Procedural texturing for pixel art

Making pixel art resemble real materials

Francisco Cunha[1]

Supervisor(s): 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: Francisco Siqueira Carneiro da Cunha Neto
Final project course: CSE3000 Research Project
Thesis committee: Petr Kellnhofer, Mathijs Molenaar, Joana de Pinho Gonçalves

Abstract

In pixel art, texturing is the process of adding detail to an object to make it resemble a real material. While texturing is crucial in creating high-quality pixel art, it is an arduous process that is time consuming for artists. We present an algorithm that automates texturing by arranging elements throughout an input image in a way that implies a feature of the reproduced object, outlines its tridimensional shape, and respects the original shading. We additionally present heuristics on how to determine the color of a feature drawn over existing pixel art, and a method to create vector fields from user brush strokes. Aside from easing artists' work, this algorithm is an important step towards the procedural generation of pixel art. Quality is demonstrated by comparing results to manually authored textures.

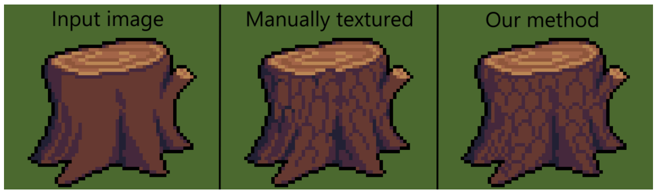

Figure 1: Left: Untextured tree trunk, given as input to our method. Center: Tree trunk textured manually by an artist. Right: Textured tree trunk, produced by our method. Left and center images are drawn by Pedro Medeiros [14].

1 Introduction

Pixel art is a type of digital art created in very limited resolution, such that individual pixels are discernible to an observer. Historically it developed as a consequence of the limited graphical power of early computing systems, but it is still a style often used by artists today either to evoke its nostalgic associations, due to its many practical advantages over high-resolution digital art, or simply because many find it aesthetically pleasing [19].

The constrained scale of pixel art means artists have to take additional care with the color and placement of each pixel, and often can only afford a handful of pixels to denote an object or object feature. However, even with the low amount of detail artists are still able to create images that appear as if they're made from real materials through what they call texturing – a technique that combines usage of colors, shading, placement of small details and repeating pixel patterns [16]. While texturing is an important aspect of the pixel art look, it is an arduous and time consuming process, and part of the reason the style is difficult to procedurally generate.

Schlitter describes a manual pixel art texturing technique which consists of arranging repeating elements into a pattern [18]. "Element" is used to mean a very small pixel art image that represents a feature of the real material, e.g. bark scales in Figure 1. In this paper, we present an algorithm that automates this technique for pixel art texturing, while allowing for large amounts of artist control over the final result. Our approach leverages user annotations to create a vector field, then places elements throughout the image which are oriented according to the vector field. Each element's color may be taken from the image's palette according a choice between two heuristics, avoiding the introduction of new colors.

This tool allows pixel artists to focus on the broader creative decisions and later detailed improvements, while automating part of the manual repetitive labor that comes in between. Combined with other procedures that generate base shapes for our tool to texture, one may create fully procedural high quality pixel art, expanding the realm of possibilities in computer game design. To our knowledge, we are the first to present a procedural texturing technique that works for pixel art.

2 Related work

Pixel art processing Many algorithms have been developed to bring pixel art to different mediums, such as vector images [8, 13], as well as from other mediums to pixel art [4, 6, 10, 20]. However, we have found little work on processing pixel art while remaining in the same artistic medium. Methods exist to create pixel art animations from static pixel art [9], and to extract normal maps from pixel art [15]. Our work is novel in that it is, to our knowledge, the first in automating part of the process of making pixel art itself.

Procedural texturing Procedural texture generation is a problem that has been extensively studied in computer graphics, see Dong et al.'s 2020 work for a recent survey on the subject [3]. However, the process of texturing in pixel art is very different to the one in high-resolution images or 3D models, since it does not involve tiling images or projecting 2D images onto 3D models.

Element arrangement Element arrangement is a method that is closer to our definition of texturing, as it entails placing repeating elements close to one another in a way that builds a pattern, while following user-defined constraints such as boundaries and directions [5]. Previous work on 2D element arrangement focuses on vector graphics or high-resolution images [5], or 3D models [12], while we explore how to adapt these methods for the constrained resolution of pixel art.

Vector fields from brush strokes Vector fields are a widely used structure that are usually created through precise mathematical definitions. We know how to direct regions of a tensor field according to a user's brush strokes [2], but that requires an existing tensor field whose regions can be modified. Diffusion curves present a method to smoothly vary colors throughout a pixel grid based on some initially colored pixels [17]. While originally applied to colors, the diffusion curve method can also be used to vary directions [1], an idea which we expand upon by gathering the initial directions from user-drawn brush strokes.

3 Texturing pixel art

Our goal is to synthesize textured pixel art from an input image that depicts an untextured or partly textured object. To achieve this, we arrange repeating elements throughout the

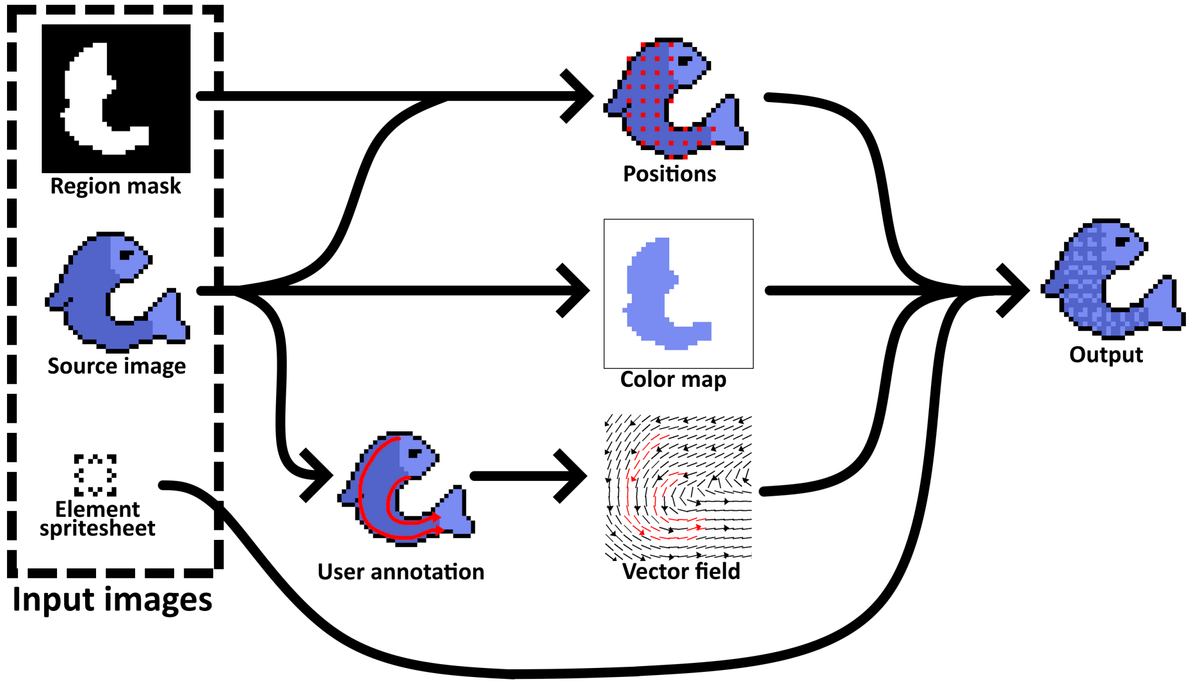

Figure 2: Overview of the algorithm. Each arrow roughly corresponds to a step, pointing from its inputs to its output.

input image. These elements must imply a feature of the reproduced object, outline its tridimensional shape, and respect the original shading.

Three images are required as input. First is the source image that will be textured; second is a binary mask that delineates which region of the image will be textured; third is a sprite sheet of the element that will be placed, with eight variations of it oriented 45 degrees apart from one another. We provide multiple strategies for computing position and colors that are selected depending on the desired output, as no single strategy works in all scenarios. Additionally, several parameters can be set to fine tune the results: element margins (m), density (ρ), HSV displacement (Δ_{HSV}), excluded colors, and max color distance, all of which are described in the following subsections.

Our method is broadly comprised of four steps, which are illustrated in Figure 2. The first three are preprocessing steps in which we compute attributes for each element: their positions, colors and orientations. These three steps can be done in any order, as their outputs are used simultaneously in the final step, placement. Section 3.1 explains how the position of each element is computed; Section 3.2 details how to create color maps which inform the colors elements take; Section 3.3 shows how user annotations are leveraged to create a vector field that informs the orientation of elements. The final step, explained in Section 3.4, uses these attributes to draw elements on the image.

3.1 Position

This step's goal is computing a list of suitable positions to place elements at. There should be no overlap between two different elements (except if negative element margins are used), and all elements should be contained within the region defined by the input binary mask.

Assume we have a potentially suitable position that may be chosen to place an element at. If we choose it, the region that will be occupied by its corresponding element is marked as unavailable by removing the equivalent pixels from the binary mask. The parameter margins m consists of a tuple of two integers m_x and m_y. We add (or remove, in case of negative values) m_y pixels to the top and bottom and m_x pixels to both sides of the region considered as occupied by the element, which allows elements to be placed further apart or closer together. Figure 3 shows this process, with an element being placed at the position indicated by the red dot and using $m = (2, -1)$, the resulting mask is shown in the rightmost image.

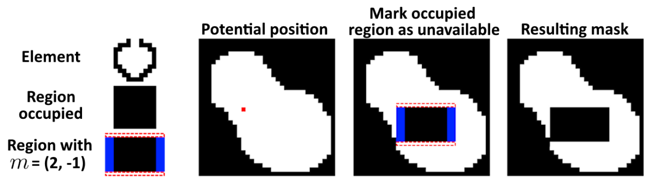
Element
Region occupied
Region with m = (2, -1)
Potential position
Mark occupied region as unavailable
Resulting mask

Figure 3: Process of marking a region as occupied so that a position is added to the list. White pixels represent available positions and black unavailable. Considered position is marked by red pixel.

A position is only considered valid if the element that would be placed at that position fits entirely within the mask, with one exception: the user may choose to allow elements

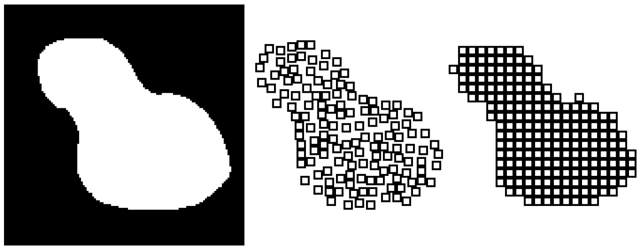

Figure 4: Left: Input binary mask. Center: Positions obtained through the random positions strategy. Right: Positions obtained through the packed positions strategy.

to be placed only partially within the original bounds. In this case overlapping elements are still forbidden, but only one of the element's pixels has to be inside the original input mask for it to be placed. This is repeated until all suitable positions are occupied.

The parameter density ρ is a percentage value that gives control over how full the texture should appear. After a position is validated, it has $\rho\%$ chance of actually being added to the list of suitable positions, while the region it would occupy is still considered occupied.

To find potential positions, two strategies are available. This is to accommodate for two possible types of textures: regular or "packed" textures, and irregular or "random" textures. In both strategies the first position tested is a random position within the original mask, and they differ in how they choose subsequent positions after the first.

Random positions In this strategy, each subsequent position is chosen at random. This results in irregular, natural looking patterns, as exemplified in the center image of Figure 4. This strategy is suitable for many textures, such as the bark shown in Figure 1.

Packed positions This strategy keeps a queue of positions to try and when one position is selected, we add its eight immediate neighboring positions to this queue. If none of the neighbors were suitable, the queue becomes empty. In this case, if there are still positions available within the mask, the next position is chosen at random. This results in completely regular patterns, as exemplified in the right image of Figure 4. This strategy is suitable for some textures such as fish scales, as shown in Figure 2.

3.2 Color

To determine the color of each element, a color map is computed from the input image. When each element is placed, the color map is sampled at that region in order to determine the element's color. Our goal is to choose colors that are both aesthetically pleasing when placed against the colors in the same region of the input image, and also do not stand out visually, since the purpose of texturing is to add detail, not striking features.

Three different strategies are presented to compute the color map: HSV shift, color difference, and shared border. The simplest is HSV shift, which requires additional input, while the other two are heuristics that try to achieve our goals while using colors already present in the image.

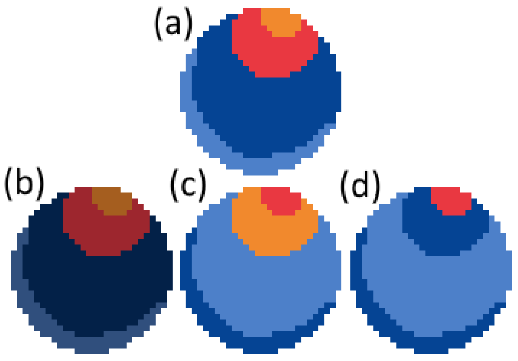

Figure 5: a. Input image. b. Color map obtained by shifting its HSV by $(0, 0, -30\%)$. c. Color map obtained through the color difference strategy. d. Color map obtained through the shared border strategy.

HSV shift This strategy shifts the hue, saturation and value of each pixel by the user-determined amount Δ_{HSV}. That is, taking c_i^{in} to be the color of pixel i in the input image and c_i^{out} to be the color of the equivalent pixel in the color map, $c_i^{out} = c_i^{in} + \Delta_{HSV}$. Figure 5b shows an example color map obtained through this strategy, where $\Delta_{HSV} = (0, 0, -30\%)$.

Color difference In this strategy, first the image's palette is extracted, then each color in the image is mapped to the color in the palette with lowest Euclidean distance to it in CIELAB color space, except itself. CIELAB was chosen as it is a perceptually uniform color space, i.e. a color space in which Euclidean distances correspond directly to perceived color difference [11]. It is possible that multiple colors in the input are mapped to the same color. Figure 5c shows an example color map obtained through color difference.

Shared border At each contiguous color region, we sample the pixels that border the region, choose the color that is most frequent in those pixels, and map the entire region to that color. Certain colors may be excluded as possible coloring options, e.g. black may be excluded to remove a black outline's influence on the region's color, and in case an excluded color were to be selected, the next most frequent color is selected instead. Figure 5d shows an example color map obtained through shared border.

Each of these strategies fit different scenarios, and we additionally present a procedure to automatically select one of the strategies based on the input image.

Automatic strategy selection We start by counting the colors in the image, except those that are excluded, and if they amount to one, the HSV shift strategy is selected. If more than one color is available, a color map is created with the shared border strategy. For each color region in this map, the Euclidean distance in CIELAB color space between it and the color of its equivalent region in the input image. If all these distances are smaller than the maximum color distance parameter, the shared border color map is selected, and otherwise we use the color difference method.

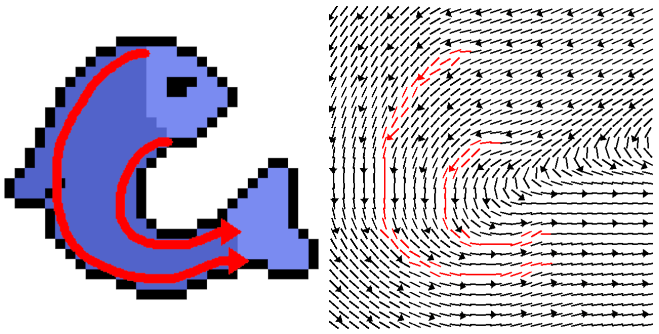

Figure 6: Left: User makes annotations through brush strokes directly on pixel art. Right: Annotations are converted into vectors for each annotated pixel (in red), and pixels without annotations have vectors diffused from others (in black).

3.3 Orientation

Since elements are used to represent features of the real-world object, the orientation of those features needs to be taken into account when placing elements. In order to orient each placed element in the final image, user annotations are used to gather a direction for each pixel where an element can be placed.

As a first step, the system asks the user to draw freehand curves (i.e. brush strokes) on top of the input image, according to the way they wish the elements to be oriented. An example of these brush strokes is shown on the left image of Figure 6. Because of the typical low resolution of input images, it can be difficult to precisely draw on them. To mitigate this, during the annotation step we present a version of the image that is upscaled using nearest neighbor interpolation by an user-determined factor. Each annotated pixel in the upscaled image gets assigned the vector that points to the next annotated pixel. We then downscale the image by the same factor, merging vectors by their average.

Following this procedure many pixels may be left without an assigned vector, as the user does not need to draw over the entire image. Let $\mathcal{A}$ be the set of pixels that are annotated, and V_k be the vector assigned to pixel $k \in \mathcal{A}$. To generalize the annotated directions to the unannotated pixels, we make use of the insight by Bezerra et al. of using diffusion curves to smoothly vary directions in a grid where some points have fixed values [1]. The vectors are therefore given by the solution to a discrete Poisson equation:

$$\Delta I = 0,$$
$$I_k = V_k,\ \forall k \in \mathcal{A}$$

where I is the image, Δ is the Laplace operator. Setting ΔI to zero gives us an uniform diffusion. Figure 6 illustrates how the vector field is built from user annotations.

3.4 Placement

In the final step, all three computed attributes are used to place elements. For each position an element will be placed at, we compute the average vector of all pixels covered by the element and find which element variation is oriented closest to

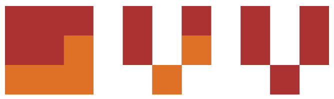

Figure 7: Left: Color map at the region element is placed. Center: Resulting element when using per-pixel coloring. Right: Resulting element when using region coloring.

that direction. This element is then placed over the input image at that position, and either each of its pixels take the color of its equivalent pixel on the color map (per-pixel coloring) or all its pixels take the mode color of the region its placed in (region coloring), as chosen by the user. Figure 7 shows a comparison of these two alternatives.

As explained in Section 3.1, if a certain parameter is set, it is possible that some elements do not fit entirely within the region mask. To account for this case, pixels that would fall outside of the mask are not placed, such that some elements are only placed partially.

4 Results

Our system is implemented in Python, using OpenCV and NumPy. It runs on a Intel(R) Code(TM) i5-12500H, GeForce RTX 4060, with 16GB RAM. For all the images we generate the algorithm runs in less than one second, excluding the time taken by the user in drawing curves, with the majority of this time being taken up by the Poisson equation described in Section 3.3. All input images are either drawn by us or sourced from publicly available online blogs of individual pixel artists, and accompanied by attribution in the latter case.

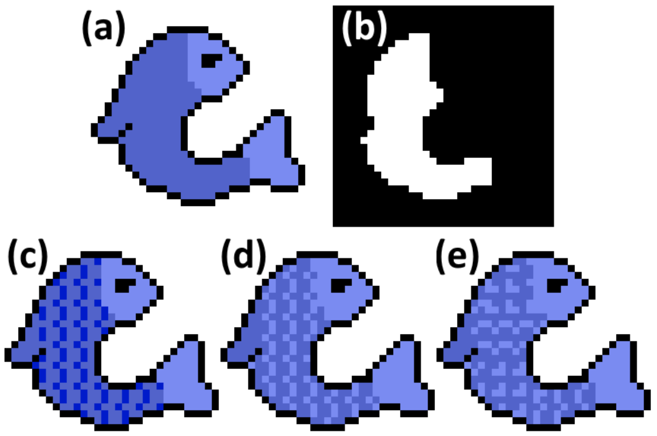

Figure 8: a) Input image of an untextured fish. b) Input boundary. c) Fish textured with the same orientation (down) for all elements and color chosen by HSV shift of (0, 75%, 0). d) Fish textured with colors chosen by shared border, and same orientation as (c). e) Fish textured with elements oriented to follow the body's shape, and same color as (d).

Figure 8 provides an overview of the different effects our method achieves. On the top left, it shows the untextured fish used as an input image. The bottom row shows three results of our algorithm. On the bottom left is a fish with a scale texture

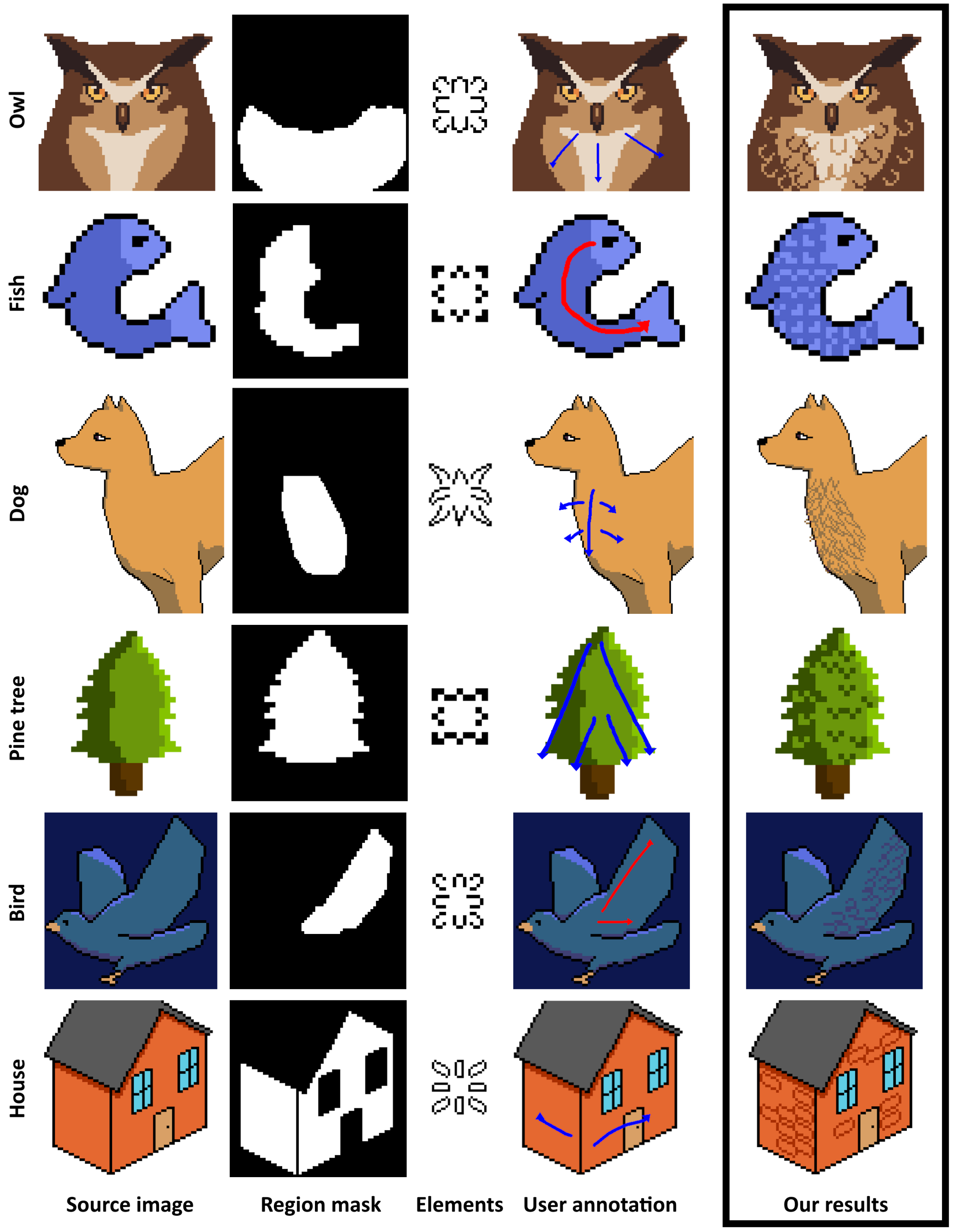

Figure 9: Results of our method for different texture types. The first three columns are input images, the fourth columns shows the brush strokes drawn by the user over the source image, and the last column shows the textured image generated by our method.

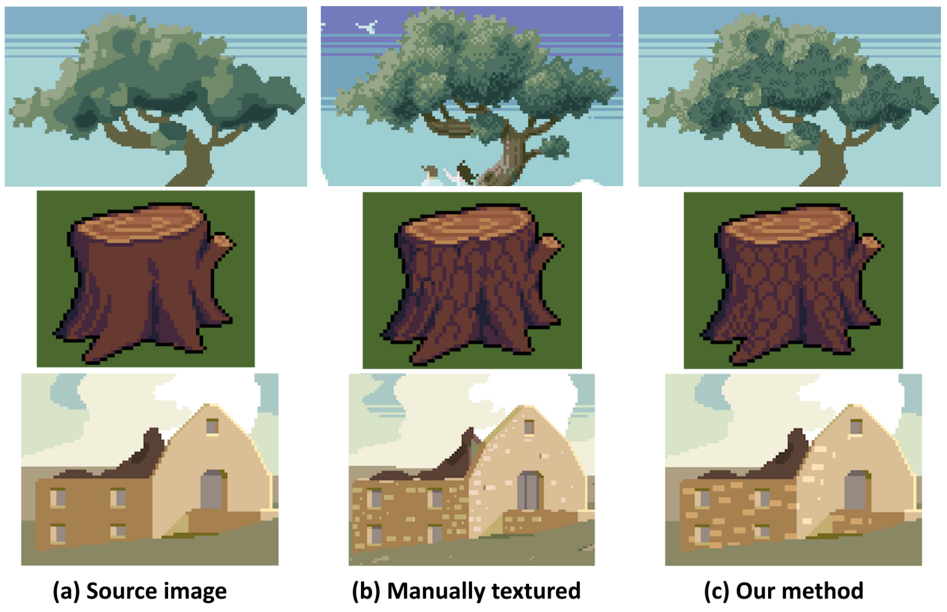

Figure 10: Comparison between our method and results achieved through manual texturing. Top (b) and bottom (b) images are drawn by Raymond Schlitter [18], top (a) and bottom (a) are recreations of those. Center (a) and center (b) are drawn by Pedro Medeiros [14].

that is entirely oriented in the same way, making the fish look flat. Note that the texture is contained within the boundaries of the binary mask given as input, shown in the top right. The bottom center image uses the input's own palette for coloring, making the textured version appear more harmonious with the original. Lastly, the bottom right image shows the texture as oriented by user annotations, which more clearly delineates the fish's shape.

We present several images textured by our method in Figure 9, alongside the input images and annotations required to generate them. Rows 2 and 4 (fish and pine tree) exemplify texturing images of very small resolution (32x32). The method is capable of supporting sharp orientation transitions, such as going from one wall of the house to the other in row 6, as well as smooth orientation transitions between very disparate orientations, such as the fish in row 2 whose scales are oriented along an entire half circle. Real world objects with round or cylindrical shapes are reproduced in rows 1 and 4 (owl and pine tree). Both regular and irregular relative position of elements are supported, as shown respectively in row 6 (house) and row 3 (dog).

The center column of Figure 10 shows textured pixel art that was manually crafted. We attempt to recreate these images using our method, the result of which can be seen in the right column of this same figure.

5 Discussion

This study set out to build an algorithm that procedurally adds texture to pixel art in order to make it resemble a real material. We discuss the results of this algorithm in Section 5.1, the advantages and disadvantages of the different coloring strategies in Section 5.2, and the algorithm's and study's limitations in Section 5.3.

5.1 Achieved textures

Our algorithm enables the synthesis of many textures types, including foliage, bark, brick (shown in Figure 10), feathers, fish scales and fur (shown in Figure 9). Since elements come from user input, it also boasts enough flexibility to support several more, according to the user's creativity.

The position parameters of density and margins give control over the texture's appearance. For example, it supports creating both dense, disordered textures such as the fur on Figure 9's dog, and sparse, regular textures such as the brick on Figure 9's house. The binary mask additionally gives precise control over what region of the image should be textured.

By allowing element orientations to vary according to user annotations, the created textures are able to effectively communicate the shape of the object that is represented in the pixel art. For example, both owl and dog in Figure 9 appear to have a curved chest, as elements go from downwards-left to downwards-right orientations, as would happen when facing a cylindrical object head on. Subtler shape changes are also shown for example on Figure 9's bird, whose feathers start turned to the right and gradually angle upwards, suggesting a slightly bent wing lifted up. A similar effect is seen on the center row (tree trunk) of Figure 10, where the bark starts out angled on the tree's roots and turns upwards to follow the trunk's shape. Some simple sharp orientation transitions are also supported, such as the bricks in Figure 9's house that align with the wall's sharp turn cause by the isometric perspective.

The colors of textures shown make the added features blend well into original images. They help textures appear as subtle details that are visually pleasing and do not stand out. Additionally, the colors are selected from the original image's palette, such that the common pixel art limitation of number of colors used is not violated.

Finally, while our method requires much less effort than manual texturing, comparing our results to manually textured images, as in Figure 10, reveals promising similarities.

5.2 Coloring strategies

In Section 3.2, we describe three different strategies for determining the color of each element. Each of these strategies has its own advantages and drawbacks, which influence which of them is most appropriate according to the input image.

HSV shift Because each output color is a transformation of its respective input color, this strategy enables subtler textures with colors very similar to the ones where they're placed. However, it introduces new colors to the palette, which is generally undesirable – a core aspect of pixel art is a limited and fixed number of colors, often carefully selected by artists.

Color difference This strategy tries to create subtle textures by minimizing the starkness of color changes, but it can use colors from an unrelated part of the image which results in visually discordant elements. This case is shown in the center image of Figure 11, where the most similar color to the mid brown on the tree's exterior is the light brown from inside the tree. This creates discordant elements, as the lighter brown is not used in the area where the elements are placed. For comparison, the right image of Figure 11 shows the same input image with better coloring, using the shared border method.

Shared border This strategy makes use of the fact that artists will generally place colors that work well together next to each other, and frequently selects the color that was used to shade the area. However, it might present an issue if the area contains a sharp color transition, such as going from orange to blue in Figure 12, as it might select the dissimilar color, creating an element that stands out against the background, as shown in the center image. Another issue arises from the common usage of black outlines in pixel art – since these generally border an entire shape, black is frequently selected by the strategy, but usually contrasts starkly against where the element is placed. This issue is mitigated by allowing the user to exclude black from possible colors.

In all, the shared border strategy is the most reliable. Since it makes use of the artist's intent when placing colors next to one another, it is more likely to select a color that creates visually pleasing details, even if that color is not the most similar to the region it's placed at. In the failure case of very distinct colors being placed next to one another, the color difference strategy is the best alternative, as it avoids both this issue and introducing new colors to the palette. The HSV shift strategy should only be selected to achieve effects that go against the goal of subtle element coloring, or if not enough colors are available in the palette. These considerations are taken into account for the described method of automatic selection, which prefers the shared border strategy given its reliability, unless it detects a color difference between input and output that is too stark – then choosing the color difference strategy – or if the palette contains only one color – then choosing the HSV shift strategy.

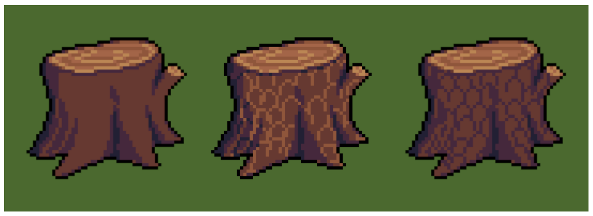

Figure 11: Fail case of the color difference strategy, where a color is taken from an unrelated area of the image. Left: Input image. Center: Textured tree trunk, produced by our method with the color difference strategy. Right: Textured tree trunk, produced by our method with the shared border strategy. Left image is drawn by Pedro Medeiros [14].

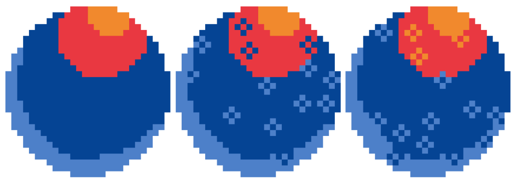

Figure 12: Fail case of the shared border strategy, where a stark color transition leads to discordant elements. Left: Input image. Center: Textured sphere, produced by our method with the shared border strategy. Right: Textured tree trunk, produced by our method with the color difference strategy.

5.3 Limitations

Not all texture types can be synthesized by our method, as the technique described by Schlitter [18] is not suitable for materials that do not exhibit repeating features, such as metals which are commonly identified in pixel art by continuous reflection streaks [14]. Additionally, the algorithm places only equal elements throughout an image, such that it cannot produce textures that require a great variety in the elements placed.

The center columns of Figure 13 shows attempts at creating textures for which our method is unsuited. The cloud shown on the top row requires differently sized elements to be textured, similar to what's shown in the top right image.

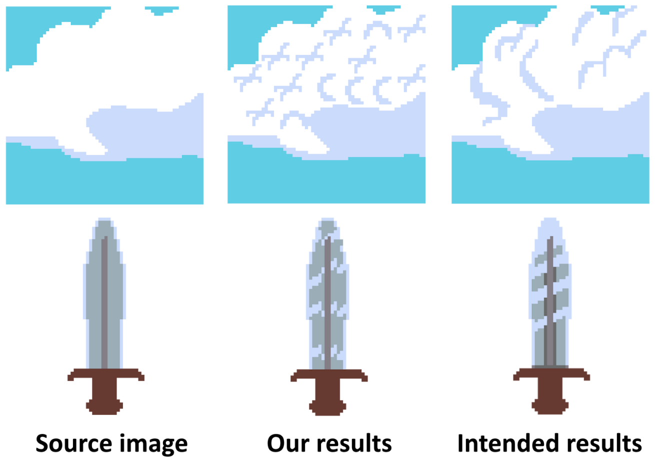

Figure 13: Textures for which the algorithm fails.

The bottom row's sword could use reflection streaks to appear metallic, however our method places streaks without continuity which shatter the illusion of light being reflected.

Finally, this study's evaluation is limited in that we are unable to devise formal metrics of quality, and lack a quality assessment performed by an independent party. Additionally, since the algorithm's intended usage is during the process of creating pixel art, it requires "incomplete" pixel art as input, making it difficult to find test data. Consequently, the majority of input data is created by us, limiting the number of examples we are able to present.

6 Responsible Research

In the course of this research, we have aimed to uphold the Netherlands Code of Conduct for Research Integrity in abiding by its five principles of honesty, scrupulousness, transparency, independence and responsibility [7].

We uphold the principle of honesty by presenting our algorithm's results alongside the input required to achieve them, with the precise parameters used for generating each figure detailed in Appendix A. This principle is further taken into account when reporting all known limitations of the research in Section 5.3.

Our research methods have been made reproducible by precisely describing the algorithm in Section 3. In order to clarify any questions about implementation details, we additionally make our source code available at https://github.com/franciscunha/pixel-art-texturing/. All pixel art that exemplifies our algorithm throughout the paper is publicly available online, and either is properly attributed to its authors or was created by us. Through these actions, we aim to abide by the principles of scrupulousness and transparency.

We additionally increase the research's transparency by reporting that large language models (LLMs) were used on occasion. These tools were used mainly as an aid in the algorithm's implementation in the Python programming language. In this sense, they served to clarify and help fix specific programming errors, to find existing implementations of given features in Python libraries, or to generate code that is conceptually simple but laborious to write. On two occasions, LLMs were used to clarify the usage of certain terminology in existing literature. LLMs were not used to aid in the manuscript's writing, nor in the development of the algorithm itself. We provide a full list of the prompts used in Appendix B.

Lastly, we uphold the responsibility principle by "acknowledging the fact that a researcher does not operate in isolation" [7, p. 13], and considering the broader societal implications of our research. As with any tool that automates or simulates human labor, we recognize the risk of our research being expanded upon in a way that seeks to make the people who currently do that labor redundant. We wish to emphasize that though we enable the procedural generation of pixel art, our tool requires annotations and parameter tweaking to produce high-quality results, promoting the close involvement of artists in its usage.

7 Conclusions and Future Work

This paper describes an algorithm that procedurally adds detail to pixel art in order to make represented objects look more like they are made from real materials, that is, an algorithm that partly automates pixel art texturing. Our method works by placing small user-provided elements that resemble a feature of the real object throughout a user-determined region of the given image. We derive the position and color of these elements through simple heuristics, and their orientation from a vector field built from user annotated brush strokes. Several parameters can be tweaked to achieve a desired look. By using our system, artists can bypass the laborious process of drawing each element by hand throughout the entire image.

We make two additional contributions, as we derive (1) heuristics on how to determine colors for features that are drawn over existing pixel art, and (2) a method to create vector fields from user brush strokes. Contribution (1) is a necessary step in any future work on intradomain pixel art processing – given that the limited usage of colors is core to this art form, any introduced element in pixel art should adhere to the image's palette. Meanwhile, contribution (2) improves the usability of any software that makes use of vector field as inputs, which is significant given their wide applications. While these two methods are useful first steps, further research is warranted to develop their robustness.

Future research may investigate how elements can be drawn in order to resemble given texture types, and propose how to generate these procedurally. The algorithm can be extended with different heuristics for element's colors and positions. Additionally, improvements can be made to its runtime, by making optimizations such as parallelizing the computation of color, position and orientation. Finally, one could integrate our method as part of a larger procedural generation system, enabling procedural generation of high-quality pixel art.

References

[1] Hedlena Bezerra et al. "Diffusion constraints for vector graphics". en. In: *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering - NPAR 10*. Annecy, France: ACM Press, 2010, pp. 35–42. DOI: 10.1145/1809939.1809944.

[2] Guoning Chen et al. "Interactive procedural street modeling". en. In: *ACM Transactions on Graphics* 27.3 (Aug. 2008). Publisher: Association for Computing Machinery (ACM), pp. 1–10. DOI: 10.1145/1360612.1360702.

[3] Junyu Dong et al. "Survey of Procedural Methods for Two-Dimensional Texture Generation". en. In: *Sensors* 20.4 (Feb. 2020), p. 1135. DOI: 10.3390/s20041135.

[4] Timothy Gerstner et al. "Pixelated image abstraction with integrated user constraints". In: *Computers & Graphics* 37.5 (Aug. 2013), pp. 333–347. DOI: 10.1016/j.cag.2012.12.007.

[5] Takashi Ijiri et al. "An Example-based Procedural System for Element Arrangement". en. In: *Computer Graphics Forum* 27.2 (Apr. 2008), pp. 429–436. DOI: 10.1111/j.1467-8659.2008.01140.x.

[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] KNAW et al. *Netherlands Code of Conduct for Research Integrity*. nl. 2018. DOI: 10.17026/DANS-2CJ-NVWU. URL: https://phys-techsciences.datastations.nl/dataset.xhtml?persistentId=doi:10.17026/dans-2cj-nvwu (visited on 20/06/2025).

[8] Johannes Kopf and Dani Lischinski. “Depixelizing pixel art”. en. In: *ACM SIGGRAPH 2011 papers*. Vancouver British Columbia Canada: ACM, July 2011, pp. 1–8. DOI: 10.1145/1964921.1964994.

[9] 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.

[10] Peng Lei, Shuchang Xu and Sanyuan Zhang. “An art-oriented pixelation method for cartoon images”. en. In: *The Visual Computer* 40.1 (Jan. 2024), pp. 27–39. DOI: 10.1007/s00371-022-02763-0.

[11] Ming Ronnier Luo. “CIELAB”. en. In: *Encyclopedia of Color Science and Technology*. Ed. by Ronnier Luo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 1–7. DOI: 10.1007/978-3-642-27851-8_11-1.

[12] Chongyang Ma, Li-Yi Wei and Xin Tong. “Discrete element textures”. en. In: *ACM SIGGRAPH 2011 papers*. Vancouver British Columbia Canada: ACM, July 2011, pp. 1–10. DOI: 10.1145/1964921.1964957.

[13] Marko Matusovic, Amal Dev Parakkat and Elmar Eisemann. “Interactive Depixelization of Pixel Art through Spring Simulation”. en. In: *Computer Graphics Forum* 42.2 (May 2023), pp. 51–60. DOI: 10.1111/cgf.14743.

[14] Pedro Medeiros. *Pixel Art Tutorials*. English. Blog. July 2020. URL: https://saint11.art/blog/pixel-art-tutorials/ (visited on 28/04/2025).

[15] Rodrigo D. Moreira, Flavio Coutinho and Luiz Chaimowicz. “Analysis and Compilation of Normal Map Generation Techniques for Pixel Art”. In: *2022 21st Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. Natal, Brazil: IEEE, Oct. 2022, pp. 1–6. DOI: 10.1109/SBGAMES56371.2022.9961116.

[16] onimille. *The big texturing tutorial*. English. Tumblr blog. June 2018. URL: https://onimille.tumblr.com/post/175820210378/the-big-texturing-tutorial (visited on 26/04/2025).

[17] Alexandrina Orzan et al. “Diffusion curves: a vector representation for smooth-shaded images”. en. In: *ACM Transactions on Graphics* 27.3 (Aug. 2008), pp. 1–8. DOI: 10.1145/1360612.1360691.

[18] Raymond Schlitter. *Pixelblog - 2 - Texture*. English. Blog. Feb. 2018. URL: https://www.slynyrd.com/blog/2018/2/15/pixelblog-2-texture (visited on 26/04/2025).

[19] Daniel Silber. *Pixel art for game developers*. eng. Boca Raton: CRC Press, 2016.

[20] 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.

A Parameters used to generate shown images

Key	Parameter	Reference
ρ	Density	Section 3.1
m	Margins	Section 3.1
Partial	Allow elements to be placed partially within bounds	Section 3.1
Pos. str.	Position strategy	Section 4
Col. str.	Color strategy	Section 3.2
Δ_{HSV}	HSV displacement	Section 3.2
C_{ex}	Excluded colors	Section 3.2
d_{max}	Maximum color distance	Section 3.2
El. col.	Per-pixel or region coloring	Section 3.4

Additionally, *(SB)* indicates shared border strategy selected and *(CD)* indicates color difference strategy selected.

Name	Trunk	Fish	Owl
Figure	1, 10, 11	2, 6, 8, 9	9
ρ	100%	100%	100%
m	$(-1, -1)$	$(0, 0)$	$(0, 0)$
Partial	Yes	Yes	No
Pos. str.	Random	Packed	Random
Col. str.	Auto (SB)	Auto (SB)	Auto (SB)
Δ_{HSV}	NA	NA	NA
C_{ex}	$(0, 0, 0)$	$(0, 0, 0)$	None
d_{max}	60	50	50
El. col.	Region	Region	Per-pixel

Name	Dog	Pine tree	Bird
Figure	9	9	9
ρ	100%	80%	100%
m	$(-2,-1)$	$(0,0)$	$(0,-1)$
Partial	No	No	No
Pos. str.	Random	Random	Random
Col. str.	Auto (CD)	Auto (SB)	Auto (SB)
Δ_{HSV}	NA	NA	NA
C_{ex}	$(0,0,0)$	None	$(0,0,0)$
d_{max}	50	50	50
El. col.	Per-pixel	Per-pixel	Region
Name	House	Foliage	Ruins
Figure	9	10	10
ρ	80%	60%	70%
m	$(-1,-1)$	$(0,0)$	$(1,0)$
Partial	No	No	Yes
Pos. str.	Packed	Random	Random
Col. str.	Auto (SB)	Auto (CD)	HSV shift
Δ_{HSV}	NA	NA	$(0,3\%,17\%)$
C_{ex}	$(0,0,0)$	None	NA
d_{max}	50	50	NA
El. col.	Region	Per-pixel	Per-pixel
Name	Ball	Cloud	Sword
Figure	12	13	13
ρ	50%	100%	100%
m	$(1,1)$	$(0,0)$	$(0,2)$
Partial	No	No	Yes
Pos. str.	Random	Random	Packed
Col. str.	In figure caption	Auto (SB)	Auto (SB)
Δ_{HSV}	NA	NA	NA
C_{ex}	None	None	None
d_{max}	NA	50	50
El. col.	Region	Region	Region

B List of LLM prompts

- What do brush strokes usually refer to in computer graphics or image processing?
- Are tensor fields the same as flow fields and vector fields? If not how do they differ from each other?
- This line of python cv2.bitwise_and(map, map, mask=mask) gives the error

```
cv2.error: OpenCV(4.11.0) :-1: error:
    ↪ (-5:Bad argument) in function
    ↪ 'bitwise\_and'
> Overload resolution failed:
>  - mask data type = bool is not
    ↪ supported
>  - Expected Ptr<cv::UMat> for
    ↪ argument 'mask'
```

  where mask is a numpy ndarray with shape (H, W, 1) and map is a opencv image with shape (H, W, 4)
- Help me deal with this OpenCV error

```
cv2.error: OpenCV(4.11.0) D:\a\opencv
    ↪ -python\opencv-python\opencv\
    ↪ modules\imgproc\src\drawing.
    ↪ cpp:2426: error: (-215:
    ↪ Assertion failed) p.
    ↪ checkVector(2, CV_32S) > 0 in
    ↪ function 'cv::fillPoly'
```

- Using Python, OpenCV and numpy, is there some built-in function to normalize a vector represented by the datatype cv2.Point?
- What is a python library that would enable me to easily solve a system of linear equations?
- How can I apply a mask to an image in opencv?
- I need to create a simple GUI for a Python program. It's just for prototype development, to make some things easier for me as a developer, so the GUI can be very simple and preferably very easy to code. Which framework/library should I use?
- Write code that draws a grid using opencv in Python, taking as parameters: the cell size in pixels, height and width of the grid
- Please write some lines of code that create an np.array of shape (16, 16, 2), where most value pairs in the 16x16 matrix are 0, but some are random normalized vectors
- Given that exlude is an array of colors in BGRA format, how can I filter out the colors in exclude from the colors in palette?

```
def extract_palette(img: cv2.Mat,
    ↪ exclude: np.ndarray = []):
    palette = np.unique(img.reshape
        ↪ (-1, img.shape[-1]), axis
        ↪ =0)

    not_excluded = ... # code here

    no_transparent = not_excluded[
        ↪ not_excluded[:, 3] > 0]
    return no_transparent
```

- Write a python function that takes an OpenCV image, scales it by a given factor, and places a red border around pixels whose coordinates are given.
- I have two numpy arrays, one with shape (4,) and another with shape (x, 4). How can I check if the first is one of the rows of the second?
- given a rect defined by

```
padded_y0, padded_y1 = y0 -
    ↪ pattern_padding, y1 +
    ↪ pattern_padding
padded_x0, padded_x1 = x0 -
    ↪ pattern_padding, x1 +
    ↪ pattern_padding
```

  write code that gets the 'x0' and 'y0' of each of its 8 neighbors

- Write python code using tkinter to create a GUI with these requirements: * Both code and GUI itself are simple and straightforward, as it is only a prototype * It allows selecting several parameters, which are defined below * It additionally has two buttons, one labeled ”Annotate” and another labeled ”Texture”. Leave room in the code for me to call a function when each button is pressed, but note that these functions will also call openCV’s cv2.imshow, so make sure the code is prepared to handle this

  The parameters are: * density = float between 0 and 1 * placement_mode = ”packed” or ”sampling” * allow_partly_in_mask = boolean * boundary_mask_padding = integer between -10 and 10 * element_padding = tuple of two integers, each between -5 and 5 * scale = integer between 1 and 32 * excluded_colors = list of colors, represented by an np.array of a form like ‘np.array([[0, 0, 0, 255], [255, 255, 255, 255]])‘ in BGRA space * color_map_mode = ”border”, ”hsv” or ”similarity” * element_color_mode = ”region” or ”per-pixel” * hsv_shift = tuple of three integers, each between 0 and 255 * max_attempts = positive integer * source_file = file path * boundary_file = file path or None * element_sheet_file = file path