Pixel Art Vectorization with Gradients

Kaldis Berzins[1]

Supervisor(s): Petr Kellnhofer[1]**, Mathijs Molenaar**[1]**, Elmar Eisemann**[1]

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

Abstract

Pixel art is an art style that uses low-resolution raster images with few colors. This style was ubiquitous in video games and computer programs in the late 20th century before high-resolution screens and powerful computers were widely available. In order to remaster these old games, or to use pixel art as an intermediate format to create vector images, conversion to vector formats such as Scalable Vector Graphics (SVG) is desirable. Existing tools have several limitations, notably a lack of support for gradients. We introduce a new method that expands upon an existing pixel art vectorization solution to add support for gradients, while maintaining compatibility with SVG and enabling real time editing for ease of use. Our method works by blurring parts of the image and then using various techniques to improve the quality of the output. We test our solution by creating example images with our tool's automatic generation capabilities and modifying them with the available controls. We conclude that our method is useful for imitating smooth lighting and real-world gradients, enabling artists to create more expressive vector versions of pixel art.

1 Introduction

The limitations of a medium often create art styles. Pixel artists use few colors in low-resolution images to create art that emulates the look of older video games and computer graphics. Despite these limitations they can create expressive curves and details. In contrast, vector artists manipulate mathematically defined curves and lines to create infinitely scalable cartoon-like images. They are limited by the geometric nature of their art; but, they still encapsulate the essence of real-world objects. Sometimes converting between these mediums might be desired. This could be because pixel art is often faster to create than vector art, so artists might want to make fast pixel art and then convert it. Game studios might want to remaster their old games by converting their pixel art to vector art. However, converting between these two representations is nontrivial, as an algorithm can not truly determine an artist's intent. A human might be able to.

A pixel art to vector art conversion tool has various desirable qualities. Firstly, user control. It gives the user the ability to interpret the artist's intent and pull information from the image an algorithm cannot deduce. Secondly, tools should produce output that represent the input well. Objects in the pixel art should persist in the vector art, and gradients should be well represented. Gradients offer artists more tools to express their ideas with, reduce the use of discrete color bands when representing shading, and offer a unique art style. Conversion tools should create Scalable Vector Graphics (SVG) output, as it is by far the most used vector graphics file format.

Existing solutions offer one or two of the qualities above, but all lack at least one. There are several vectorization tools for cartoon images and clip art that produce high-quality output with SVG support and gradients. However, they are not intended for low-resolution images and do not offer users the ability to edit their output. Other approaches offer good pixel art support, but require a custom program to be displayed, instead of exporting to SVG.

Our research question then is as follows: **Can an existing pixel art vectorization solution be retrofitted to include support for gradients, while retaining ease of user interaction and SVG support?**

In this paper we present a new method that has all three qualities. We extend the work of existing solutions to create a tool that can turn pixel art into vector art with the added option of creating gradients in the vector art. Our process automatically generates gradients based on heuristics, but allows for instant feedback when a user changes input parameters. The main contributions of our paper are:

- Method for creating artifact-free color gradient blurs for vector images.
- Automatic detection of where to create gradients.

2 Related Work

Converting pixel art images into vector formats is a relatively niche field. Image vectorization techniques are typically aimed at converting larger images, and produce varying results [7] when applied to small images. Kopf et al. [5] introduced a novel way to convert pixel art images into a vector representation using a graph of connected pixels and fitting Bézier curves to the graph. This idea was taken further by Matusovic et al. [7] who uses the same graph representation but fits curves using a spring system. This second approach allows users to choose how pixels connect to each other and how the curves should be fit to the pixels. It has the additional benefit of supporting the Scalable Vector Graphics(SVG) format [8], which is widely used to store vector images.

However, the approach from Matusovic et al. does not support gradients. The method introduced by Kopf et al. supports gradients, but only by implementing a custom renderer that samples the underlying pixels and then blends their colors based on distance. Another similar system by Orzan et al. [10] uses diffusion curves, which were considered [1] for inclusion in the SVG specification, but were ultimately excluded from the final format. The diffusion curve approach thus requires a custom renderer and does not convert to SVG trivially. Both approaches also do not support user input, and only allow the user to edit the resulting vector image.

An additional paper by Du et al. [4] describes a vectorization method that is compatible with SVG, but which is not intended for pixel art. The method decomposes the raster image into multiple layers of linear gradients that form an approximation of the image when overlaid. These gradients are simple enough to be compatible with the SVG format, and so a comparable approach is desirable. However, the approach works best for images that were originally vector images that were rasterized, and significant improvements are required to make the approach work for hand-drawn pixel art images.

3 Method for Creating Gradients

This chapter explains our method for adding gradients to vectorized pixel art images. An overview of our method is pro-

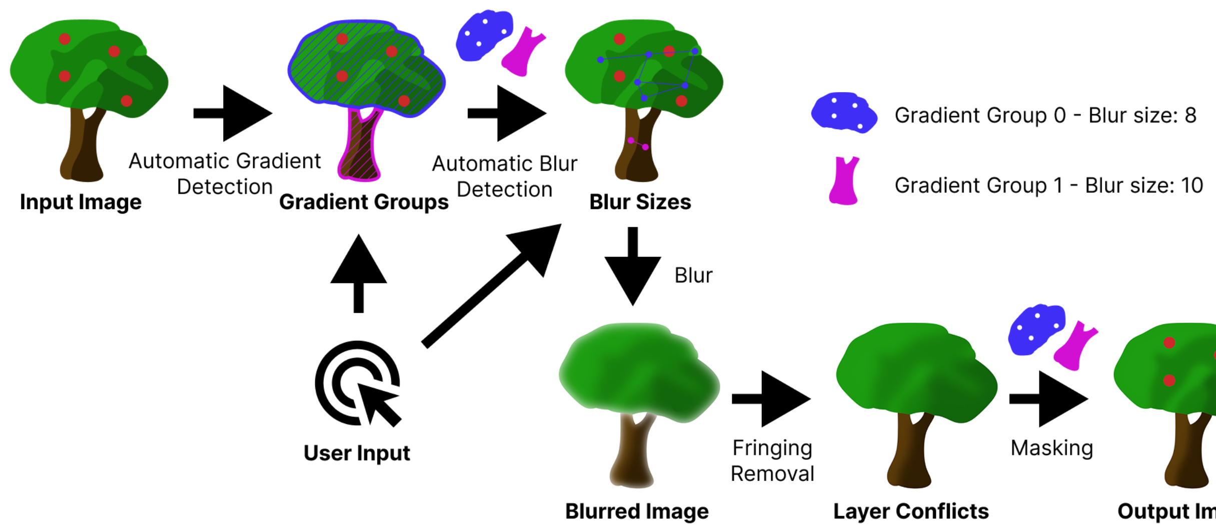

Figure 1: Full pipeline of our method. Arrows represent steps used in our method, with intermediate results shown. The magenta and blue shapes represent gradient groups.

vided in Figure 1. The input to our method is an SVG image generated by the tool created by Matusovic et al. [7]. Applications of our method for other SVG inputs are discussed in section 5.

3.1 Definitions

This subsection describes terms used in our method explanation. SVG-specific definitions are given first.

Objects are the basic building blocks of an SVG image and represent individual shapes with a single color, linear gradient or radial gradient. Objects are placed according to their order in the SVG file, with the last object appearing at the top of the image.

Filters are a set of post-processing operations that allow for adding different effects to SVG images, such as blurring, adjusting colors, and others. When rendering SVG images, objects are first rasterized, then the pixel data is passed to the filter which then performs filter operations on each pixel of the filter region.

Filter regions are areas on which SVG filters apply. By default [2], the filter region is 10% larger than the object in each axis. For parts inside the region, but outside the object, the filter is provided with values of zero for red, green, blue and alpha (transparency) channels.

Masks are SVG primitives that allow for restricting the rendering of an object to a certain area. They contain one or more areas outside of which the object is not rendered. Masks are applied after filters; they cannot be used to alter the inputs to a filter.

The following terms are specific to our method.

Gradient groups are a set of objects that are to be blurred together. An image may have multiple gradient groups.

Adjacency graphs are graphs where each node is an object, and edges are placed between adjacent objects.

Fringing is the soft border around blurred objects that is the result of a smooth alpha value transition over the edge of the object.

3.2 Automatic Gradient Detection

In order to determine where to place gradients, we created an automatic gradient detection algorithm. In short, we create gradients where adjacent objects have similar colors. Specifics of this method are described in this section.

First, adjacency is determined by finding the union of a pair of objects. The union of two objects is the area that either of the two objects occupies. If the union of two objects is continuous (one path completely encompasses the area) then these objects are adjacent. This complicated method of determining adjacency is used because objects that are adjacent may overlap or not share points on their border, making other tests for adjacency unreliable. Objects in the image form an adjacency graph.

After creating the graph, we remove edges that are not similar in color. We test for similarity by comparing the hue, saturation and brightness of the two objects. Likely candidates for a gradient in pixel art are places where the artist intended to shade the image to show how light interacts with the objects. These places will have a similar hue, but differ in saturation and brightness. To find these edges between objects we check if the hue of the two objects is within a narrow range and then whether the brightness and saturation fall within a wide range:

$$\text{similar} = \begin{cases} \text{true} & \text{if } \Delta H < 40 \wedge \Delta S < 0.3 \wedge \Delta B < 0.3 \\ \text{false} & \text{otherwise} \end{cases}$$

Where ΔH is the difference in hue ($0 < H < 360$) between the two objects, ΔS is the difference in saturation ($0 < S <$

1) and ΔB is the difference in brightness ($0 < B < 1$). The values provided here are values we found worked best with our example images.

After removing edges that fail the check, we are left with multiple disconnected components. Each component of the disconnected graph forms a gradient group. The objects in the components are placed into SVG groups for the later stages of the algorithm. The resulting graph can be seen in Figure 1 in the image labeled "Blur Sizes".

3.3 Blurring

To create the impression of a gradient we use a Gaussian blur. It produces results that are superficially similar to linear and radial gradients (which are available for SVG images), but with different transition functions. Linear and radial gradients have linear transitions between colors, but a Gaussian blur has a transition shaped like a Gauss function. Blurring has the additional advantage of working with the arbitrarily shaped Bezier curves that make up an SVG image. A comparison with linear and radial gradients can be seen in Figure 2.

Blurring is the process of averaging pixels in an area (called a kernel) around a given pixel. Pixels in the kernel are given weights that dictate how much they contribute to the final average pixel. The size of the kernel is called the **blur radius**. A kernel with weights distributed like a 2-dimensional Gauss function is called a Gaussian blur. When using a blur to imitate a gradient, as the pixel gets closer to one side, more and more of the kernel is occupied by one of the colors. This leads to the color of the pixel smoothly shifting from one color to the other.

When we initially generate gradient groups, we give each group a blur radius. This radius is computed as the average distance between the centroids ($\vec{C}$) of neighboring objects.

$$\vec{C} = \frac{1}{n} \sum_{i=1}^{n} \vec{v}_i$$

$\vec{v}_i$: the points that comprise each shape
n: number of points in a shape

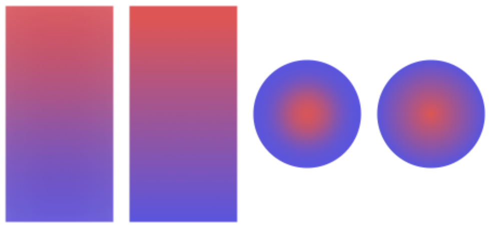

Figure 2: Comparison of blur to linear and radial gradients. In each pair left is blur and right is gradient

3.4 Fringing Removal

Fringing is undesirable because objects in the image occupy areas outside their original area, making the effect look like a blur and not a gradient. It is caused by the way filters are implemented in the SVG specification and the nature of blurring. We remove it by only blurring the red, green and blue channels of the image, while leaving the alpha channel intact.

Fringing happens because the parts of the filter outside the object are also blurred, giving them partial color from the object. For these pixels, part of the kernel falls outside of the object, lowering their alpha value and therefore making them translucent. Pixels inside the object also have some of their kernel fall outside and thus their alpha is also lowered. This creates a soft edge, where the background can be seen through the blur.

We solved this problem by blurring the color values (red, green and blue channels) and keeping the alpha channel from the original object. This creates a hard border around the object, but keeps the soft edges between colors, imitating gradients. This process is illustrated in Figure 3.

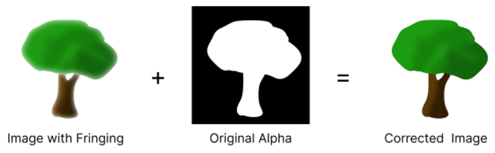

Figure 3: Diagram showing the fringing removal process. For alpha, white represents an alpha value of 1 and black represents 0

3.5 Masking

In Figure 1 the image labeled "Layer Conflicts" lacks the apples that exist in the input image. This section explains how such issues are solved with masking.

In the case of the apples, the layer conflict can be solved by moving the gradient group below the red circles. However, this approach does not generalize. In the case of Figure 4, we are trying to blur the red and blue circles together. If we were to group these objects together we could only place the black rectangle above or below the group. To maintain the appearance of the input image, we generate a mask (middle image) that has a hole through which the black rectangle is visible. The result can be seen on the right.

We create the mask as follows. First, we find the bottom object in the gradient group. Then, we iterate over every object above the bottom object going upwards. For each object that is in the gradient group, we add the areas to the mask. For each object that is not in the gradient group we subtract that area from the mask. The resulting mask is a union of all the blurred objects with holes for every object above. In the case of Figure 4 the resulting mask can be seen in yellow in the middle. The whole group is then placed on the top layer of the SVG image.

4 Results

This section describes how we created the example images in Figure 5 and what each image serves to demonstrate about the vector gradient creation process.

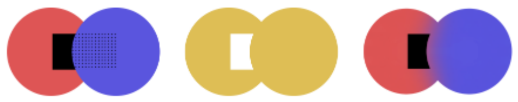

Figure 4: Diagram demonstrating masking. Left is the object configuration where the black dotted region is below the blue object. Middle is the resulting mask. Right is the resulting blurred object.

4.1 Implementation Details

We wrote the tool that adds gradients to the existing vectorization algorithm in Javascript using Svelte [3] to create a webserver and Paper.js [6] to manipulate SVG files. The system runs in real time and upon loading a vectorized file presents a set of auto-generated gradient groups. It then lets the user create and edit gradient groups and change their blur radius. It also exports to SVG, only using features that are supported by most SVG renderers.

The user interface for the gradient generator is presented in Figure 6. The left panel of the interface is the gradient editor. Using it, the user can select which objects are put into gradient groups, which are listed in the middle panel. The slider for each gradient group controls the blur radius to control the ”smoothness” of a gradient. Pressing the button labeled ”Generate Gradients” updates the image on the right to reflect editor changes in the output.

4.2 Image Creation Process

We created the images by first drawing the pixel art (first column of Figure 5). For this, we used various techniques, levels of detail, color amounts and resolutions to test different aspects of the gradient creation method. We then vectorized each image using the tool made by Matusovic et al. [7], utilizing the full extent of its manual correction capabilities (second column). Without changing the default output of the automatic gradient group generator (described in subsection 3.2), we created the third column of images and then adjusted the gradient groups manually to create the images for the final column.

4.3 Image Descriptions

The rainbow image (a) shown in Figure 5 demonstrates the method’s ability to blend an arbitrary amount of colors together in an analogous way to linear gradients with multiple color stops. The automatic gradient detection does not form a gradient between all the colors of the rainbow (seen on the third row), as the boundary between blue and green exceeded the color difference threshold, however, we manually corrected this for the fourth image.

The image of a conical flask (b) demonstrates a simple gradient between colors of a similar hue as well as the ability of the gradient generator to correctly mask the gradient such that it does not obscure the white shine on the flask. The final image is unchanged from the auto-generated result.

The watermelon (c) example shows how hard edges can be created between gradients and their surrounding objects.

The orange (d) image is an example of how discrete shading or ”banding” can be turned into smooth shading with our method. We lowered the blur amount slightly for the final image to add shine to the orange.

The tree (e) demonstrates how our method handles noise to create the appearance of depth. The blur in the automatically generated image is too high to show any detail in the leaves, but we lowered it to reveal the depth created by the shading.

The beer bottle (f) demonstrates how both diffuse and specular lighting can be imitated with our method. The automatic generator detected the diffuse case and we added the gradient to make the glass of the bottle look shiny.

The skull image (g) shows how complex diffuse lighting patterns are handled by our method. The automatic generator set the blur too high, but lowering it maintains the same lighting patterns as in the original pixel art.

The deer (h) example tests the suitability of our method for use in animal fur patterns. Once again, the blur is set too high automatically, but when we manually correct it, the light spots on the deer are visible and smooth.

The glass (i) image shows our method’s ability to indicate transparency. The shine is not detected because of the high contrast between blue and white, but with manual correction we achieve an illusion of depth and transparency.

The sword image (j) is a demonstration of how the effect can be used in a way that does not necessarily imitate real-life lighting, but rather creates a glow effect on the green and white parts of the sword. The automatically generated image depicts a regular shine on the metal part of the sword, but we instead chose to modify the internal part of the sword to make the glow effect.

5 Discussion

In this section we interpret the results presented in the previous section and present strengths and limitations for our method, as well as how the method could be improved in the future.

5.1 Strengths

Our method represents smooth lighting well. Glows, shines and shadows are no longer discrete bands of color, but rather smooth transitions between colors. Examples (d), (f), (g), and (i) from Figure 5 show this best. In these examples the real world object that the images represent is smooth, so when soft light hits them the color transition from light to dark is gradual. Pixel art typically limits the number of colors used and so it can only approximate the smooth transition. We add smooth lighting as another option for the artist to choose.

Another strength of our approach is in representing real life gradients. This is shown in examples (a), (c) and (h). It handles transitions between different hues well as the rainbow example demonstrates. There are no additional hue shift artifacts over those expected of gradients in the RGB color space. The rainbow looks like a rainbow, and the rind of the watermelon looks like a watermelon rind.

An additional strength of our approach is how easy it is to use. Instead of having to run the program again from the start when the output does not match the artist’s desires, the program immediately shows how the output changes when the

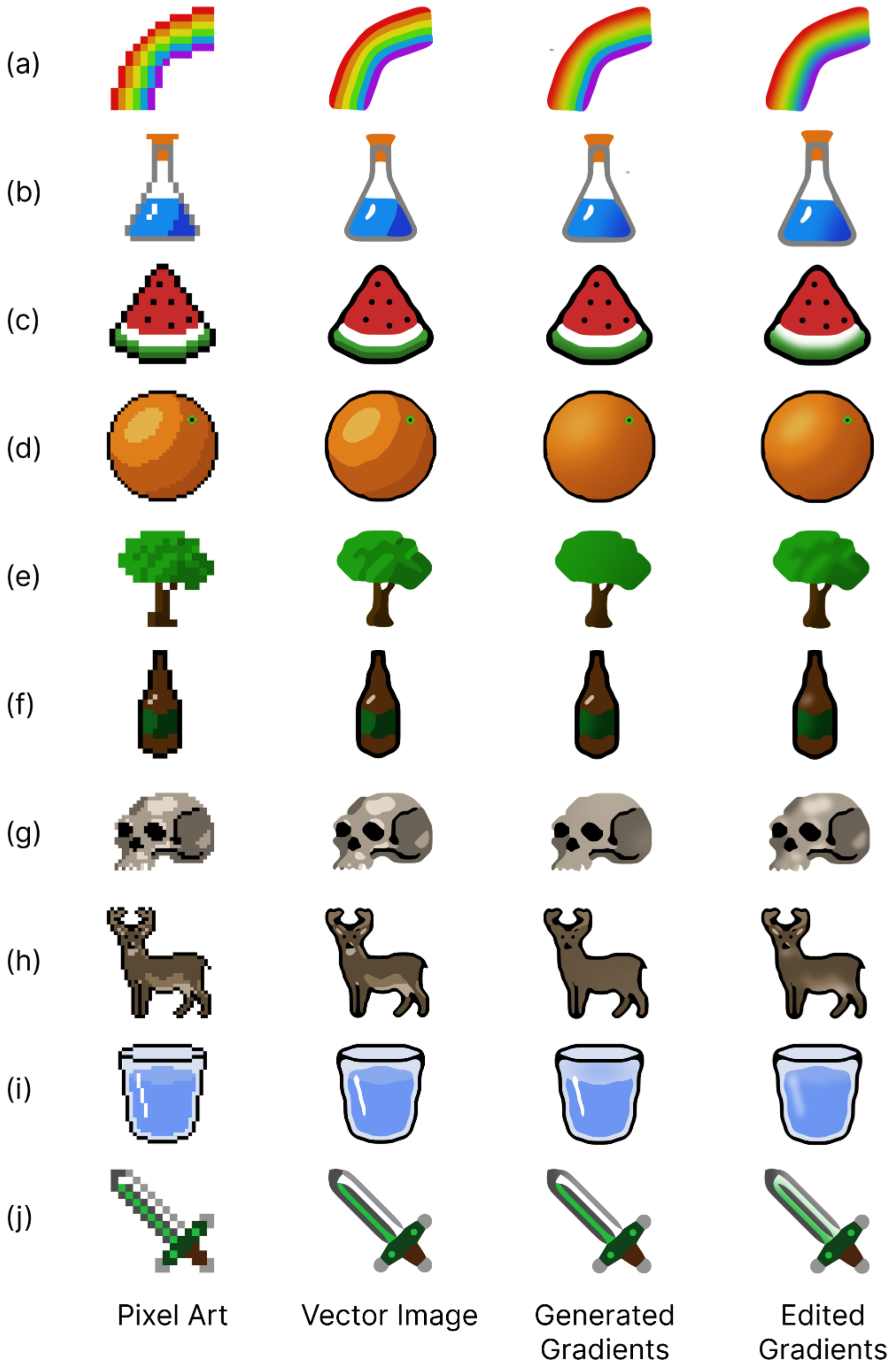

Figure 5: Left to right in each row: Original pixel art image, vectorized image, auto-generated vectorized image with gradients, user created vectorized image with gradients. All images are drawn and edited by the author.

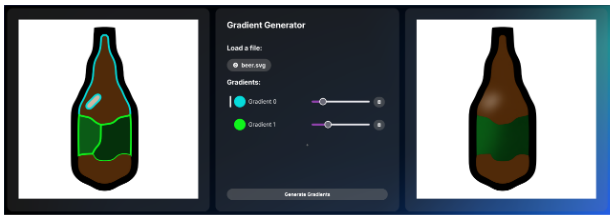

Figure 6: Screenshot of our application. The left section is the gradient editor. The middle section is the control panel for the editor. The right section is the resulting SVG image.

input parameters are altered. The interface is also simple and intuitive with a color coded gradient group selection pane, and a single slider to control the size of the gradient. When loading a file the automatically generated gradients give the user a starting point from which they can quickly get to the result they desire. The program also does not require any technical knowledge to use, because SVG specification specifics and filter settings are abstracted away into a simple selection pane and a few sliders.

5.2 Limitations

A weak point of our current method is the automatic gradient group generation. In example images (a), (c), (f), and (i) the generator produced suboptimal results. For example, the rainbow was not entirely added to the gradient group. A universal threshold for what makes two colors similar will not always create the desired groups. Moreover, a simple color comparison may not even be what should determine whether colors should be grouped. There are many context clues that make human vision perceive a gradient. We think that to improve the grouping algorithm there should at least be an additional heuristic to detect specular lighting in images, as it otherwise works well with our method.

Similarly, blur size detection produces results that closely match our manual version in many cases ((a), (b), (d), (f), (j)) but otherwise creates varying results. Our method works well when there are few objects that are arranged with similar distance to each other's centroids. It fails when the objects in the image are chaotically arranged or there are a few objects that are much larger or farther away than others, as these heavily skew the average distance between centroids of objects that the blur size is based on.

Both of the above limitations are mitigated by the ease of use of the program. If the user desires a different output, they can change the parameters for the gradient generator with a few inputs.

5.3 Future Work

Major improvements to our method could come in the form of improved heuristics for gradient group generation. We think it fails in scenarios where objects are chaotically spaced or in scenarios where a shine effect is desired. One possible improvement to the blur radius problem could be to instead measure the size of an object in the direction of other centroids in a group as a metric for blur size. This would make blurs that completely erase an object less likely as the size of the blur could be constrained to the size of the object, not just the distance to other objects' centroids. Shines could be detected by looking for small light objects over darker backgrounds, however this could present its own set of problems, for example for stars in the night sky.

A second major improvement to the existing method would be the ability to individually control the blur over every boundary between objects. Example (b) in Figure 5 demonstrates this well. The white specular reflection on the flask could be blurred for a better visual shine effect, but unfortunately we already grouped the blue object below it with the dark blue object to create a gradient in the water. If we were to add the shine to that blur group, the blur would be far too large and completely erase the shine. With individual blurs on every boundary we could control the blur on the white object and the blue objects individually. We can imagine a scenario where blur groups are components of a disconnected graph of objects where every edge in the graph has a separate blur radius. We attempted various techniques involving compositing of alpha channels but were unable to realize this idea in time. We believe it is possible to achieve with just SVG primitives.

Our method could also be expanded to process arbitrary SVG images, for example from cartoons or other similar sources. Currently, while it does accept SVG, our method makes several assumptions about the input it is provided. The most important of these is it assumes that all objects have a single color without gradients or other SVG attributes. Expanding our tool to encompass these elements would allow for more use cases.

6 Responsible Research

6.1 Integrity

We used OpenAI's ChatGPT[9] occasionally during this project to assist with reading documentation and to better understand SVG, Paper.js and Svelte. For transparency, we have included a list of prompts used for this project in Appendix A. No text for the paper was written using Large Language Models. We also recognize that copyright can be an issue for images, so we decided to draw every example image and figure ourselves.

6.2 Reproducibility

We believe that reproducibility is a vital part of science. Therefore we have taken several measures to make sure that our results are reproducible. First, we have published our project in a freely available Github repository[1]. Second, we have made our method deterministic, so that a set of inputs always generates the same outputs, allowing for easy reproduction. Lastly, we have provided a set of example images that can be tested to verify that they produce the same results for other researchers using our tool.

6.3 Impact on Artists

Like any tool that automates some part of making art, we acknowledge that our tool could be used to replace artists. However, we would like to stress that our tool is intended for use by artists to improve their workflow and reduce the need to

[1]https://github.com/kaldis-berzins/gradients_reloaded

create every part of a piece of vector art from scratch. Therefore we believe that artists are not excluded by our tool, but rather their ability to create is enhanced.

7 Conclusion

This paper described a new method for creating SVG images from pixel art with gradients. We achieve this by blurring regions of the image and then using various techniques to modify the blur to prevent unwanted artifacts. Our results indicate that our method is applicable to various scenarios, especially when representing soft lighting or when imitating real world gradients. Our solution is user friendly and does not require advanced knowledge to use. We believe that the answer to the question posed in our introduction to be a firm ”yes”.

References

[1] Canon. Proposals/advanced gradients. https://www.w3.org/Graphics/SVG/WG/wiki/Proposals/Advanced_Gradients, 2014.

[2] World Wide Web Consortium. Filter effects region. https://www.w3.org/TR/SVGFilter12/#FilterEffectsRegion.

[3] Svelte contributors. Svelte: web development for the rest of us. https://svelte.dev/.

[4] Zheng-Jun Du, Liang-Fu Kang, Jianchao Tan, Yotam Gingold, and Kun Xu. Image vectorization and editing via linear gradient layer decomposition. *ACM Transactions on Graphics (TOG)*, 42(4), August 2023.

[5] Johannes Kopf and Dani Lischinski. Depixelizing pixel art. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011)*, 30(4):99:1 – 99:8, 2011.

[6] Jürg Lehni and Jonathan Puckey. Paper.js. http://paperjs.org/.

[7] Marko Matusovic, Amal Dev Parakkat, and Elmar Eisemann. Interactive depixelization of pixel art through spring simulation. *Computer Graphics Forum*, 42:51–60, 2023.

[8] Mozilla Developer Network. Svg: Scalable vector graphics. https://developer.mozilla.org/en-US/docs/Web/SVG.

[9] OpenAI. Chatgpt. https://chatgpt.com/.

[10] Alexandrina Orzan, Adrien Bousseau, Holger Winnemöller, Pascal Barla, Joëlle Thollot, and David Salesin. Diffusion Curves: A Vector Representation for Smooth-Shaded Images. *ACM Transactions on Graphics*, 27(3):92:1–8, August 2008.

A Use of Artificial Intelligence

We used ChatGPT[9] during this project. A full list of prompts used for the research project can be found in the following subsections.

A.1 Prompts about SVG

- ”I have 3 svg shapes, with the side two overlapping the middle shape. I want to blur each size shape with the middle one, but with different blur radii for each. Is this possible?”
- ”How would I blend two svg groups together using the screen blend mode?”
- ”What are the different types of inputs for SVG filters? I have seen SourceGraphic and BackgroundImage. What does each one do?”
- ”Is there a way to make an svg blur that only blurs the internals of the image, not creating a gradient of alpha around the edges?”
- ”how do I wrap an svg path with id #mask-path in mask tags using javascript?”
- ”I want to perform operations on SVG objects using javascrpt, like unioning paths, finding if objects are encompassed by others, applying blurs, etc. What library is best for this?”

A.2 Prompts about Paper.js

- ”In Paper.js how do I get the tangent line of one of the points?”
- ”How do I create a copy of an item in Paper.js for modification before svg export?”

A.3 Prompts about Svelte

- ”Using svelte 5, what is the idiomatic way of making a TODO list where pressing x on the list item removes it from the array?”
- ”How do I add a global CSS file in sveltekit?”