Pixel Fixer

Semi-Automated Techniques for Correcting Pixel Art

Rares Bites[1]

Supervisors: Elmar Eisemann[1]**, Petr Kellnhofer**[1]**, Mathijs Molenaar**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Rares Bites
Final project course: CSE3000 Research Project
Thesis committee: Petr Kellnhofer, Mathijs Molenaar, Joana de Pinho Gonçalves

(a) (b)

Figure 1: Results produced by our correction method on examples of pillow-shading and banding in pixel art. (a) Pillow-shading: input (left) and two corrected outputs (middle and right). (b) Banding: input (left), taken from *Pixel Logic* by Azzi [1], and one possible result (right).

Abstract

Pixel art can suffer from perceptual artifacts, such as banding and pillow-shading, which result from poor pixel placement and weaken visual quality. Banding occurs when two adjacent pixel segments of different colors align their endpoints along a shared axis. Pillow-shading is a pronounced form of banding, characterized by concentric strips that perfectly follow the object's outline. Prior methods for fixing artifacts in pixel art rely heavily on auxiliary reference images and do not address banding or pillow-shading. We present a semi-automated method that corrects both artifacts using established artistic guidelines and without relying on references. It supports user annotations to resolve ambiguities when multiple corrections are possible and to help preserve the artistic intent. We tested our software tool, Pixel Fixer, on several pixel artworks affected by banding or pillow-shading. Using an error metric that counts pairs of banded pixel segments, we observed consistent reductions across all inputs.

1 Introduction

Pixel art is a fascinating digital artistic style where every pixel is deliberately placed, often with adherence to carefully established rules [1, 2]. The style emerged from the technical constraints of early computer hardware, which forced artists to condense visual detail using limited resolutions and color palettes until the mid-1990s. Yet, what began as a necessity has since transfigured into an artistic choice and a demonstration of implying more with less. Pixel art remains relevant even in today's era of hyper-realistic visuals, continuing to inspire creative expression from video games to accessible editing software [3–5].

Human perception plays a central role in how pixel art is understood. At such small resolutions, viewers must interpret minimal visual cues, and even minor flaws may unintentionally reveal the underlying pixel grid, compromising the subtlety of the style. Due to this perceptual sensitivity, artists developed conventions and guidelines to create compelling pixel art and correct common imperfections [1, 2].

However, creating professional-looking pixel art can be a laborious task requiring precise control and a keen eye for artifacts. Despite the need, no tools exist to automate artifact correction, raising the question:

How can we design semi-automated techniques to correct common pixel art imperfections?

In this paper, we address two types of artifacts: banding and pillow-shading [1, 2]. Both result from problematic pixel placement. **Banding** arises when adjacent pixel segments of different colors align, causing their edges to terminate along the same horizontal or vertical axis (Figure 2). This alignment creates a "staircase" effect, resulting in blocky transitions. **Pillow-shading** occurs when shading layers follow the shape outline perfectly (leftmost image in Figure 1a). It causes banding across nearly all segments and makes objects appear flat by suppressing their sense of volume.

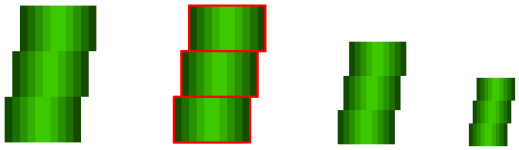


Figure 2: Banding artifact with affected areas marked in red. Banding becomes clearer when rendered at smaller, typical pixel art sizes.

To the best of our knowledge, no previous method has addressed the correction of pillow-shading or banding, leaving both as open problems. Related work addresses other spatial imperfections but relies on a strong assumption: access to an auxiliary reference, such as a correctly-drawn pixel art or a high-quality vector version. As such, their proposed corrections fail in practical scenarios where the only input is the pixel artwork, and the absence of auxiliary images becomes the central challenge to overcome.

Our main contribution is a method to semi-automatically correct banding and pillow-shading (Figure 1), implemented as a software tool titled **Pixel Fixer**. Our method translates established artistic guidelines into automated procedures. The base algorithm can handle these corrections without human input, but its results can sometimes misalign with the original artistic intent. To resolve this and handle cases with multiple plausible corrections, we include user annotations and adjustable parameters, allowing precise control over the output. For both pillow-shading and banding artifacts, our approach is novel and holds significance for the graphics and visualization community by supporting more accessible, high-quality pixel art creation.

2 Related Work

Previous research in pixel art has not focused directly on correcting imperfections, which were typically addressed only as part of broader tasks. In contrast, work on natural images has approached artifact correction more directly, proposing techniques like dithering and anti-aliasing. However, methods designed for natural images do not enforce the rules of pixel art, often failing to preserve its stylistic integrity.

Domain-Specific Pixel Art Correction As far as we are aware, only two approaches address the correction of pixel art imperfections. The first is *Superpixelator* [6], a method that converts vector line art into pixel art while reducing jagged edges and lone pixels that stick out. Although effective at minimizing these flaws, it relies on the availability of the original vector image as a structural reference. Consequently, it cannot correct artifacts when only pixel-level data is available without a vector source.

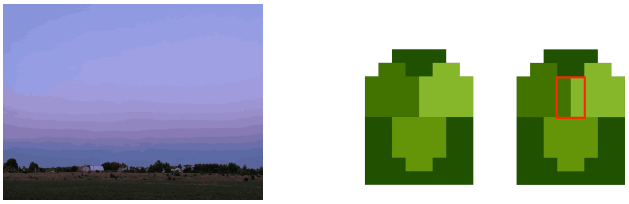
The second approach is a keyframe-based animation technique [7]. It generates intermediate frames from an input pixel artwork by extracting and deforming prominent feature lines. The system reduces visual artifacts in these frames through corrections guided by the feature lines extracted from the input. Thus, the algorithm implicitly assumes that the input pixel art is correctly drawn. This constraint limits the method's usefulness as a general correction tool, as it cannot operate when no reliable reference exists.

Natural Image Dithering Classical dithering algorithms, most notably Floyd–Steinberg error diffusion [8], were developed to address the problem of color quantization in continuous-tone images. Floyd–Steinberg introduces noise to spread quantization errors across neighboring pixels. This approach reduces visible color banding while maintaining the image's volumetric perception and tonal structure despite fewer colors.

Importantly, the nature of banding differs between pixel art and natural images. In natural images, banding is typically defined as visible transitions between stripes of similar colors where smooth gradients should exist (Figure 3a). On the other hand, in pixel art, banding more broadly refers to the illusion of lines appearing along clusters of pixels, even when those pixels are not arranged in a gradient (Figure 3b). Here, the artifacts do not arise from limited color variety, as in natural images. Instead, they come from spatial misalignment, since pixel art already uses a limited color palette. As a result, dithering offers little benefit for correcting banding in pixel art.

However, we draw inspiration from the core principle of dithering: performing localized adjustments based on an error metric, which aligns with our goals. While dithering modifies pixel colors to reduce quantization errors, our approach focuses on adjusting spatial arrangements to avoid structural patterns that weaken shading quality.

Anti-Aliasing Techniques Anti-aliasing consists of methods designed to reduce jagged edges and visual artifacts caused by sampling continuous signals into discrete pixels. Common approaches include: supersampling, which renders the image at a higher resolution before downscaling to smooth edges; multisampling, which selectively samples multiple points within each pixel to improve edge quality; and post-processing filters, which apply blur or blending effects after rendering. In pixel art, anti-aliasing is usually manually performed by artists to smooth transitions or soften lines. Such manual anti-aliasing can also help reduce banding [10], but if applied improperly or automatically, it may itself introduce banding. Some automated methods aim to replicate manual anti-aliasing in pixel art, but are limited to simplified images composed exclusively of pixel line work [6].



(a) (b)

Figure 3: Banding in natural images versus pixel art. (a) Banding in a natural image, producing visible stripes in the sky gradient. Image by Gorbovskii [9], color reduced. (b) Banding in pixel art caused by spatial alignment without an intended gradient (highlighted in red). Image from Pixel Art Historical Society [10].

3 Methods

We introduce two methods: one for banding correction (Section 3.1) and one for pillow-shading correction (Section 3.2). While pillow-shading can be regarded as a particular form of banding, using the same method for both problems proved ineffective. Banding is usually localized, so we let users select specific regions and apply targeted corrections to better preserve artistic intent. However, transferring this approach to pillow-shading, which affects virtually *all* shading layers, would require excessive user input and produce rough-looking results. In fact, pillow-shading presents a broader issue: it creates an unintended front-lighting effect that can make objects appear flat, which artists typically want to avoid. We therefore propose a separate method that, while reducing the banding aspect of pillow-shading, can also improve perception of the geometric shape.

3.1 Banding Correction

Banding occurs when adjacent pixel segments of distinct color align their endpoints along the same horizontal or vertical axis, creating unintended visual lines that draw attention to the underlying pixel grid. The main idea behind our algorithm is to identify pixel segments and analyze their spatial relationships to detect aligned edges that cause banding, then modify these segments so that they don't align anymore.

Figure 4 summarizes the pipeline of the algorithm, hereafter referred to as InteractiveCorrection. The input consists of a pixel art image. The algorithm pre-processes the image by splitting it into adjacent clusters of the same color, which are divided into horizontal and vertical segments (Section 3.1.1). Next, it detects banding by identifying all segment pairs that exhibit problematic alignment (Section 3.1.2).

The user can select one of these segments to correct, and the system shrinks or expands its endpoints to break the alignment (Section 3.1.3). Optionally, users can control the correction step to disambiguate and match the artistic intent.

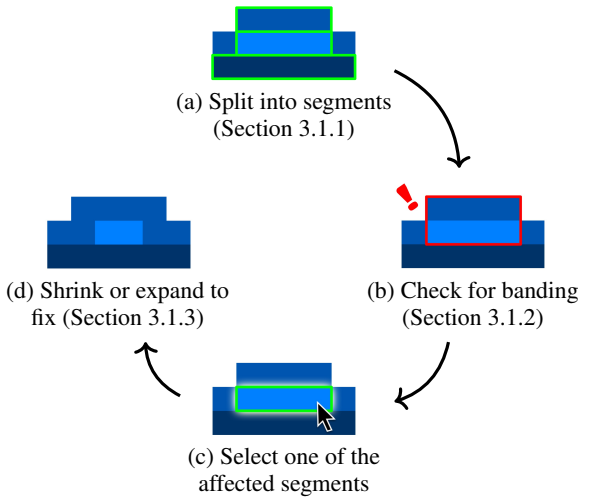


Figure 4: Pipeline of the interactive banding correction algorithm.

We extend the algorithm to AUTOCORRECTION, a fully automatic variant that requires no user interaction. It applies a fixed correction strategy from Section 3.1.3 to all affected segments, repeating until no banding segment pairs remain.

3.1.1 Subject Extraction and Division into Segments

This pre-processing step corresponds to Figure 4a. We isolate the subject from the background and organize its pixels into clusters, with each cluster further divided into segments (Figure 5). We define a **cluster** as a maximal 4-connected region of pixels that all share exactly the same color. Within each cluster, a **segment** is defined as a consecutive linear sequence of pixels oriented either horizontally or vertically. Using 8-connectivity yields different clusters but identical segments, so the chosen connectivity does not affect generality. This preparatory stage is necessary because banding can only occur between two adjacent segments belonging to different clusters; therefore, it reveals all potential areas where banding detection could be applied.

Figure 5: Pre-processing stages of an input image. (a) Input image, taken from *Pixel Logic* [1]. (b) Partial view of the identified clusters (in green). (c) The horizontal segments of those clusters (in green).

For simplicity, we assume the subject is drawn on a white background, which allows its extraction as a binary mask through color thresholding. The algorithm iterates through all masked pixels and applies depth-first search to each unvisited pixel, grouping connected regions of the same color. This yields a list of clusters. We scan each cluster in two separate passes, one per direction, to split into horizontal and vertical segments.

3.1.2 Banding Detection

We now detect banding (Figure 4b). The banding detection heuristic is adapted from *Pixel Logic* [1]. For the purposes of our algorithm, we define **banding** to occur when two segments of different colors (i.e., from different clusters) satisfy the following conditions: both segments are non-degenerate (i.e., length of at least 2 pixels), lie adjacent along the longer edge, and have aligned endpoints along the shared boundary. Figure 6 shows examples of valid and invalid segment pairs.

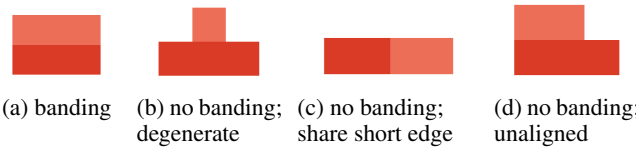


Figure 6: Segment configurations evaluated by the banding criteria.

Let S be a segment. To detect banding involving S, the algorithm examines all clusters other than the one containing S. From those other clusters, it identifies the set $\mathcal{N}(S)$ of **all neighboring segments**. Such segments satisfy two conditions: they share the same orientation as S (horizontal or vertical), and they contain at least one pixel that is 4-neighbor-adjacent to a pixel in S. The algorithm then creates a filtered set $\mathcal{N}'(S) \subset \mathcal{N}(S)$, which discards any segments that lie in the continuation of S, retaining only those positioned adjacent in the perpendicular direction: for horizontal segments, those in the rows above or below; for vertical segments, those in the columns to the left or right.

For each segment $T \in \mathcal{N}'(S)$, the algorithm flags the pair (S, T) as a **banding pair** if both segments begin and end at the same coordinate values along their aligned axis: for horizontal segments, this means the same start and end column indices; for vertical ones, the same start and end row indices. Let $\mathcal{B}(S)$ be the set of such pairs. If $|\mathcal{B}(S)| \geq 1$, the algorithm marks S for correction and stores $\mathcal{N}(S)$ for further use.

Running detection over all segments, we define the **banding error** of the image as the number of unique banding pairs:

$$\mathcal{E}_b = \frac{1}{2} \sum_S |\mathcal{B}(S)|,$$

where the division by 2 prevents double counting.

Pixel Fixer can visually mark each banding pair in an image with a red rectangle (Figure 7) and report the banding error. The interface highlights segments on mouse hover, letting the user select one by clicking. The algorithm then applies banding correction to the selected segment.

3.1.3 Correction Strategies

Let S denote an arbitrary segment. In INTERACTIVECORRECTION, S is user-selected; in AUTOCORRECTION, it is selected automatically as part of the iteration over all affected

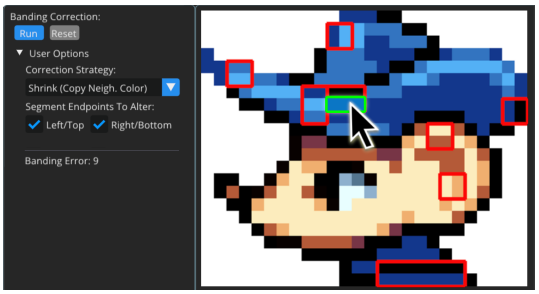


Figure 7: User interface and options for banding correction in Pixel Fixer. Red highlights indicate detected banding pairs. Users can select segments via click. Pixel art created by Azzi [1].

segments. The banding correction step modifies S to fully eliminate banding in all pairs involving S (recall Figure 4d).

Following artist guidelines [1], this correction can be performed by expanding S (adding pixels to one or both ends) or by shrinking it (removing pixels from one or both ends). Because the appropriate choice depends on the image's artistic context, AUTOCORRECTION cannot always determine which correction will give the best visual outcome.

For INTERACTIVECORRECTION, Pixel Fixer optionally allows users to configure the correction. They can select the type of modification (shrink or expand) and specify which endpoint(s) to adjust: for horizontal segments, the left, right, or both; for vertical segments, the top, bottom, or both.

Before detailing the correction strategies, let us introduce key notations and assumptions. Without loss of generality, assume that S is horizontal and only its left endpoint is set to be adjusted. Other cases are analogous. Recall that $\mathcal{N}(S)$ denotes the set of all neighboring segments. We select some segment $T \in \mathcal{N}(S)$ such that the pair (S, T) forms a banding pair. Let the remaining neighbors be $U_i \in \mathcal{N}(S) \setminus T$, with $1 \leq i \leq n$. By construction, T lies on one side of S (above or below), while $U_1, \ldots, U_n$ lie on the opposite side. The following paragraphs describe each correction mode.

Shrink Segment This operation shortens S by removing the specified pixel, thereby breaking the alignments responsible for banding. To remove this pixel, the algorithm replaces its color with one derived from the pixel immediately to its left. Two approaches are possible: directly copying the left neighbor's color (Figure 8b) or averaging it with the original color (Figure 8c). Users can choose the former to preserve the color palette or the latter to achieve smoother transitions.

(a) (b) (c)

Figure 8: Shrink correction strategies applied to the light blue segment in the center of the input image. (a) Input. (b) Copy left color. (c) Average original and left colors.

Shrinking breaks the banding alignment between S and T while preserving visual continuity. If it causes new banding with any segment U_i on the opposite side of S, continue by overwriting the next pixel using the same replacement color. This solves banding on both sides of S.

An edge case arises when the selected segment has length two and its continuation pixels share the same color (Figure 9). Using color averaging on both endpoints causes banding to reappear. If we detect this, we randomly select one endpoint and adjust it, while leaving the other unchanged.

Expand Segment With only the left endpoint adjustable, the expand mode adds one new pixel immediately to the left of S, assigning it the original color of S to maintain consistency. If new banding occurs with any of the segments U_i, a second pixel is added, further extending S in the same direction, using the same color (Figure 10). This ensures banding is fully resolved on both sides of S.

Figure 9: Color average on black segment replaces both ends with an identical color, keeping the banding.

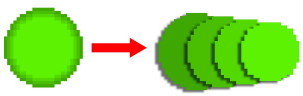
(a) (b)

Figure 10: Expand correction applied to the light blue segment in the input center. (a) Input. (b) Expand left.

3.2 Pillow-Shading Correction

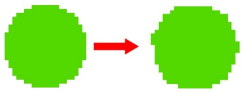
Pillow-shading occurs when shading is organized in concentric color layers of increasingly lighter color that follow the subject's outline almost perfectly. The main idea is to perturb these layers stochastically and optionally adjust their position, breaking the front-lighting effect. We aim to mimic Lambert shading with heavily quantized output colors, such that the color brightness depends on the 3D position of each pixel relative to the light source and the normal of the surface.

Figure 11 summarizes the pipeline of our method. The input consists of a pixel art image containing a single pillow-shaded object. Users decide the presence of pillow-shading, as we only correct but do not detect the artifact. The pillow-shaded region should be visually isolated, meaning no additional elements (e.g., decorative details) occlude or overlap it. Users may add annotations to guide shading adjustments.



(a) Extract the shading layers (Section 3.2.1)

(b) Translate the layers to break front-lighting (Section 3.2.2)



(c) Perturb each layer stochastically (Section 3.2.3)

(d) Construct the corrected image (Section 3.2.4)

Figure 11: Pillow-shading correction pipeline.

The algorithm consists of four main steps: extracting the nested shading layers from the image (Section 3.2.1); translating these layers to break front-lighting (Section 3.2.2);

varying their shapes through stochastic adjustments (Section 3.2.3); lastly, compositing the adjusted layers into a new image as the algorithm's output (Section 3.2.4).

3.2.1 Shading Layer Extraction

We define one shading layer per unique color. Each layer is generated by performing a flood fill on the binary image that highlights all pixels with the corresponding color. If a layer is disconnected, we connect its components by replacing it with its concave hull to preserve geometry (Figure 12).

(a) (b) (c) (d)

Figure 12: Handling a disconnected layer. (a) input image. (b) the shading layer. (c) computed concave hull, preserving geometry. (d) convex hull shown only for comparison; not part of the method.

Assuming the input follows our pillow-shading definition, we sort the resulting layers by increasing brightness. This ensures an outward to inward ordering, with the first layer corresponding to the subject's outline (Figure 13). Let the ordered layers be denoted as $L_1, L_2, \ldots, L_n$. If $n \leq 2$, the algorithm exits early, as no adjustments are required for images with only an outline and a solid fill. Otherwise, it proceeds.

(a) Input (b) L_1 (c) L_2 (d) L_3 (e) L_4

Figure 13: Example of layer extraction: the input image followed by its extracted layer masks L_1 to L_4, sorted by brightness.

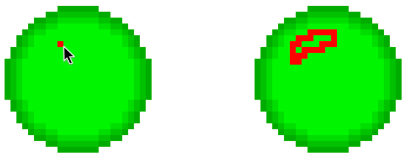
3.2.2 Shading Translation

If no user annotations exist, this step is skipped. Else, the algorithm translates layers L_k through L_n, where the value of k depends on whether the user opts to preserve the shape outline. If the outline is preserved, $k = 3$, leaving the first two layers unchanged; otherwise, $k = 2$ (Figure 14).

Figure 14: Comparison of shading translation options: preserving the shape outline (left); not preserving it (right).

Two annotation modes are supported by Pixel Fixer (Figure 15). The user can either select a **generator pixel** (a location from which light is intended to reflect the most), or draw a shape directly over the image, representing the intended highlight region for the innermost layer. In the second case, we compute the generator as the annotation's center of mass.

Denote by $\vec{v}_i$ the vector from the geometric center of layer L_i (for $i \geq k$) to the generator pixel. We aim to translate outer (lower-index) layers less, keeping them closer to their original positions, while translating more internal layers closer to the generator pixel. To achieve this effect, we translate layer L_i by a vector $f_i\vec{v}_i$, where f_i is a scaling factor that increases with i and ranges from 0 to 1. For ease, we chose $f_i = \frac{1}{n-i}$.



(a) (b)

Figure 15: Possible user annotations, in red. (a) generator pixel. (b) user-drawn shape, intended to replace the innermost layer.

3.2.3 Stochastic Layer Variation

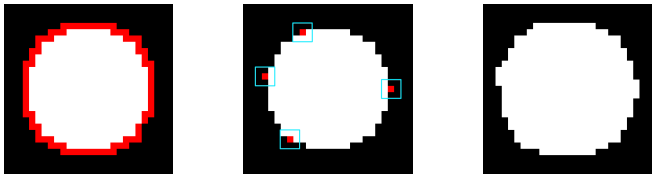
After translation, each layer L_i for $i \geq k$ is perturbed stochastically (Figure 11c). The adjustment ensures enough variation to correct pillow-shading and banding, while still resembling the original shape. We proceed in three steps: a deterministic erosion of the shape; then, a stochastic expansion to introduce variation; and lastly, smoothing to fill potential gaps.

Erosion We erode each L_i using a 3×3 box kernel. Two erosion modes are available in the system: constant (1 iteration) and linear (dynamic number of iterations). In linear mode, the number of erosion iterations scales with the layer index as $i \cdot e_f$, where e_f is a user parameter. This approach allows controlling the size of inner layers to reduce overlap if consecutive shading layers were initially nearly equal in surface area (Figure 16).

(a) $e_f = 0$ (b) $e_f = 0.4$ (c) $e_f = 0.8$ (d) $e_f = 1.2$ (e) $e_f = 1.6$

Figure 16: Effect of the linear erosion factor e_f on final result. Higher e_f values produce gradually smaller, more separated layers.

Stochastic Expansion Following erosion, each layer L_i is expanded back through a custom procedure. We begin by identifying pixels to add to L_i from the 8-connected neighbors of its boundary (Figure 17a). Among these, we call a pixel *candidate* if it has at least three adjacent pixels within L_i, ensuring that adding it would preserve overall shape coherence while introducing variation. Candidates are added to L_i with a user-controlled probability (Figure 17b). We perform dilation to restore the layer's surface area to its original size before erosion (Figure 17c). Figure 18 shows how the probability affects the results.



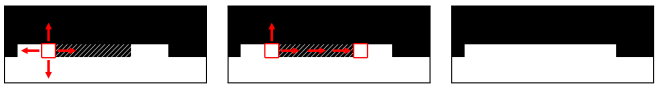
(a) (b) (c)

Figure 17: Illustration of the expansion process. (a) a layer L_i after erosion, in white, with neighbors shown in red. (b) selected candidate pixels, shown in red, outlined by 3×3 squares to highlight their neighbors; each candidate has at least three neighbors, confirming the validity of the candidates. (c) the resulting dilated shape.

(a) $p = 0$ (b) $p = 0.25$ (c) $p = 0.5$ (d) $p = 0.75$ (e) $p = 1$

Figure 18: Effect of the candidate addition probability p on the results. Higher values of p lead to more pixels being added.

Gap-Filling We scan the contour of each layer to close horizontal or vertical gaps by extending pixels along open directions (Figure 19a). Starting from each contour pixel, we follow non-contour 4-neighbors, collecting pixels in a constant direction until reaching another contour pixel (Figure 19b). If found, we add the collected pixels to bridge the gap; otherwise, we discard the attempt (Figure 19c). Figure 20 shows how gap filling improves the result.



(a) (b) (c)

Figure 19: Gap-filling process. (a) portion of a layer with a gap (white) and a contour pixel (red). (b) only the two open directions (top and right) are extensible; we check whether they can eventually reach another contour pixel. (c) if so, we bridge those contour pixels.

3.2.4 Final Image Drawing

Let F_i denote each final, adjusted layer. We composite these layers onto a blank canvas (Figure 11d). To prevent shading from leaking outside the subject, pixels not in L_{k-1} are removed from each F_i, for $i \geq k$. If the user provided a custom-drawn shape, it is flood-filled and replaces the top layer F_n.

To improve results, the full pipeline runs multiple times. In each iteration, the algorithm produces a candidate corrected pixel art image and applies the banding detection method to compute $\mathcal{E}_b$. We select the candidate with the smallest $\mathcal{E}_b$ value as the final result.

4 Implementation

Pixel Fixer is implemented in C++ [11] using Dear ImGui [12] for the graphical user interface and OpenCV [13] for mask handling, erosion, and dilation. We compute concave hulls using an open-source implementation [14], and color brightness via ITU-R 709 luminance [15]. To ensure reproducibility, random operations use a fixed seed.

Figure 20: Effect of gap filling: without it (left) and with it (right).

5 Results

Banding Correction We ran the fully automatic algorithm (AUTOCORRECTION) on one image from *Pixel Logic* [1], under the three correction strategies (Figure 21). Results 21b (color-copy shrink) and 21d (expand) noticeably alter structure, disconnecting outlines. In contrast, 21c (color-averaging shrink) accurately preserves shapes.

(a) $\mathcal{E}_b = 23$ (b) $\mathcal{E}_b = 0$ (c) $\mathcal{E}_b = 0$ (d) $\mathcal{E}_b = 0$

Figure 21: Results of our AUTOCORRECTION algorithm on a *Pixel Logic* [1] image. We set both segment ends modifiable. (a) Input, (b) Shrink (Color Copy), (c) Shrink (Color Average), (d) Expand.

We ran the INTERACTIVECORRECTION on five images from two artists and compared our results to their manual corrections (Figure 22). In 22a and b, our results look similar to the artist fixes and have no larger banding errors, indicating our method performs well in simple cases. The user-driven result in 22a improves upon the automatic ones (see Figure 21) by better balancing the three correction strategies and more accurately reflecting human intent.

In Figure 22c, we fixed banding around the mouth region at the cost of altering the facial expression, which the artist left intact. In both Figures 22c and d, detection shows some banding in segments of the outline color. To address it, we mostly used color-averaging shrink. Color-copy shrink would have disconnected the outline, and expansion would have doubled line width, which artists typically avoid. In comparison, the artist did not introduce new colors and only selectively corrected banding based on aesthetics. These cases suggest that visual quality matters more than eliminating *all* banding.

In 22e, banding is most visible around the face, as broad shaded regions. While the artist removed entire segments to thin the bands, our method allows only shrinking or expanding. Our output appears blurry, but it stays closer to the input by making only minimal changes; in comparison, the artist's fix looks cleaner but deviates from the original.

Pillow-Shading Correction Figure 23 shows the results of our pillow-shading correction algorithm, which reduces the banding error in all inputs. Remaining banding occurs mostly

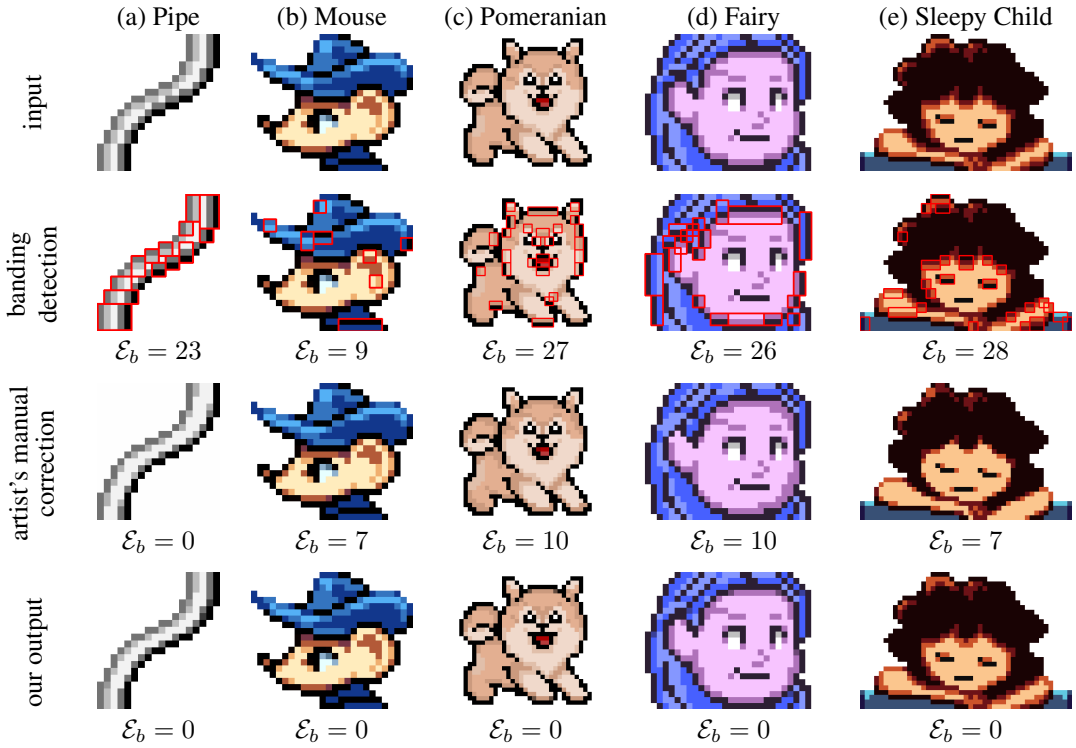


Figure 22: Results of our INTERACTIVECORRECTION algorithm. Inputs (a), (b) and (e) are taken from *Pixel Logic* by Azzi [1]; (c) and (d) are designed by Mateo Nasse. Each column shows the original art, the detected banding artifacts, the manual corrections by the two artists, and our correction. Banding errors are shown below each case.

in the outline segments. Applying banding correction as post-processing can further reduce the error but may not enhance the image visually.

The input image in Figure 23a contains clearly separated layers and a low initial banding error (from the outline). We produced the first output using the default one erosion iteration and no user annotations, confirming that the algorithm performs as expected under standard conditions. In the second output, we annotated a generator pixel which successfully translated the shading.

The Green Circle (Figure 23b) contains closely spaced shading layers and a disconnected layer (with multiple clusters) around the innermost region. The first output uses the default setting of one erosion iteration but exacerbates overlap between layers. The second output applies linear erosion instead, which successfully separates the layers. Both variants connect the previously disconnected layer and preserve the shape of the innermost layer (plus-shaped).

Figure 23c introduces a more irregular shape than previous examples. We annotated the topmost layer similarly to Figure 15b. The first output successfully preserves the Apple's form and shifts the shading towards the annotation to suggest diagonal lighting. For the second output, we disable the outline preservation option, which introduces a broken black outline and causes overlap at the top of the Apple. This result justifies the importance of outline preservation when the outer contour includes elements that must remain separate.

The Butterfly in Figure 23d has a more complex shape with multiple concavities and disconnected layers. In the first input, we can see how the erosion step eliminated the brightest layer due to its original thinness. To address this, we decreased the linear erosion factor for the second output. Both outputs lose the input's symmetry.

In Figure 23e, multiple pillow-shaded petals are separated by an outline. In the first output, we enable outline preservation: as expected, it maintains the black outline and the second darkest layer in their original shapes, restricting drawing to within the second layer. Consequently, most other areas fall outside bounds, which produces an undesirable output. The second output, generated without outline preservation, ignores the outlines but keeps the original geometry.

We can draw the following conclusions. The algorithm performs best when the input image contains a single, isolated pillow-shaded region, as in the first three test cases. It struggles, however, when multiple regions are present or visually connected, as in Figure 23e. We can address this by masking each separate region where the algorithm should be applied. The algorithm also fails to preserve symmetry, as seen with the Butterfly example (Figure 23d). A possible solution is to let the user annotate a symmetry axis, then automatically add matching candidate pixels on both sides.

6 Discussion

This section outlines the contributions of the paper in relation to prior work, identifies limitations, and proposes directions for future research.

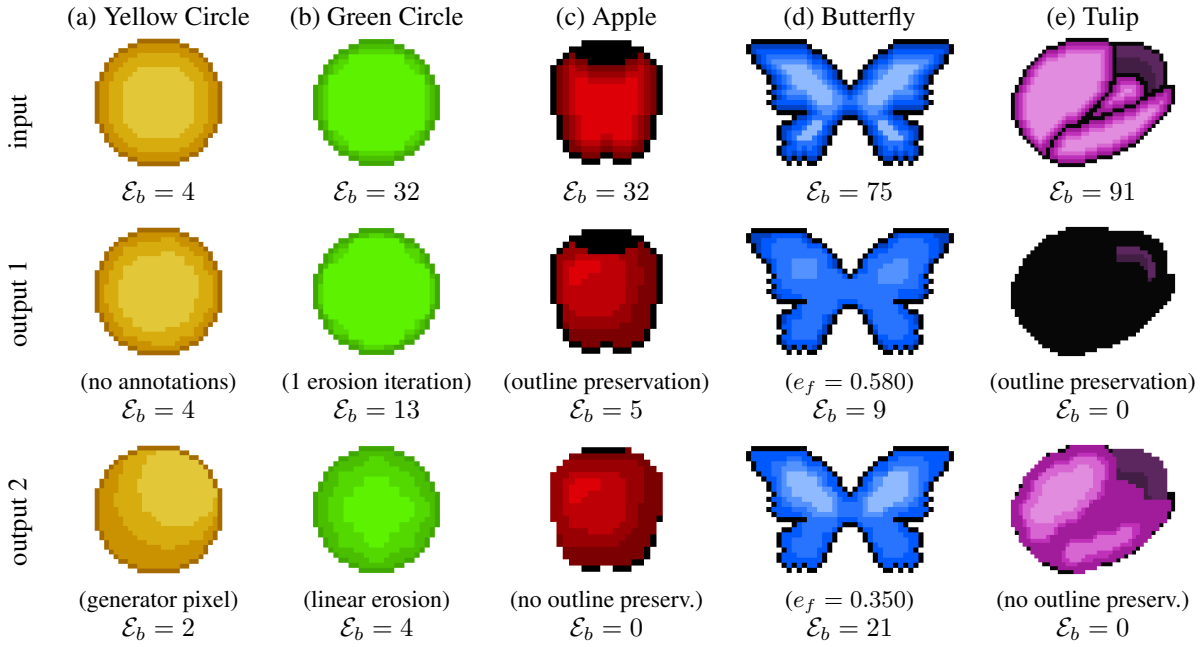


Figure 23: Results of our pillow-shading correction algorithm, ran on original designs. Each column shows the input, followed by two possible outputs. Banding errors are shown below each image.

Contribution Relevance In animation contexts [7], our pillow-shading correction method can offer fast frame-to-frame shading adjustment. In scenes with fixed light sources, the shading must shift appropriately as the sprite moves. Our generator pixel approach (Section 3.2.2) automates these adjustments, considerably reducing the manual effort otherwise needed for redrawing shading across frames.

In the context of AI-driven pixel art systems [16, 17], our banding correction solution can act as pre-processing when pixel art is used as input or as post-processing when pixel art is output, refining the results. Beyond banding correction, the idea of an error minimization pipeline that applies shrink and expand operations to segments until convergence can be a contribution of its own. For example, if guiding vector art is available, the error can be defined by the deviation between the pixel outline and the corresponding vector curve, as in Inglis, Vogel, and Kaplan [6]. This would allow the use of shrink and expand to smooth outlines and reduce jaggedness.

Limitations The pillow-shading correction cannot handle shading layers containing drawn details on top, as they interfere with layer extraction. We can address this limitation by interactively masking obstructive details, then automatically closing the resulting gaps and applying the basic algorithm.

As for banding correction, AUTOCORRECTION does not yet achieve the same quality as INTERACTIVECORRECTION. One solution can be to exclusively use the shrink operations, automatically determining whether to apply color averaging or color copying per segment. To guide this decision, we can introduce an additional error term alongside $\mathcal{E}_b$. This term would aim to minimize the introduction of new colors (possibly through quantization) while preserving the original structure of the image as closely as possible.

Future Work Beyond the directions opened by current limitations, banding itself remains a highly variable problem. Our method focuses on the most restrictive form, where segments align horizontally or vertically with no in-between gaps. Other variants to consider in future work include 45-degree banding, where affected segments align diagonally, and skip-one banding, where affected segments are separated (Figure 24) [18]. The first can be addressed by extending our method to detect diagonal segments and applying curvature to correct them. The second can be addressed by generalizing detection to include segment pairs separated by gaps, not just adjacent ones.

Figure 24: Other forms of banding: 45-degree banding (left) and skip-one (right; in the brown columns).

7 Conclusion

This paper presents a semi-automated software tool for correcting common pixel art imperfections. We introduce a banding detection algorithm that identifies faulty segment alignment, a correction algorithm with effective structural operations, and a pillow-shading correction algorithm supporting both user-annotated and automatic use. Our results show that these pipelines reduce visual artifacts and banding errors effectively, running smoothly with little human intervention. Applications to related areas further demonstrate the broad utility of our methods. With Pixel Fixer, improving pixel art becomes easier, quicker, and less labor-intensive.

8 Responsible Research

The content of this section follows the criteria outlined in the course guidelines on responsible research practices [19].

Research Integrity We ensured that any image used as a test case is original or used with explicit permission from its artist. Michael Azzi granted permission to include pixel art from pages 56–57 of Pixel Logic [1] (Figure 22a, b and e). Mateo Nasse allowed the use of the inputs and artist corrections from Figure 22c and d. We appropriately credited all visual content.

All our methods and ideas are original. To accelerate implementation, we used ChatGPT [20] to generate, debug, refactor, and document code for all features. We manually reviewed, debugged, and improved the code until it functioned as intended. Appendix A contains the used prompts.

Reproducibility We made the full code available via a publicly accessible GitHub repository[1]. We fixed the random seed to ensure consistent results under the same conditions. We provide additional material containing the input images, used parameters, and annotations needed to reproduce the test cases present in this paper. However, we cannot share Michael Azzi's images, as their use is restricted to this paper.

Broader Impact The primary risk arising from our research is the potential for misuse of the software in ways that conflict with its intended purpose. While the algorithms are designed to improve the visual clarity of pixel art, they could also be used to slightly alter copyrighted works, potentially to bypass copyright detection systems or repackage existing art for unauthorized use, including as training input for generative models. We emphasize that the tool should only be used with content the user has the rights to modify and explicitly discourage any use that infringes on artists' rights. Lastly, given growing concerns among artists about automation threatening creative labor, we affirm that this tool is meant to support, not replace, human creativity.

References

[1] Michael Azzi. *Pixel Logic: Pixel Art Tutorials*. 2022 Edition. Self-published, 2022.

[2] Daniel Silber. *Pixel Art for Game Developers*. CRC Press, 2016.

[3] Square Enix and Acquire. *Octopath Traveler II*. Video game. Tokyo, Japan, 2023. URL: https://www.square-enix-games.com/games/octopath-traveler-ii.

[4] ConcernedApe. *Stardew Valley*. Video game. Originally released for Windows. Seattle, WA, 2016. URL: https://www.stardewvalley.net.

[5] Igara Studio S.A. *Aseprite*. Latest release v1.3.7, May 23, 2024. 2001. URL: https://www.aseprite.org.

[6] Tiffany C. Inglis, Daniel Vogel, and Craig S. Kaplan. "Rasterizing and antialiasing vector line art in the pixel art style". en. In: *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*. Anaheim California: ACM, July 2013, pp. 25–32. DOI: 10.1145/2486042.2486044.

[7] Ming-Hsun Kuo, Yong-Liang Yang, and Hung-Kuo Chu. "Feature-Aware Pixel Art Animation". en. In: *Computer Graphics Forum* 35.7 (Oct. 2016), pp. 411–420. DOI: 10.1111/cgf.13038.

[8] Robert W. Floyd and Louis Steinberg. "An Adaptive Algorithm for Spatial Greyscale". In: *Proceedings of the Society for Information Display* 17.2 (1976), pp. 75–77.

[9] Aleksandr Gorbovskii. *A field with houses in the distance*. Photo. Free to use under the Unsplash License. 2022. URL: https://unsplash.com/photos/a-field-with-houses-in-the-distance-Coqv4CE1S_A.

[10] Pixel Art Historical Society. *Banding*. Accessed: 2025-05-21. 2019. URL: https://pixelwiki.comun.se/doku.php?id=paag:banding.

[11] Bjarne Stroustrup. *The C++ Programming Language*. 4th. Addison-Wesley Professional, 2013.

[12] Omar Cornut. *Dear ImGui*. GUI library for C++. 2025. URL: https://www.dearimgui.com.

[13] Adrian Kaehler and Gary Bradski. *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*. O'Reilly Media, 2016.

[14] senhorsolar. *concavehull*. GitHub repository. 2024. URL: https://github.com/senhorsolar/concavehull.

[15] International Telecommunication Union. *Parameter values for the HDTV standards for production and international programme exchange*. Recommendation ITU-R BT.709. ITU-R, 2015. URL: https://www.itu.int/dms_pubrec/itu-r/rec/bt/r-rec-bt.709-6-201506-i!!pdf-e.pdf (visited on 06/10/2025).

[16] Zongwei Wu et al. "Make Your Own Sprites: Aliasing-Aware and Cell-Controllable Pixelization". en. In: *ACM Transactions on Graphics* 41.6 (Dec. 2022), pp. 1–16. DOI: 10.1145/3550454.3555482.

[17] Flávio Coutinho and Luiz Chaimowicz. *Generating Pixel Art Character Sprites using GANs*. Version Number: 1. 2022. DOI: 10.48550/ARXIV.2208.06413. URL: https://arxiv.org/abs/2208.06413 (visited on 04/14/2025).

[18] cure. *The Pixel Art Tutorial*. 2010. URL: https://pixeljoint.com/forum/forum_posts.asp?TID=11299 (visited on 04/14/2025).

[19] Andrea Gammon. *Responsible Research for EWI Research Projects*. CSE3000 – Tutorial/2nd Lecture. Presented at TU Delft. May 6, 2025.

[20] OpenAI. *ChatGPT*. Large language model. 2025. URL: https://chat.openai.com/.

[1] https://github.com/raresbit/pixelfixer (accessed June 22, 2025)

A Usage of Large Language Models

We used ChatGPT [20] to develop a user application that is easily extensible with a general algorithm interface. We also used it to convert verbal instructions into code, write documentation, refactor, and debug. The prompts are as follows.

General Application Interface and Common Files

- Create a base ImGUI application with a left-side menu bar and a right-side image display of arbitrary size. Configure the project with CMake.
- Adjust the layout so that the left menu remains fixed, similar to modern dashboard interfaces. Create a visual theme with a more modern style. Ensure the left and right panels appear visually connected and apply a modern font.
- Implement a dropdown in the menu to allow switching between all pixel art images in the `assets/images` directory. Sort these files in natural string order.
- Enable users to draw a path on a `PixelArtImage` and run a placeholder algorithm by clicking a "run" button. Allow users to switch modes between loading an existing image and drawing on a canvas.
- Refactor the main function into smaller, more manageable methods.
- Develop a method using OpenCV that detects the main subject of the pixel artwork (assuming a white background) and returns a list of the corresponding pixels.
- Create a method that takes a `PixelArtImage` and a list of selected pixels as input and highlights those pixels in red on the screen.
- Create a structure for an extensible left menu that supports adding multiple algorithms. Each algorithm should implement a common interface to enable consistent ImGui rendering, and common methods like "run" and "reset".
- Write an export function to save `PixelArtImage` outputs to a folder called `exports`. Ensure it handles filename conflicts by generating unique filenames.
- Add zoom functionality with a slider positioned beneath the image on the right side of the application.
- Implement a debug system in `PixelArtImage.cpp` and `main.cpp`, allowing the application to store a list of debug lines. Render these lines within `renderCanvas()`.

Banding Detection and Correction

- Update `main.cpp` and `PixelArtImage` to let users select color clusters explicitly within the `PixelArtImage`. Detect these clusters via depth-first search, identifying connected components of identical color.
- Allow users to hover over clusters, highlighting them in red using a separate `highlightedLayer`. This layer must not interfere with existing layers and should render using OpenGL and ImGui.
- Split each cluster into horizontal and vertical segments.
- Identify segments outside the current cluster that neighbor segments from the selected cluster. Match segments based on the below rules:
  - For horizontal segments: aligned if endpoints share same column indices and are vertically adjacent.
  - For vertical segments: aligned if endpoints share same row indices and are horizontally adjacent.
- Draw outlines around all such aligned segment pairs.
- Count unique segment pairs without double-counting (i.e., treat (a,b) and (b,a) as equivalent).
- Develop a variant of `bandingDetection()` that returns the list of all affected segments, not just their count.
- When two segments align, define a helper method `determineReplacementColor()` to create the pixel color replacements and store them in the map `std::unordered_map<Pos, Pixel> finalReplacements`.
- Implement segment modification modes based on the UI implemented. For instance, if a horizontal segment's left endpoint is selected and the operation is "Shrink Segment", remove the left endpoint and replace it using `determineReplacementColor()`.
- Extend `determineReplacementColor()` with an additional parameter to indicate edge type (top, left, right, bottom). Use this to simplify continuation logic by directly querying color from the specified direction.

Pillow-Shading Correction

- Implement a method using OpenCV to detect the main subject of the pixel artwork, assuming a white background. Return the list of extracted pixels.
- Extract the shading layers of the subject as a map from a color to a list of all its pixel positions.
- Erode each layer. Modify the `run` method to store shading layers after erosion. Then implement a `renderUI()` method with a debug mode and a dropdown to select which layer to highlight in red on the screen.
- In `constructCorrectedCanvas()`, take the first debug pixel (which will be called a generator) from `getDebugPixels()`. If found, compute the center of each layer (from index 2 onward) and translate them towards the generator. Apply this only within the subject mask defined by layer 0.
- Render layers from 2 onward only inside layer 1's mask.
- If no generator exists, run `getDrawnPath()`. If the retrieved path is non-empty, convert it into a filled mask, and replace the final layer with it. Compute its center and set it as the generator.

Miscellaneous

- Debug *[insert code here]*
- Write docstring for *[insert code here]*
- Explain *[insert error description here]* and fix it
- Refactor *[insert code here]* for clarity, readability and conciseness