of a pixel span to be the slope of its diagonal.

Figure 5b shows a vector curve with positive slope order pixelated in two different ways, one of which has jaggies. The two pixelations can be written as slope sequences: $\{\frac{1}{2}, \frac{1}{2}, 1, 2, 1, 3, 4\}$ for the left and $\{\frac{1}{2}, \frac{1}{2}, 1, 1, 2, 3, 4\}$ for the right. The second sequence is nondecreasing, and therefore has the same slope order as the vector curve. In contrast, the first sequence contains a jaggie because it does not respect the slope order. In fact, we can sort any sequence of pixel spans by slope to obtain a new sequence of spans corresponding to a jaggie-free pixelated curve. Such a curve is said to be in *sorted slope order*.

We also subjected Pixelator [Inglis and Kaplan 2012] to the ellipse test of Figure 4. We obtained results that did not have blips, missing pixels, or extra pixels. Although the algorithm produced fewer jaggies than other rasterizers, jaggie removal is sometimes unsuccessfully. Of greatest concern is that in spite of reducing most artifacts, shape symmetry is not preserved.

3.1 The Pixelator algorithm and its limitations

Our aim is to build on Inglis and Kaplan's [2012] Pixelator algorithm. In this section we describe it in more detail, note its limitations, and discuss how it can be improved.

Figure 6 illustrates the algorithm steps. Each vector path is first split it into monotonic curve segments by dividing at local extrema points and non-differentiable points (Figure 6a). Then, each curve segment is realigned by shifting its endpoints to the nearest pixel centre (Figure 6b). The realignment ensures that the resulting pixelation does not have blips, since all the local extrema points are now on pixel centres. Next, a preliminary rasterization is applied to the curves (Figure 6c). At this point, the pixelated curves are mostly artifact-free, but they may still contain jaggies. Pixelator then attempts to remove jaggies in each pixelated curve by sorting its pixel spans by slope (Figure 6d). Finally, the results are merged to form a completed pixelated path.

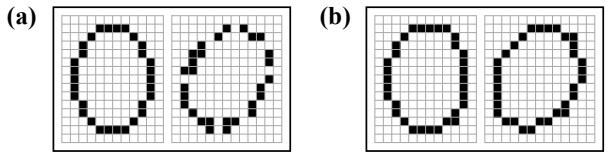


Figure 3: *Rotating an ellipse in (a) raster and (b) vector editors.*

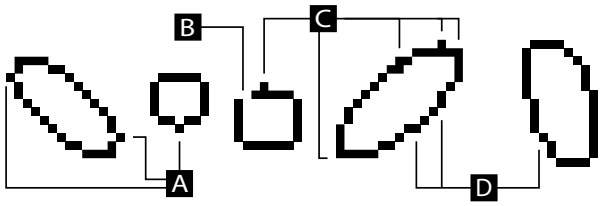


Figure 4: *Artifacts in pixel line art: (a) blips, (b) missing pixels, (c) extra pixels, and (d) jaggies.*

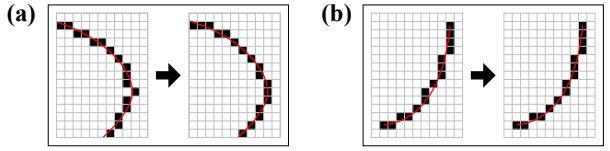


Figure 5: *(a) Blips can be removed via curve realignment. (b) Jaggies can be removed by rearranging pixel spans.*

In some cases, sorting can lead to an inaccurate pixelated representation of the vector curve. For example, Figure 7a shows a pixelated curve before and after slope sorting. Since the original curve consists mostly of pixel spans of slopes 1 and 2, sorting creates a significant deviation between the pixelated curve and the original vector curve, while also producing a noticeable kink in the curve. Due to this problem, Pixelator always calculates the maximum deviation between the two curves, and if the value exceeds the height of one pixel, sorting is not applied. In addition, Pixelator only applies sorting to curve segments with well-defined slope orders. Unfortunately, curve segments with inflection points should be sorted as well (Figure 7b).

4 Algorithm summary

Superpixelator significantly improves Pixelator by introducing bounding box adjustment to preserve symmetry and other shape properties, replacing sorting with partial sorting to increase curve smoothness while limiting the amount of deviation, and adding manual antialiasing to draw clean antialiased paths.

Superpixelator begins by preprocessing the input vector path, which involves adjusting the path by its bounding box (Section 5), splitting the path into curve segments and then realigning each segment by shifting endpoints. The splitting step is modified so that all the curve segments have a well-defined slope order (see Section 6). At this point, the algorithm diverges depending on whether the shape will be aliased or antialiased.

For aliased paths, Superpixelator rasterizes each curve segment and improves rasterization quality by partial sorting (Section 6). Unlike the complete sorting step in Pixelator, partial sorting tries to maximize smoothness while minimizing deviation from the vector curve. As a result, jaggies are almost entirely removed and the pixelated curve remains a close approximation of the vector curve. Once all curve segments are pixelated, the pixels are merged into a complete pixelation for the input path.

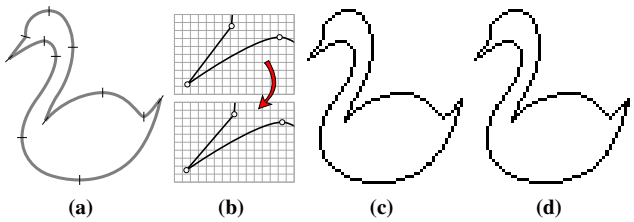


Figure 6: *In Pixelator, a path is first (a) split into curve segments. Each curve segment is then (b) shifted to align better with the pixel grid and (c) rasterized. (d) Finally, the pixel spans are sorted to remove jaggies.*

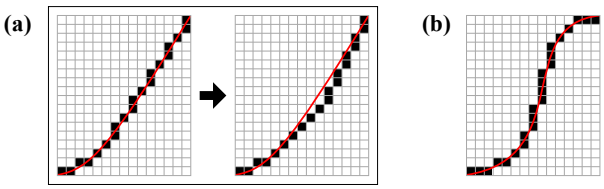


Figure 7: *(a) Sorting may remove jaggies, but the resulting pixelated curve may deviate significantly from the vector curve. (b) Curves with inflection points should also be considered for jaggie removal.*

For antialiased paths, Superpixelator uses a method that closely appoximates manual antialiasing (Section 7). Manual antialiasing is what pixel artists do by hand; it looks cleaner than regular antialiasing and uses fewer colours. Our algorithm draws a thinner path via supersamping, reduces the colour palette, and normalizes opacities to maintain a constant perceived stroke thickness.

5 Preserving symmetry

5.1 Bounding box adjustment

Pixelator does not preserve the overall symmetry of a path because it splits a path into curve segments and shifts them separately. For example, it would split the rotated ellipse in Figure 8a into four arcs and shift each one by its endpoints for better grid alignment, resulting in an asymmetric pixelation (see Figure 8b) even though the original vector path has 180° rotational symmetry about its centre.

To preserve symmetry, let us examine the overall effect of shifting a path's curve segments. The shifting step moves all the critical points—including global extrema—to the closest pixel centres. The new bounding box of the shifted path therefore has corners that lie on pixel centres, as shown in Figure 8c. Knowing this, we can solve the asymmetry problem by first shifting the entire path to the new bounding box, then ensuring that all subsequent steps are performed symmetrically about the box's centre. Operations such as rounding and computing the floor of a number should be made symmetric, and any point that lies on either the horizontal or vertical central axis should not be shifted. Figure 8d shows the symmetric pixelated ellipse created as a result of these changes.

5.2 Preserving other global properties

Preserving symmetry is not always the top priority. Consider the star-shaped path in Figure 9a, for instance. If we shift to the nearest bounding box, the star will be centred horizontally between two columns of pixels (see Figure 9b), causing the topmost vertex to be drawn as a 2×2 pixel cluster.

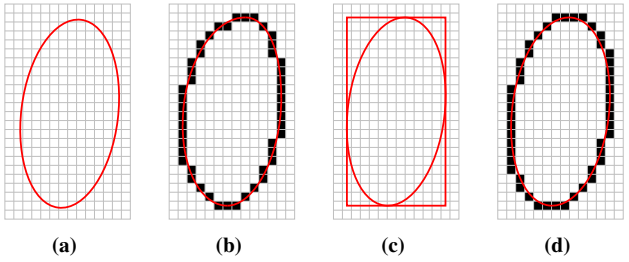


Figure 8: *(a,b) Shifting curve segments causes paths to be drawn asymmetrically. (c,d) Adjusting a path's bounding box and specifying the drawing order can help preserve symmetry.*

We want to keep the acute angle looking sharp by representing it as a single pixel. We can shift the bounding box by half a pixel so that vertex lies on a pixel centre. If we decide to preserve symmetry, then the width of the shifted star (outlined in red in Figures 9c and 9d) would be 12 or 14 pixels; in either case, it would differ from the original 13-pixel-wide star by one pixel. This example demonstrates the difficulty in choosing the most important properties to preserve, whether they are symmetry, angle sharpness, or size.

We can still adjust bounding boxes, but we need to find the optimal bounding box that trades off between several desirable geometric properties. We consider the following properties: symmetry, dimension (i.e., width and height), absolute position, sharpness of acute angles, and aspect ratio. We assign a cost to each possible bounding box in terms of these properties and search for a bounding box that minimized the cost.

Figures 10a and 10b show respectively the bounding boxes before and after adjustment, with various positions and lengths labelled. Note that the box is originally centred on $(x_c, y_c) = (\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2})$, but after the adjustment, (x'_c, y'_c) is not necessarily the box's centre because we want to explore asymmetric options as well.

Using the values labelled in Figure 10, we define the cost of the new bounding box as a weighted sum of costs due to symmetry (C_s), dimension (C_d), position (C_p), acute angles (C_a), and aspect ratio (C_r):

$$C_{total} = k_s C_s + k_d C_d + k_p C_p + k_a C_a + k_r C_r, \quad (1)$$

where each component is defined as

$$C_s = \frac{\max(\tilde{w}_1, \tilde{w}_2)}{\min(\tilde{w}_1, \tilde{w}_2)} + \frac{\max(\tilde{h}_1, \tilde{h}_2)}{\min(\tilde{h}_1, \tilde{h}_2)}, \quad (2)$$

$$C_d = |w - \tilde{w}| + |h - \tilde{h}|, \quad (3)$$

$$C_p = |x_1 - \tilde{x}_1| + |x_c - \tilde{x}_c| + |x_2 - \tilde{x}_2| + \quad (4)$$

$$|y_1 - \tilde{y}_1| + |y_c - \tilde{y}_c| + |y_2 - \tilde{y}_2|, \quad (5)$$

$$C_a = \#\{\text{acute angles with } x = x_c\} \cdot \mathbf{1}_{\{x_c \in \mathbb{Z}\}} + \quad (6)$$

$$\#\{\text{acute angles with } y = y_c\} \cdot \mathbf{1}_{\{y_c \in \mathbb{Z}\}}, \quad (7)$$

$$C_r = \mathbf{1}_{\{w=h, \tilde{w} \neq \tilde{h}\}}. \quad (8)$$

The symmetry term C_s tries to make the left and right halves of the bounding box the same width, and the top and bottom halves the same height. The dimension term C_d measures the change in width and height between the old and the new bounding boxes. The position term C_p tracks the change in the positions of the sides and the central axes of the bounding box. The acute angle term C_a counts the number of acute angles that lie on the central horizontal or vertical axes. The aspect ratio term C_r measures the change in aspect ratio. In most cases, this property is not crucial and is already taken care of by the dimension term C_d. However, for shapes with square bounding boxes (e.g. circle), it is important to preserve the aspect ratio exactly. Therefore, we set $C_r = 1$ if a square bounding box becomes non-square, and 0 otherwise.

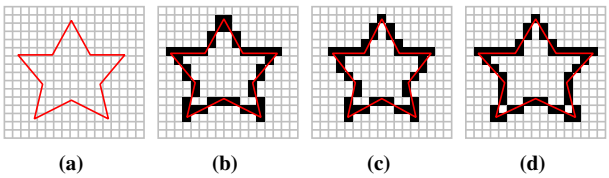


Figure 9: *(a) Pixelating a shape symmetrically is not easy. It can lead to (b) pixel clusters, or make the shape (c) too narrow or (d) too wide.*

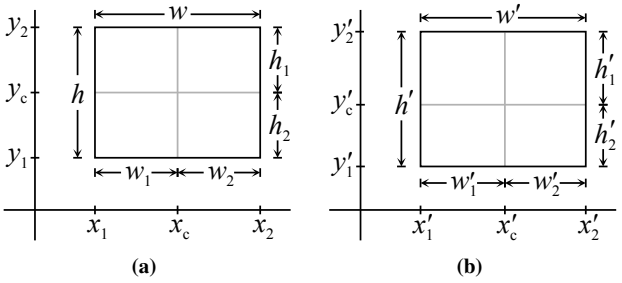


Figure 10: *Bounding box adjustment helps preserve symmetry and other properties of the vector shape after pixelation. The bounding boxes (a) before and (b) after the adjustment as labelled as shown.*

Through trial and error, we arrived at the following set of weights: $k_s = 10000$, $k_d = 40$, $k_p = 1$, $k_a = 10$, and $k_r = 1$. These values indicate that symmetry is still considered most important but other factors are used as tie breakers. To keep the algorithm fast, we only test bounding boxes that are within one pixel of the original bounding box on all four sides. As for the shifted centre (x'_c, y'_c), it must be within 0.5 in both x and y of $(\frac{x'_1+x'_2}{2}, \frac{y'_1+y'_2}{2})$.

6 Curve smoothing via partial sorting

After adjusting a path to preserve global properties, the next step is to pixelate it and remove any remaining artifacts. In this section, we describe how to split a path into curve segments with well defined slope orders, and how to use partial sorting to remove jaggies in a pixelated curve while minimizing its deviation from the vector curve.

6.1 Splitting at inflection points

A curve segment has a well-defined slope order only if it has monotonically changing slope. We can divide a path into such curve segments by splitting it at all non-differentiable points, local extrema, and inflection points. However, splitting at inflection points may cause a problem later in the shifting step. Figure 11a shows an example of a path divided at an inflection point into two curve segments. When each curve segment is shifted to make its endpoints line up with pixel centres, as shown in Figure 11b, the tangent slopes on either side of the inflection point are scaled differently, resulting in a slope discontinuity.

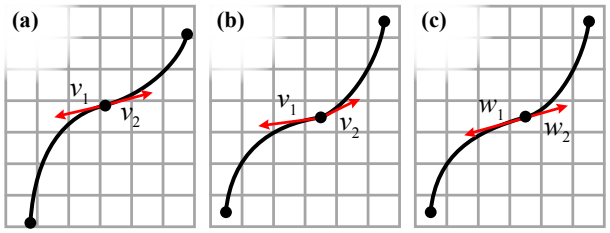


Figure 11: *(a) A curve is split into two at an inflection point. (b) Shifting each curve's endpoints to pixel centres creates a slope discontinuity at the inflection point. (c) Adjusting the tangent vectors fixes this problem.*

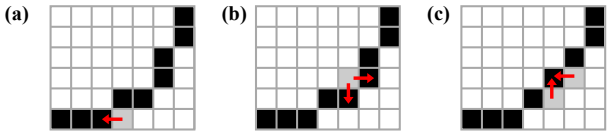


Figure 12: *To find a neighbour of a pixelated curve, either (a) shift a pixel, (b) split a pixel into two, or (c) merge two pixels into one.*

To fix the slope discontinuity, we need to adjust the tangent slopes slightly without changing the curves too much. Let the two tangent vectors be v_1 and v_2. We replace them with new tangent vectors $w_1 = \frac{||v_1||}{||v_2 - v_1||}(v_1 - v_2)$ and $w_2 = \frac{||v_2||}{||v_2 - v_1||}(v_2 - v_1)$. These two vectors point in opposite directions, and have the same lengths as the original vectors (see Figure 11).

6.2 Partial sorting

Next, we rasterize a curve segment by approximating it as a piecewise polygonal path and applying Bresenham's line algorithm. The pixelated curve will not have any blips or missing pixels, but it may contain extra pixels and jaggies. An extra pixel is one that has both horizontal and vertical neighbours, since it can be removed without disconnecting the pixelated curve. We find all such pixels and remove them in decreasing order of their shortest distance to the vector curve.

To remove jaggies, again we use an optimization approach. For each pixelated curve, we define a cost based on its smoothness and deviation from the vector curve, then try to minimize this value. We call this process partial sorting because the resulting pixelated curve will be only partially sorted by slope, but it will be a more faithful representation of the vector curve than a fully sorted pixelated curve.

The cost of a pixelated curve considers positional deviation D_p, slope deviation D_s, and the sortedness of the pixel spans S. To get positional deviation D_p, calculate the shortest distance between the vector curve and each pixel, then take the maximum of these values.

As for slope deviation D_s, we know that each pixel span corresponds to a line segment that approximates a section of the vector curve. So we compare them by taking angle difference between their slopes; the slope of the pixel span is the slope of its diagonal, and its corresponding slope on the vector curve is the tangent slope at the point closest to the pixel span's centre. D_s is defined as the maximum of these angle differences.

Sortedness S is a measure of smoothness. It is defined as the number of pairs of pixel spans that are in sorted order. The cost C is given as a weighted sum of the three quantities $C = 3D_p + 3D_s + S$. The weights are determined empirically and intended to maximize visual appeal.

In practice, we want the algorithm to run in real time, and finding the global minimum for the cost would be too slow. Instead, we use a greedy approach: given a pixelated curve, check its neighbours and accept the one with the lowest cost, if has a lower cost than the current pixelation; repeat until convergence. To find a neighbour for a pixelated curve, either shift a pixel (see Figure 12a), split a pixel into two (see Figure 12b), or merge two pixels into one (see Figure 12c), while ensuring that the pixel spans remain connected. Comparing two neighbours requires only calculating their difference in cost, which is fast because it involves only the modified pixels. We tested a number of shapes smaller than 50×50 pixels, and in all cases, the algorithm converged within 10 steps.

7 Manual antialiasing

Certain shapes just do not look good as pixel art. For example, a line with slope $^2/_3$ cannot be drawn without jaggies (see Figure 13a). In these cases, we need to resort to antialiasing to reduce the impact of visual artifacts. Pixel artists often talk about two types of antialiasing: automatic and manual. Automatic antialiasing is what image editors do when rasterizing a vector image, while manual antialiasing is what pixel artists apply to their art to soften jaggies. There are several differences between automatic and manual antialiasing. Automatic antialiasing is an approximation of the pixel coverage of the vector image, and can be calculated using various antialiasing algorithms, such as supersampling. It does not limit the number of colours used and the result can often look blurry (see Figure 13b). In contrast, manual antialiasing is done by hand using a limited palette (see Figure 13c). When applied to line art, manual antialiasing creates a cleaner look because the lines look thinner and sharper.

Our algorithm mimics manual antialiasing by drawing paths with less blur using a limited palette. Let us assume that we are given a black path of unit thickness, and are asked to pixelate it on a white background using four shades of grey. As summarized in Section 4, we first preprocess the path by adjusting the bounding box, splitting it up and shifting each curve segment. Then manual antialiasing is applied in four steps. In the discussion that follows, we speak interchangeably of the grey level of a pixel and its opacity.

First we select parts of the path that do *not* require antialiasing. In all the pixel art examples we have analyzed, lines that are horizontal, vertical, or of slope ± 1 are almost never antialiased. Therefore we do the same in our algorithm.

Next, we draw the remaining path with antialiasing. To avoid blurriness, we can rasterize a thinner version of each curve via supersampling. For example, Figure 14a shows a curve rasterized with thickness 0.5; the numbers the figure represent pixel opacity values in the range $[0, 100]$. Notice that the resulting pixelated path looks too light in colour; it cannot be fixed simply by doubling the opacities because that makes the overall line thickness look uneven. We need some way of normalizing the opacities so that the pixelated path looks one pixel thick everywhere.

Normalization can be done using the idea of apparent thickness described in Gower's antialiasing tutorial [2013]. In a pixelated path, we can pick a pixel, calculate the total opacity in that row and column, and whichever value is less, divide it by 100 to get the apparent thickness at that pixel. The reason the pixelated path in Figure 14a looks thinner than one pixel is because its apparent thickness is less than 1 everywhere. Although this measure of apparent thickness may not be exactly how we interpret a pixelated image, it works quite well for the purpose of drawing pixel art, and we use it to adjust path thickness during antialiasing.

Figure 13: *A line of slope $^2/_3$ with (a) no antialiasing, (b) automatic antialiasing, and (c) manual antialiasing.*

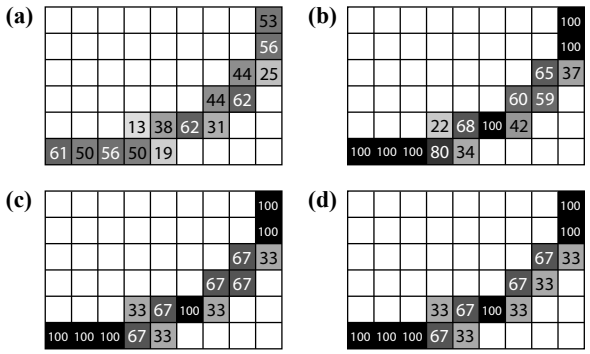


Figure 14: *We apply manual antialiasing by (a) rasterizing a thinner path, (b) normalizing opacities, (c) reducing colour count, and (d) renormalizing opacities.*

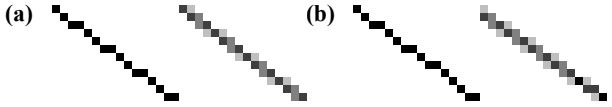


Figure 15: *(a) A pixel artist drew straight lines with regular pixel patterns. (b) Superpixelator did not.*

To normalize the path thickness in Figure 14a, let o_{ij} be the original opacity for the pixel of row i and column j. Then for each pixel, we scale its opacity by $t/\min\left(\sum_i o_{ij}, \sum_j o_{ij}\right)$, where t is the vector path thickness. Figure 14b shows the normalized path.

To reduce the colour count, we quantize each pixel's colour to the closest match in a given palette. Figure 14c shows the resulting path. To ensure the path thickness has not changed too much, we apply normalization again, allowing pixels to be replaced only by colours in the limited palette. Figure 14d shows the final pixelated path.

Different results are produced depending on how thin a path we rasterize initially. The thinner the path, the less antialiasing will be applied, which means the path will look sharper but also more jagged. Based on samples of pixel art with antialiasing, we rasterize paths with a thickness of 0.75.

If a path contains lines that are horizontal, vertical, or of slope ± 1, then it will be partially pixelated without antialiasing. As a result, the pixelated path will look uneven in thickness because antialiasing makes a path look thicker, even with opacity normalization. For such a path, we apply the initial rasterization at 0.5 thickness so that the transition from an aliased to an antialiased segment is less noticeable.

8 Evaluation of Results

Our results are evaluated in two stages with two professional pixel artists. First, both artists compared the pixelation quality of a set of shapes rasterized by Superpixelator, current state-of-the-art algorithms, and hand-rasterized by one of the artists. Second, one of the artists used Superpixelator as implemented in a full-featured pixel art image editor and evaluated the usability and pixelation quality. We were fortunate to have experienced pixel artists participate, Sven Ruthner[1] and eBoy[2]. Ruthner has been practicing pixel art professionally for about a decade, and is well-known within the pixel art community. eBoy is an internationally renowned pixel art group specializing in detailed isometric cityscapes for advertisements.

8.1 Comparison of shapes pixelation quality

To facilitate direct comparison of shapes, we selected a set of 24 common vector geometric primitives. The set comprised six rotated ellipses, six rotated rectangles, six rotated rounded rectangles, and six stars (from 3-pointed to 8-pointed). The shapes are deliberately not aligned with the pixel grid to make the pixelation task more challenging.

We rasterized these shapes using Superpixelator, Pixelator, CorelDRAW, Adobe Illustrator, and Java2D. Note that Pixelator is the only rasterizer that does not support antialiasing. For rasterizers that support antialiasing, we want to create a fair comparison by keeping the number of colours the same. Since Superpixelator uses five fixed shades of grey, we apply nearest-neighbour palette reduction to the other antialiased results so that they use the same palette. To compare these results to actual pixel art, Ruthner rasterized all 24 shapes by hand with and without antialiasing. To get a sense of how much time is involved, Superpixelator took on average 450ms without antialiasing and 950ms with antialiasing. Java2D took about 70ms both with and without antialiasing. Ruthner spent approximately seven hours on the antialiased shapes and one hour on those without antialiasing.

The complete results can be found on our project website[3], under Supplementary Material. However, due to space constraints, only a subset of the results are shown in Figure 16. We chose to compare Superpixelator to Java2D because Java2D's results contain less artifacts than those of CorelDRAW and Illustrator, and unlike Pixelator, it supports antialiasing which can be compared to Superpixelator's manual antialiasing.

For aliased shapes, both artists agree that Superpixelator's results look significantly better than those of the other rasterizers. Our algorithm correctly identifies and removes many artifacts created by the current state-of-the-art rasterizers. eBoy commented that straight lines should be drawn with more regular pixel patterns. As shown in Figure 15, the edges of the fifth rounded rectangle cannot be drawn without jaggies, but even so, drawing them with a repeating $\{1, 1, 2\}$ pixel span pattern is still preferable to a non-repeating pattern.

According to Ruthner, our aliased shapes are very solid representations and as good as the work of any pixel artist. Compared to the hand-rasterized versions, Ruthner believes Superpixelator actually captured some aliased shapes more faithfully (e.g., the third and fourth rectangles). However, we noted a few subtle differences. On the 4-pointed star, the artist drew obtuse angles with L-shaped corners to make them less rounded, whereas Superpixelator avoided such corners. On the 6-pointed star the Ruthner's version is symmetric, whereas Superpixelator sacrificed symmetry for accuracy.

For the antialiased shapes, in some cases Superpixelator's results look similar to the Ruthner's. We made the same choice as the pixel artist to draw certain parts without antialiasing. Ruthner feels that some of our shapes, especially the stars, may be a little too dark and

[1] `ptoing.blogspot.ca`
[2] `hello.eboy.com`
[3] `goo.gl/4JJHu`

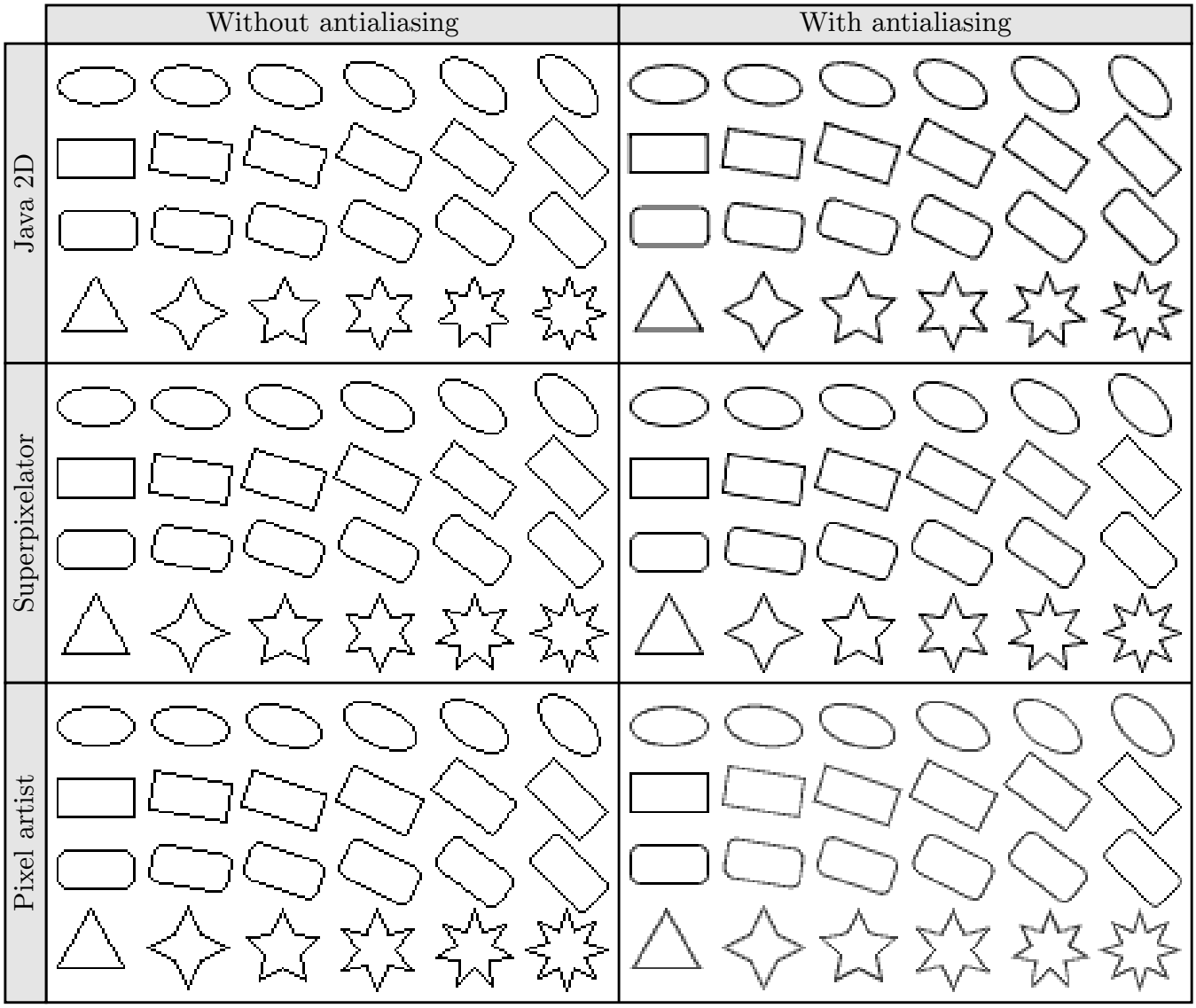
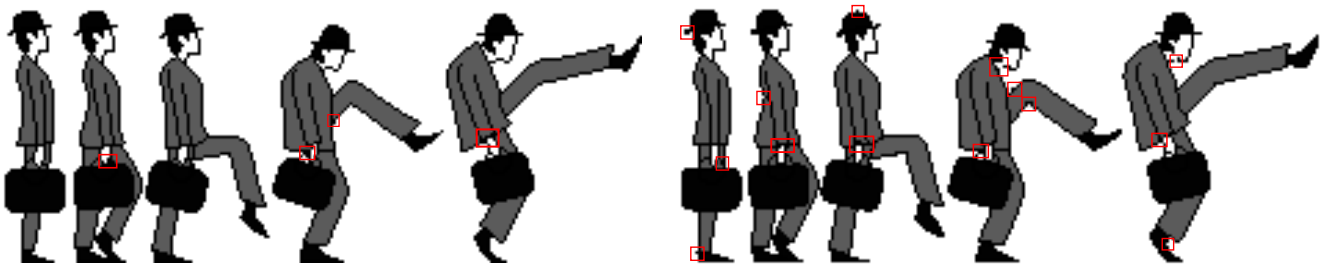


Figure 16: *Vector shapes rasterized without and with antialiasing by Java2D, Superpixelator, and a professional pixel artist (Ruthner).*



(a) *Superpixelator result* **(b)** *Adobe Illustrator result*

Figure 17: *Vector line art rasterized two different way, with red boxes indicating problematic areas.*

too strong on the antialiasing. However, he noted that it may be a stylistic choice, as some of his hand-drawn shapes may also appear too light. Pixel artists also tend to apply antialiasing with regular pixel patterns, as shown in Figure 15. Not only does it create a more consistent look, but it also adds a human aspect to the artwork.

eBoy believes that Superpixelator's antialiasing is superior to that of other rasterizers. However, he feels the aliased slope ± 1 lines look noticeably thinner compared to everything else. We believe this is a matter of taste, as some artists (such as Ruthner) also choose not to antialias slope ± 1 lines.

8.2 Feedback on the pixel art editor

Our goal in the second stage of the evaluation is to see how the algorithm performs in a realistic usage setting. We developed a pixel art editing tool which features the Superpixelator algorithm. The editor supports raster and vector layers. Artists can draw and transform vector line art on vector layers and see it pixelated in real-time using Superpixelator. The editor provides tools for drawing Bézier splines and geometric primitives, which can be transformed via translation, rotation, and scaling. The raster layers enable pixel-level details to also be added. We gave the tool to Ruthner and asked him to comment on the experience of using it to draw pixel art, and the quality of pixelation.

Ruthner found that although the shapes are drawn nicely, the fact the vertices do not always snap to the nearest pixel can be frustrating. This is a result of Superpixelator optimizing across various quality measures, but this comment suggests that a closer correspondence between vector shapes and pixelated shapes is desired during real-time editing. Ruthner also asked for more antialiasing support. For example, a path is often at the boundary between two regions of different colours. He said it would be helpful to be able to choose which side of a path to antialias. Overall, he believes that integrating our algorithm into a pixel art editor will be of great benefit for production work when working under a deadline.

8.3 Discussion

Our evaluation suggests that Superpixelator is better than other rasterization techniques for pixelation, but it is has limitations. To illustrate this, we created a vector line drawing of a walk-cycle (Figure 1a) and contrast the results when rasterized with Superpixelator and Adobe Illustrator (Figure 17). Superpixelator produces fewer artifacts than Illustrator, but important for sprites, the pixelations are also more consistent. For example, in the first three frames, the man's head (including hat, hair, and face) is simply translated, but Illustrator rasterizes it in three different ways.

Superpixelator's results are not perfect, however. Since our algorithm pixelates each path separately, sometimes it encounters pixel clusters where two pixelated paths come into contact. For future work, we will consider the problem of arranging a collection of interacting pixelated paths to minimize such artifacts.

9 Conclusion and future work

Superpixelator is a pixelation algorithm that rasterizes vector shapes in a pixel art style. Shape properties are preserved by adjusting the bounding box to an optimal configuration. Curves are drawn by applying partial sorting, which considers both smoothness and deviation. Manual antialiasing gives pixelated paths a cleaner and smoother look using a limited colour palette. Pixel artists prefer Superpixelator to other rasterization algorithms, and believe our algorithm correctly mimics what pixel artists do.

For future work, we would like to pixelate an entire vector line drawing more effectively by considering relationships between the shapes. Currently, Superpixelator treats shapes individually, so when merging the resulting pixels, some new artifacts can emerge. Finding a method to rearrange shapes to avoid these inter-shape conflicts while preserving the overall topology of the drawing is an interesting problem. It may also be possible to remove or simplify certain features to make an image more suitable for a given resolution. Such a tool would be immensely useful for icon design and sprite creation, where it is often necessary to represent an image at different sizes with varying levels of abstraction.

Acknowledgements

We wish to thank Sven Ruthner and the eBoy artists for drawings and comments they have provided for the evaluation of our work.

References

AKELEY, K. 1993. Reality engine graphics. In *Proceedings of the 20th annual conference on computer graphics and interactive techniques*, ACM, SIGGRAPH '93, 109–116.

BOYER, V., AND BOURDIN, J. 1999. Fastlines: a span by span method. *Computer Graphics Forum 18*, 3, 377–384.

GERSTNER, T., DECARLO, D., ALEXA, M., FINKELSTEIN, A., GINGOLD, Y., AND NEALEN, A. 2012. Pixelated image abstraction. In *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, NPAR '12, 29–36.

GOWERS, D., 2013. NeoTA's antialiasing tutorial. `neota.castleparadox.com/aa_tutorial.html`. [Online; accessed 14-Mar-2013].

HEARN, D., AND BAKER, M. P. 1986. *Computer Graphics*, 1st ed. Prentice Hall, 59–70.

INGLIS, T. C., AND KAPLAN, C. S. 2012. Pixelating vector line art. In *Proceedings of the 10th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '12, 21–28.

JIMENEZ, J., ECHEVARRIA, J. I., SOUSA, T., AND GUTIERREZ, D. 2012. SMAA: Enhanced morphological antialiasing. *Computer Graphics Forum (Proc. EUROGRAPHICS 2012) 31*, 2.

KOPF, J., AND LISCHINSKI, D. 2011. Depixelizing pixel art. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011) 30*, 4, 99:1 – 99:8.

LIU, Y. K., WANG, P. J., ZHAO, D. D., SPELIC, D., MONGUS, D., AND ZALIK, B. 2011. Pixel-level algorithm for drawing curves. In *Theory and Practice of Computer Graphics*, Eurographics Association, vol. 18, 33–40.

LOTTES, T., 2011. FXAA. Tech. rep., NVIDIA.

STAMM, B. 1998. Visual true type: A graphical method for authoring font intelligence. In *Proceedings of the 7th International Conference on Electronic Publishing*, Springer-Verlag, London, UK, UK, EP '98/RIDT '98, 77–92.

WU, X. 1991. An efficient antialiasing technique. In *Proceedings of the 18th annual conference on computer graphics and interactive techniques*, ACM, vol. 25 of *SIGGRAPH '91*, 143–152.

WU, X. 1991. Fast anti-aliased circle generation. In *Graphics Gems II*, J. Arvo, Ed. Morgan Kaufman, 446–450.

YU, D., 2013. Pixel art tutorial. `makegames.tumblr.com/post/42648699708/pixel-art-tutorial`. [Online; accessed 14-Mar-2013].