Interactive Depixelization of Pixel Art through Spring Simulation

Marko Matusovic, Amal Dev Parakkat, Elmar Eisemann

Interactive Depixelization of Pixel Art through Spring Simulation

Marko Matusovic[1], Amal Dev Parakkat[2], Elmar Eisemann[1]

[1]Delft University of Technology, Netherlands
[2]LTCI - Telecom Paris, Institut Polytechnique de Paris, France

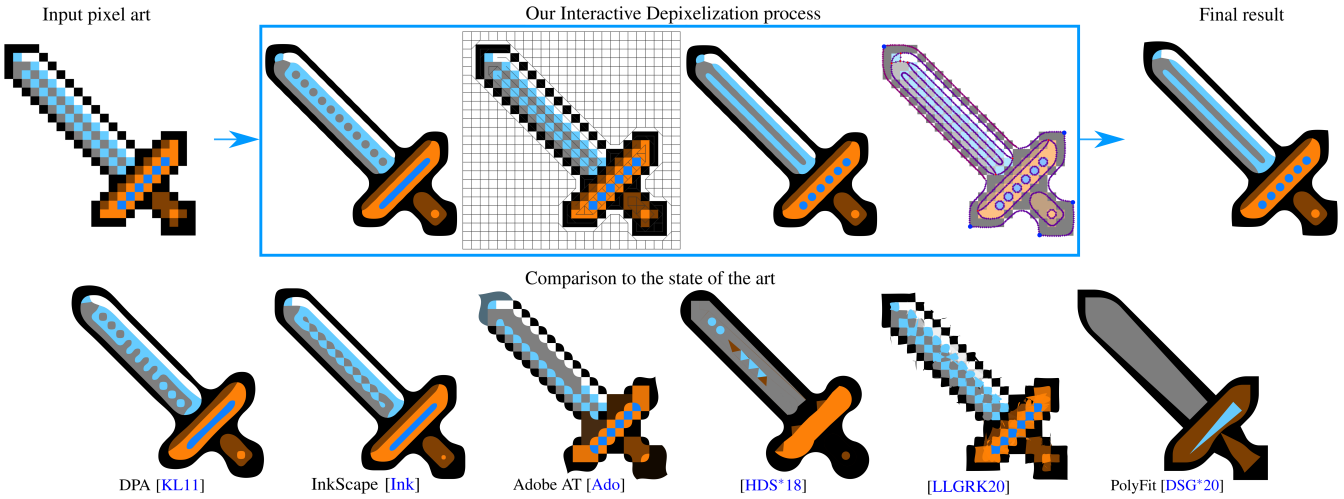

Figure 1: *Top row - Our interactive process (Left to Right): Input pixel art can be automatically vectorized by our method. Yet, the original pixel art allows for different interpretations. Our solution enables users to interactively disambiguate connectivity between pixels. Finally, the user can annotate sharp corners (blue points) and smooth connections (pink nodes) [note that these operations require only a single mouse click each] to optimize the final output. Bottom row - Comparison to the state of the art (Left to Right): Results of Depixelizing Pixel Art (DPA) [KL11], InkScape [Ink], Adobe AutoTrace [Ado], [HDS*18], [LLGRK20] and PolyFit [DSG*20]*

Abstract

We introduce an approach for converting pixel art into high-quality vector images. While much progress has been made on automatic conversion, there is an inherent ambiguity in pixel art, which can lead to a mismatch with the artist's original intent. Further, there is room for incorporating aesthetic preferences during the conversion. In consequence, this work introduces an interactive framework to enable users to guide the conversion process towards high-quality vector illustrations. A key idea of the method is to cast the conversion process into a spring-system optimization that can be influenced by the user. Hereby, it is possible to resolve various ambiguities that cannot be handled by an automatic algorithm. ***Author's copy - Not for redistribution.***

CCS Concepts

• ***Applied computing*** → ***Fine arts;*** • ***Computing methodologies*** → ***Image manipulation; Shape modeling;***

1. Introduction

Images are usually stored in raster format (a two-dimensional grid of pixels). At low resolution, pixels are visible, an artefact known from early video games (the first sprites appeared in the 1970s [Wir]). Nowadays, such "pixel art" has gained significant popularity. One benefit of pixel art is that it can be produced quickly and only with a few resources [Sil15].

Vector images are composed of mathematical curves. These are resolution-independent and lead to sharp and crisp shapes. This is a particularly interesting property given that most graphical output is shown on various display modalities, including high-resolution

screens, large projections or cell phones. However, creating vector images requires skill and effort.

Various industry experts and researchers have attempted to bridge this gap and create vector images from pixel art. They typically target an automatic conversion, which seems beneficial at first but such solutions cannot resolve all ambiguities that are inherent to the simplicity of pixel art. We show that a user can steer the process very easily to produce high-quality vector illustrations from very coarse pixel art.

The applications for our solution are manifold. It is possible to generate high-quality vector art for graphical illustrations or to convert sprite sheets of retro games to be used in a remastered implementation. The generation of clipart or editable representations are additional options that are available to artists, making our solution a useful addition to their toolbox.

Our method relies on a user-controllable spring system that is pivotal to the conversion process. Initially, our method clusters pixels based on their color. Next, nodes are placed along the boundary of these clusters and connected to yield a network. Subsequently, the connections in the network are interpreted as springs and a simulation process drives the network away from the initial pixel art to yield a smooth vector representation. Finally, after an optimization step, layers are extracted to generate the final high-quality illustration. As shown in Figure Figure 1, having the user in the loop to influence the conversion process is crucial for low-resolution pixel art. Here, the final image was obtained by influencing the spring forces, avoiding a blobby appearance and shrunken details.

Our solution produces comparable results to existing approaches automatically but then allows for user interaction to define or rectify the relationship between pixels and vector illustration. Hereby, sharp features or local smoothness of the final curve representation can be matched to the user's interpretation.

In summary, we make two contributions:

- A custom-made spring system for depixelizing pixel art.
- A simple interactive framework to modify the vectorized results (in terms of topology and shape) to match the user's interpretation.

2. Related Works

Transforming discrete samples into a smooth representation is, in fact, a problem in many domains (e.g., voxels to a surface [Fri22]). Further, specialized vectorization approaches exist for meshes [EWHS08] or isosurfaces [SEH08]. Nevertheless, most vector conversion addresses image content, which is also the focus of our method.

The related work for image content can be classified into two categories: upsampling (techniques that increase the resolution of the image/pixel art) and vectorization (techniques to obtain a vector output from a raster image).

Image upsampling or super-resolution techniques in their simplest form focus on interpolation schemes [Key81, FMS98, TM96], which were later replaced by more sophisticated techniques [Fat07, TC04]. A large leap in quality was enabled through learning-based methods. Some techniques rely on internal similarities [CCS*14, FF11] or involve priors [WPM*18, DLHT14]. It is not possible to cover all approaches in this paper but a plethora of works can be found in recent surveys [AKB20, WCH21].

Vectorization leads to a set of vector primitives that can even be effective for representing natural images (via curves [OBB*13, JCW09], gradient meshes [LHM09], or regions [LL06]). Besides natural images, inputs can be digital illustrations, such as cartoons, clipart, and logos [YCZ*16] or textures [RL16]. The first step is typically to create a polygonal representation of the input shape and then replace it with a set of best-fitting vector curves. Research in this line of work mainly concentrates on finding a good polygonal approximation [SBZ05, ZCZ*09], or a good vector approximation [YCZ*16, DSG*20]. Recently researchers have been looking at integrating machine learning to create such vector illustrations automatically [RGLM21, VPB*22]. Due to the extreme downsampling, handling pixel art requires specialized solutions. Interestingly, the same holds for converting images into pixel art, which is not equivalent to downsampling and represents a very challenging problem [IVK13, GDA*13]. In this work, we are interested in the conversion of pixel to vector art. Motivated by the large amount of available pixel art derived from games, the topic received a lot of attention in online communities. As a consequence, many conversion strategies or new image upscaling techniques [SB19] have been made available as (open-source) software implementations [hqx, eag, sca] or commercial tools for clip art vectorization, e.g., Adobe Illustrator, Autotrace, Coreldraw, Potrace, and Vector Magic.

In the research community, Kopf and Lischinski [KL11] introduced a fundamental method. It is based on perceptually-motivated rules to disambiguate pixels and group them accordingly, which provided a significant improvement in terms of quality. Splines are fitted on these pixel groups to create the final vector output. Though the algorithm was widely accepted, it had the inherent problem of not being able to capture sharp features. Using machine/deep learning, researchers introduced new methods to rely on perceptual aspects to identify such sharp corners [HDS*18, DSG*20].

It has to be noted all previous methods rely on an automatic conversion - which means that a user does not have control over the conversion process, which we show is crucial for resolving certain ambiguities in the input. Hereby, we make it possible for a user to integrate their own interpretation of often ambiguous pixel artwork.

3. Algorithm

The proposed method has four steps:

- Clustering: We extract a boundary representation of the pixel art by grouping similar pixels.
- Path Generation: Along the pixel-cluster boundaries, nodes are placed and connected to their direct neighbors.
- Spring simulation: The network node positions are optimized according to an energy formulation to define the final shapes.
- Optimization: Optimal Bezier curves are fitted, adjusted, and grouped to create easily editable vector graphics.

To enable control over the conversion, we will offer a user the possibility to intervene in the clustering and simulation, which we will describe along with the algorithmic steps of the method.

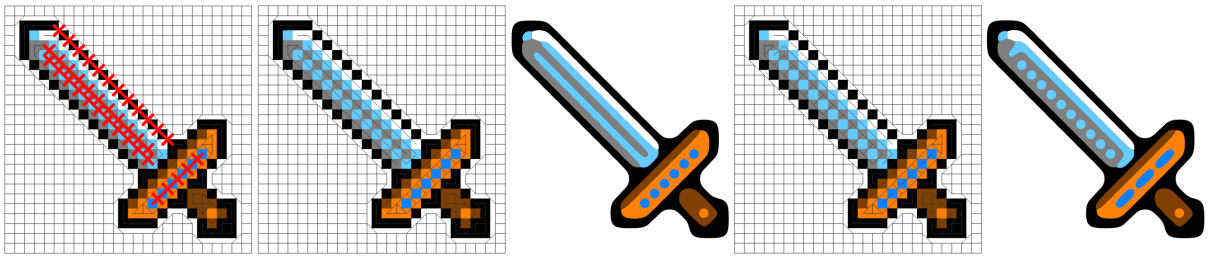

Figure 2: *Effect of manual disambiguation of edges. Left to right: Curve network with red crosses representing ambiguous connections, network after manual disambiguation, and the corresponding result*

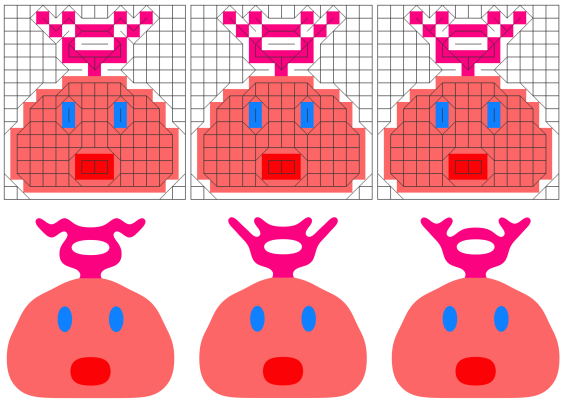

Figure 3: *Different results (bottom) from the same pixel art by altering the connections (top)*

Figure 4: *Local effect of connections on the vectorization*

3.1. Neighbor Clustering

We want to vectorize the boundaries of pixel areas that describe a shape. First, we, therefore, need to estimate, which pixels should be grouped. We involve a similarity graph, where we add a vertex for each pixel. Edges are then established between adjacent pixels, where adjacent is the 1-ring neighborhood of a pixel, based on whether the color distance is below a given threshold. We found a Euclidean distance in the RGB color space to be sufficient.

The resulting graph reveals the ambiguities that are inherent to pixel art; Figure 2 shows this problem, where red crosses indicate such places, which can be adjusted by the user to achieve different results. It is possible to add/remove connections to indicate wanted relations between pixels. Hereby, multiple outputs can be obtained from the same input (Figure 3) by influencing the local interpretation of the pixel art (Figure 4). To limit the necessary user interaction, our solution follows an automatic scheme as follows by default:

After creating the similarity graph, for each diagonal edge, we check whether it is having an intersection with its anti-diagonal counterpart. All such edges that cause an intersection are resolved. In contrast to using multiple heuristics as in DPA [KL11], our solution uses a single rule, powerful enough to give satisfactory results and resolve conflicts as follows: Let edge (a,b) and (c,d) be the intersecting edges, we removed those two edges and compute the cardinality of the vertices connected to all the four vertices a,b,c and d. By assuming that it is always better not to leave sparse pixels, which create small disconnected fragments, we place back the edge containing the vertex with the least cardinality. In other words, let the decomposed graph be G_D, the updated graph be G_{new}, and the cardinality of the subgraph in G_D containing x be $|C_x|$, then:

$$G_{new} = G_D + \begin{cases} (a,b), \text{if } |C_a| \text{ or } |C_b| \text{ is the least} \\ (c,d), \text{if } |C_c| \text{ or } |C_d| \text{ is the least} \end{cases}$$

If both edges have vertices with the same cardinality, we arbitrarily keep one of them - the user can always alter this choice if required. The entire process is shown in Figure 5.

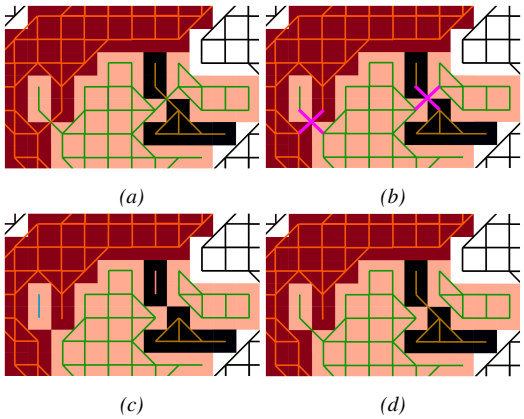

Figure 5: *(a) Graph created based on similarity, (b) Ambiguous edges marked in pink color, (c) Decomposed graph, (d) Final result after disambiguation (colors are used to ease understanding)*

This simple way of addressing ambiguities is indeed often leading to satisfactory results - some are shown in Figure 6. Yet, some cases require additional input to reflect the artist's intent, for example, the connection between the eye and the eyebrow of the girl in Figure 7.

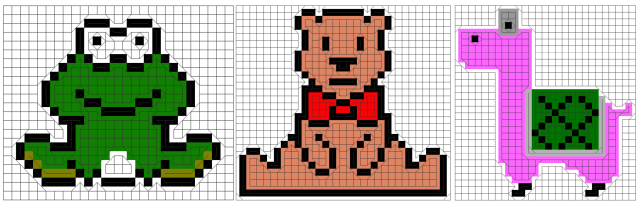

Figure 6: *A few results of our automatic disambiguation*

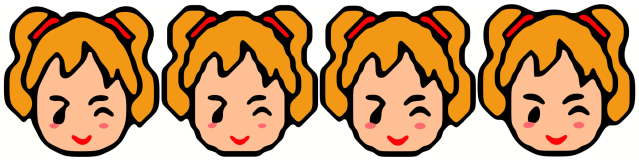

Figure 7: *Left to Right: Automatic results generated by Inkscape, Adobe AutoTrace, and Our method, Interactively disambiguated result using our method*

This holds for several automatic systems, as illustrated, but ours offers the freedom to edit connections directly.

3.2. Path Generation

In the next step of the algorithm, we will generate a path around our pixel clusters by adding nodes along the graph edges, whose positions will be optimized to derive the vector art later on.

First, each cluster receives a representative color (average of all its pixels). The cluster's boundary pixels are identified by comparing the representative color of each cluster pixel to the adjacent pixel colors (horizontally and vertically). If they differ, the border between them is considered a boundary.

Figure 8: *Corner and Edge nodes*

We create a chain of nodes that define paths along the cluster boundaries. We distinguish two types of nodes: corner nodes (at pixel corners) and edge nodes (along pixel edges) (Figure 8 - corner/edge nodes are red/blue). For all our experiments, we fixed the number of edge nodes per pixel edge as three.

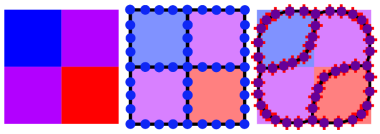

Figure 9: *An example of a diagonal link at a corner node*

Please note that this process leads to duplicated nodes - the same nodes are created for both sides of a boundary. While we fuse edge nodes immediately, corner nodes are processed later, considering the following two cases:

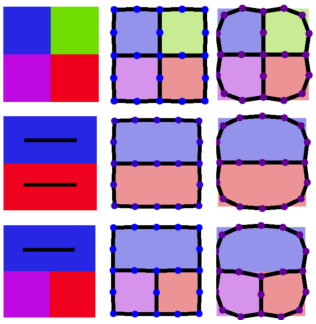

Figure 10: *A few standard cases for corner nodes.*

Diagonal links: If at a corner node, an edge in the similarity graph between the four surrounding pixels connects two adjacent pixels diagonally, it means that they belong to a common shape; they should not be separated by a border or path. Hence, we produce two corner nodes, which reflect the separation by the common shape. One corner node retains the edge-node connections on one side of the common shape, and the other corner node the rest. These two nodes will be pulled apart in the next step to establish the indicated connectivity (see Figure 9).
Standard corners: In all other cases, we know that the adjacent clusters meet in this one corner location. Thus, we fuse all corner nodes and maintain the edge-node connections (Figure 10).

3.3. Spring Simulation

After the previous step, the nodes form a network, whose closed paths surround the pixel clusters. These paths and the cluster colors define the shapes in the final vector art but their node positions currently align with the original pixel art. We will rely on a simulation step to optimize the node positions in a spring system. Such a relaxation process is also often used for graph drawing algorithms [Kob12]. To this extent, we define the following forces:

F_O: Each node's position P is connected to its original position P_O by a force F_O to avoid its location to stray far from its origin.

$$\vec{F}_O = \left(\vec{P}_O - \vec{P}\right) \cdot \left|\vec{P}_O - \vec{P}\right| \cdot K_O, \tag{1}$$

where K_O is a stiffness value.
F_{Ni}: A pulling force contracts the edges at a node with location P to each of the locations P_{Ni} of its neighbor nodes:

$$\vec{F}_{Ni} = \left(\vec{P}_{Ni} - \vec{P}\right) \cdot K_N, \tag{2}$$

where K_N is a global stiffness value for the connection to the neighbor node. Hereby, short distances are preferred between neighboring nodes, which will lead to a smoothing of the paths.
Area force F_A: To avoid uncontrolled shrinking, a force is applied to the node locations P surrounding a shape to maintain its area Ai (computed using the shoelace formula):

$$\vec{F}_{Ai} = \left(\vec{P} - \vec{P}_{Ai}\right) \cdot \left(1 - \sqrt{\frac{Ai}{Ai_O}}\right), \tag{3}$$

Where P_{Ai} is the position of the centre of the area, Ai is the current area and Ai_O is the original area.

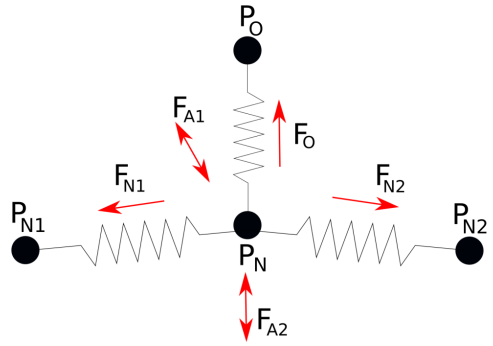

Figure 11: *Various forces acting on a node "P_N"*

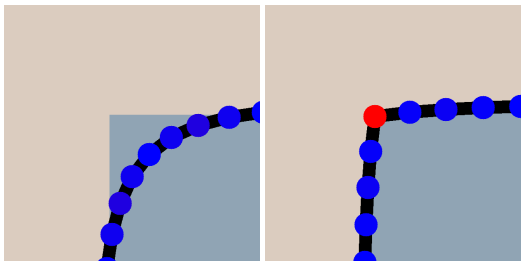

Figure 12: *The effect of different origin spring stiffness. Blue is low stiffness, red is high stiffness. Notice a sharp edge at the node with high stiffness.*

The resultant force F acting on a node location is the sum of the above-defined forces involving all its neighbour nodes Ns and adjacent areas As (see also Figure 11):

$$\vec{F} = \vec{F}_O + \sum_{i \in Ns} \vec{F}_{Ni} + \sum_{i \in As} \vec{F}_{Ai} \tag{4}$$

An important observation is that for a consistent relation, the stiffness value K_O for keeping points close to their origin should be linked to the number of neighbors. In practice, we thus define two global stiffness values for corner nodes (K_{OC}, 0.2 in practice) and edge nodes (K_{OE}, -0.1 in practice) and define:

$$K_O = \max\left(0, K_{OE} + (K_{OC} - K_{OE}) \cdot \left| \frac{2 \cdot i}{N-1} - 1 \right| \right) \tag{5}$$

Please note, while K_{OE} is negative, K_O is not, due to the use of a maximum. The usefulness of the latter can be illustrated by considering a diagonal of pixels; a too-strong force on the isolated pixel corners would result in staircase artifacts, which a lower K_O avoids.

After defining all forces, we perform the spring simulation in an iterative process as follows:

1. Calculate per-node forces (Eq. 4)
2. Calculate step size S:

$$S = S_{\max} / \max\left(1, \left|\vec{F}_{\max}\right|\right) \tag{6}$$

where $S_{\max}$ is a max step size (we use 0.1) and $F_{\max}$ is the largest force applied to a node in Step 1.

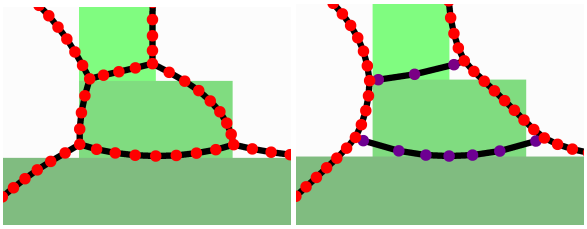

Figure 13: *The effect of different neighbour spring stiffness. Blue is low stiffness, red is high stiffness. Notice that with no adjustments the angles between branches are distributed uniformly, while the image with low stiffness at the horizontal lines allows the two vertical lines to appear smooth.*

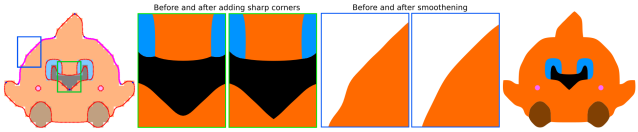

Figure 14: *Left to Right: User interactions (blue points represent sharp corners and points on paths that are to be smoothened are shown in pink), Effect of interactions, Our final result after interaction*

3. Scale all forces by S
4. Update all node locations by adding the sum of forces acting on the node to its current location.
5. If $F_{\max} > t$ goto 1, where t is a threshold (0.03 in practice)

User control: Modifying the values of K_N and K_O on a per-node basis provides easy and powerful control over the conversion process.

Role of K_O: Changing K_O affects how much a node is pulled towards its origin. If the stiffness is high, the node will be close to its origin. Hereby, we can anchor a node in its original location to produce sharp corners. (Shown in Figure 12)

Role of K_N: K_N controls tension, which causes nodes to tend towards each other. If there are shapes connecting in one location, changing the values for adjacent nodes can steer the smoothness of the region boundaries. (Figure 13)

Our UI offers options to increase/decrease the values of K_O and K_N. To leverage this concept and make it usable for both novice and advanced users, we provide two modes of interaction.

The *simple* mode enables a user to use a brushing tool to sharpen corners or smooth connections. This is achieved by assigning a global fixed value to K_N. Figure 14 shows a generated result (along with the before and after results). The *advanced* mode enables a user to individually edit the values of K_N and K_O, hereby changing the forces acting on each node. The interface relies on a brush tool that affects the forces. They are either set to a fixed value or a multiplicative factor is applied while brushing over nodes. Hereby, each brush pass, in/decreases forces locally. Figure 15 shows a result of these controls, including an explanation of the user interaction. The supplementary video shows the ease with which a user can interact with the result.

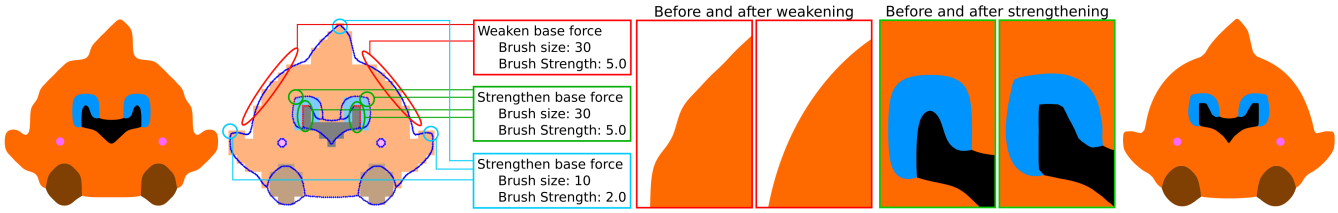

Figure 15: *Left to Right: Our result (Automatic), User interactions, Effect of interaction, Our final result (Interactive)*

3.4. Output Optimization

After the spring simulation, a vector image can be generated by creating polylines connecting adjacent nodes. Though this already gives a visually pleasing result, editing it in a dedicated software will be a difficult task because of the presence of many small line segments. To overcome this issue and to generate an easily-editable vector image, these polylines are further replaced by best-fitting Bezier curves. The overall procedure (Figure 16) is as follows:

Bezier fitting: We start by replacing the boundaries of each region with a set of best-fitting splines that approximate the polylines. As in [FLB16], we start by creating a curve network (V,E) using single-pixel-width polylines, where V denotes junctions and end-points, and E represents branches connecting two points in V. Then each pixel chain corresponding to the edges of the curve network is replaced with Bezier curves that minimize the following fitting error:

$$\varepsilon(e) = \sum_{p \in S^e} || B^e(t_p) - p ||_2^2, \tag{7}$$

where S^e is the chain of pixels associated with the edge e, while $t_p \in [0,1]$ is the normalized position of pixel p along S^e, and B^e is the parameterized Bezier curve. With additional constraints for connectivity, and continuity [FLB16] the curve can be solved for.

Curve grouping: In this step, the curves contouring a single closed external boundary are identified and merged. Hereby, the curves form a cycle, which is the boundary of a vectorial region.

Layering: Inspired by [PCS21, EPD09], we arrange the vectorial regions in layers. This layering arranges the shapes and makes it easier to edit them in standard vector applications. We follow [PCS21] and create a tree structure encoding the containment of vector shapes - contained regions form children of encompassing regions. Based on this tree arrangement, we then generate the layered output by stacking the regions accordingly (Figure 17).

Figure 17: *A figure depicting the arrangement in layering*

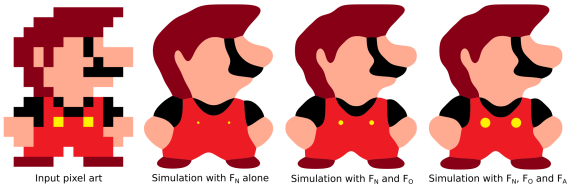

Figure 18: *Need for the various forces in our spring system. From left to right: Input pixel art, automatically generated results using only forces from neighbors (F_N), with additional origin forces (F_N and F_O), and our complete force system (with F_N, F_O and F_A). Only the rightmost result avoids regions that shrink too much and avoids the blobby appearance simpler systems produce in concave regions.*

Recoloring: The layered vector regions stem (by definition) from pixel clusters. The pixel-cluster color can be used as a fill color to provide a good match to the input pixel art.

4. Results and Discussion

Implementation details: Our interactive system is implemented in C++ using OpenCV. We run it on an Intel Xeon(R) CPU E5-1630 (v4 at 3.70GHzx8, GeForce RTX 2080Ti, and 32GiB RAM) and MacBook Pro 13" 2018 (2.3 GHz Quad-Core Intel Core i5 CPU, Intel Iris Plus Graphics 655 1536 MB GPU and 16 GB RAM). The resulting vector art can be stored in SVG format. It has to be noted that our system runs in real-time, offering seamless interaction.

Results and Comparison: As indicated, Figure 15 illustrates an

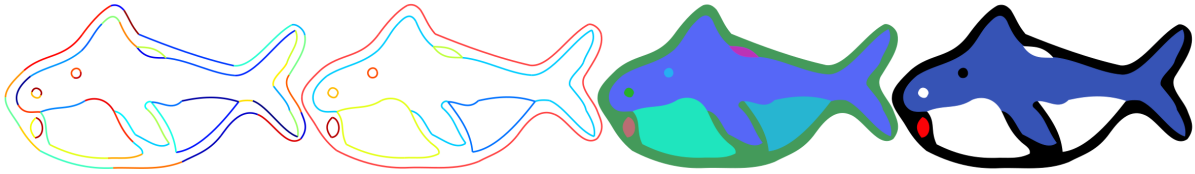

Figure 16: *Left to Right: Result after spring simulation, Bezier fitting, Curve grouping and Layering, Recoloring*

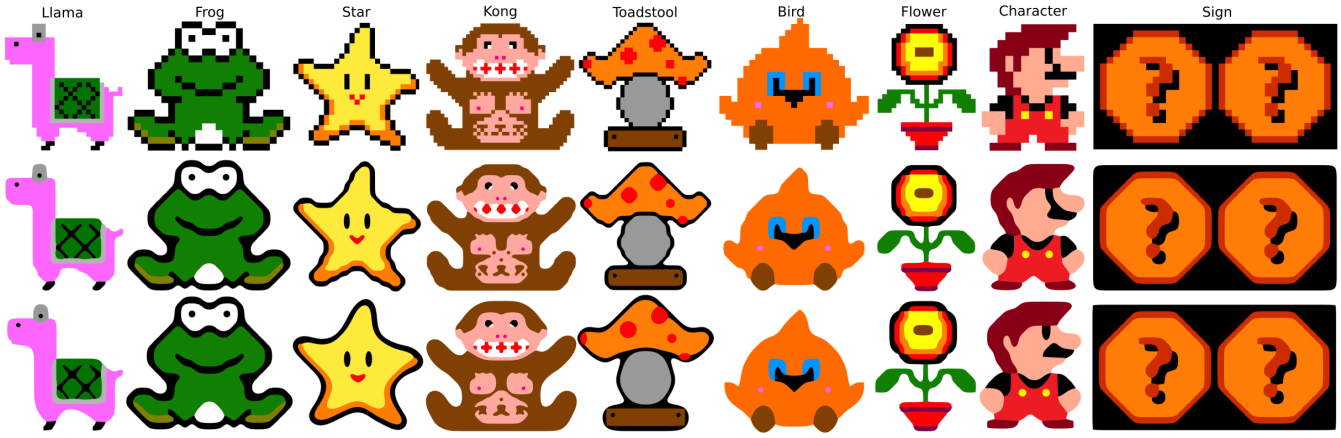

Figure 19: *A few results generated using our solution. Our method is able to handle several challenging cases that depend highly on the interpretation of the shapes by the user. Top to bottom: Input pixel art, automatic result, and the result after interaction*

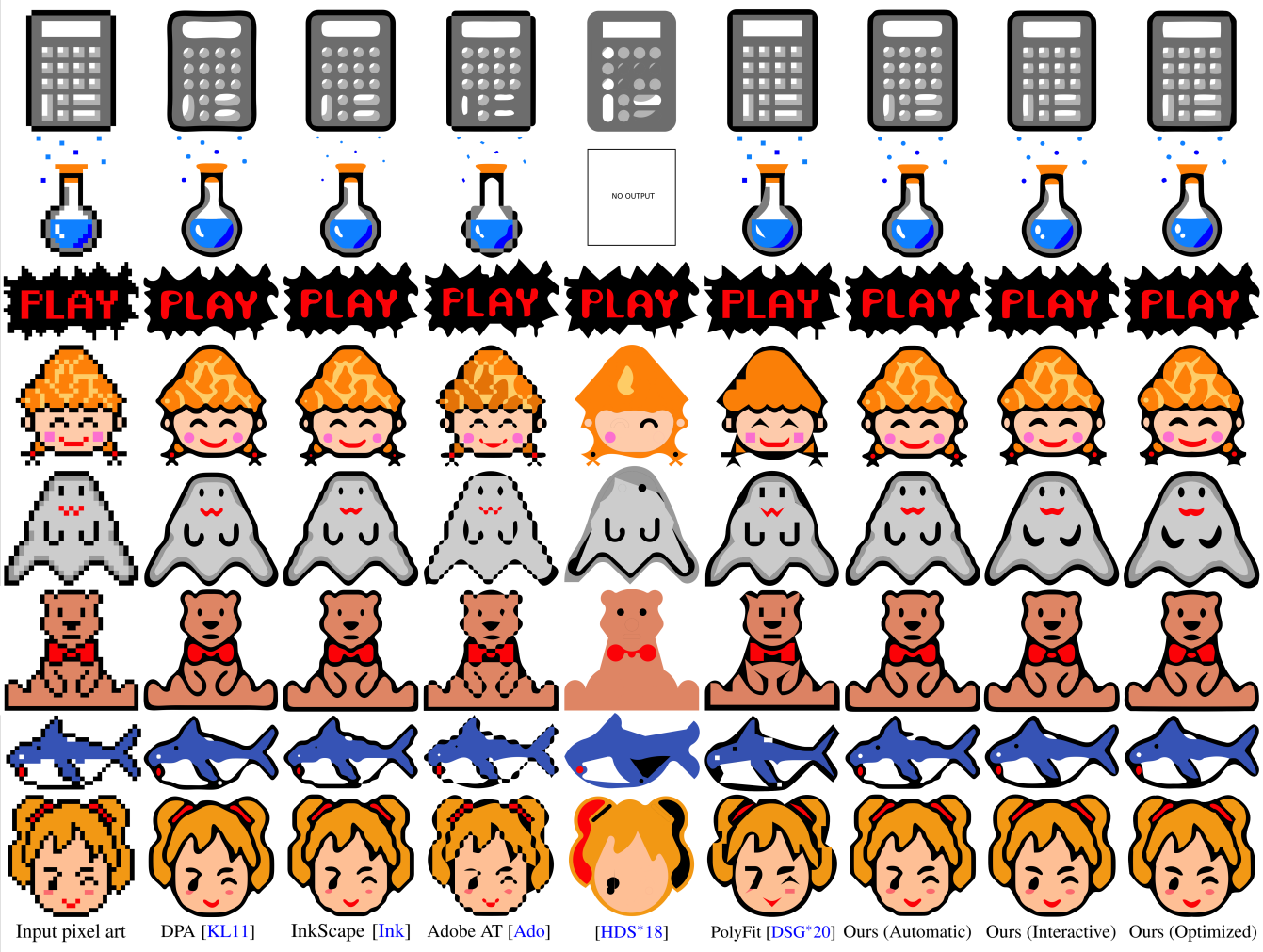

Figure 20: *State-of-the-art Comparison. The three rightmost examples are our results (automatic, interactive, splines)*

example of typical user intervention. Here, a user manipulates the forces to adjust sharp corners or ensure smooth curves. In this case, the user decided to make the eyes and pupils pointy in the upper left corner. The head boundary is additionally smoothed to remove unnecessary details on the silhouette. The interaction using our brushing metaphor is very easy and a modification like the one shown here, including some experimentation typically only takes 1-3 minutes (including time for processing along with visual inspection and decision-making).

It has to be noted that a simple curve smoothing procedure or a basic spring system is not enough for creating suitable conversions. Figure 18 shows an example that illustrates the results with and without various forces that our solution relies on. In the absence of the origin force F_O, the boundaries appear more blobby (especially visible concave regions - e.g., the sideburns, the curve connecting the legs or the moustache). Without the area force F_A, several regions shrink too strongly (e.g., the buttons and the gap between the cap and the face). In contrast, our complete spring system captures concavities well and preserves the area of important elements.

Figure 19 shows various very challenging results that show how we can disambiguate connections and create aesthetically-pleasing results. Starting on the left, we show that our solution handles regular and fine structures very well (cloth on the back of the llama). It is possible to maintain symmetry and even control the degree to which the pattern is tapered off. Such a case is very challenging for automatic approaches.

The frog shows a consistent outline. It is well captured in the vectorization and the controls enable us to maintain a smooth boundary. The eye ovals are fused well and the tummy shows a clean boundary. A particularly noteworthy part is the arm contour lines ending in the interior of the body and the well-handled overlap with the legs.

The star shape is a good example of how shading is handled. In this case, the smoothness of the exterior and interior contour work together nicely to support the roundish surface of the star.

The Kong example illustrates that even complex shapes are easily handled by our solution. Features such as the gaps between the teeth, as well as the indications of fur, can be adjusted to the liking of the user, here producing smoother shapes in the chest region.

The toadstool example exhibits the maintenance of a straight line at the bottom and well-shaped dots. Further, even pixel-sized screws are maintained.

The next illustration of the bird shows that smooth and pointy features can be well combined. The outer contour is clean and the feet overlap with the body, giving an impression of depth.

The flower has a round overall shape, yet the user opted for underlining the flower petal shapes in the final vector illustration. Four petals become visible, while the final output remains faithful to the original pixel drawing. Further, the precise placement of the flower on top of its pot (with its stripe and base being perfectly aligned), shows that our method can maintain parallel lines.

The next example of a character shows how even complex pixel combinations are handled. Here, the buttons are correctly integrated into the trouser, despite being also adjacent to the collar region, while the ear region is fused correctly with the face. Also, note how the eye was disambiguated to increase the impression of a comic character.

Finally, the signs combine straight boundary edges and a curvy but well-recognizable question-mark character with shadow indication, which our approach handles well. Both signs are converted identically, showing the consistency of our process.

To compare to other methods, we take a closer look at Figure 20.

DPA [KL11]: The state-of-the-art method gave an outstanding result but lacked the option for user interaction. This drawback made the algorithm perform slightly worse in some cases. Especially, sharp corners and some ambiguities are not fully resolved. For example, the calculator display appears round, the mouth in the fourth row and the arms of the ghost below cannot be modified to have pointy endpoints, the bow tie of the bear is broken, the white dot below the orca fin, or the eye of the girl.

InkScape [Ink]: It contains one of the best open-source pixel-art tracers available for professional usage, which was built upon DPA. The results are comparatively neat and aesthetically pleasing compared to the original DPA. It still fails at similar locations and boundaries do not appear entirely clean.

Adobe AT [Ado]: This method stays very close to the original pixel content. It is usually intended for natural images and struggles with the pixel-art input for which it was not conceived.

Perception-driven semi-structured boundary vectorization [HDS*18]: The learning-based algorithm has a special focus on perception-driven sharp feature detection. It struggles significantly with the provided input. The paper indicates a focus on slightly higher resolutions, where regions/structures are more pronounced, which is not the case for low-resolution pixel art. This could explain the poor results and that the algorithm would not accept one input.

PolyFit [DSG*20]: The vectorization technique captures shapes overall well but the polygonal appearance of the output shows the difficulty of handling ambiguous low-resolution input.

Our: Our automatic results are visually comparable to the results of InkScape. As all methods, our automated solution struggles with some of the mentioned ambiguities. Further, our boundary is not always as smooth as that of DPA. Yet, the fact that our method also involves the region area had a positive benefit, e.g., for the sparkles around the bottle, which many methods reduced. By involving the user, we were able to alleviate all significant problems other methods faced. Sharp features (e.g., calculator keys/display, hair of the girl), local smoothness (e.g., the boundary of the bear was made particularly smooth, including the outline, but also the bottle or ghost appear smooth), and manual connections (the bow tie, the stylistic choice for the bottle reflection, the orca outline) all contributed positively and were designed in less than 3 minutes.

User evaluation: To evaluate the various aspects of our interface, we invited ten users (with various levels of expertise from school kids to design students aged between 14 and 45) to participate in our user study. To access different features, we conducted the user study in three parts, namely:

- User study 1 - Interactivity - Evaluating interface ease of use
- User study 2 - Editability - Evaluating ease of editing results generated by various similar systems

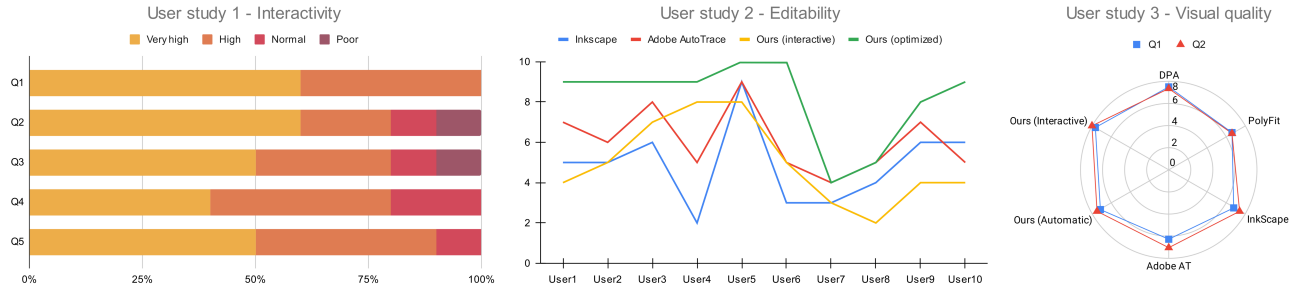

Figure 21: *User scores on 1) Interactivity scores of our interface, 2) Editability scores of our results compared to other methods, 3) Visual aesthetics and perceptual resemblance scores for results generated using various methods*

- User study 3 - Visual quality - Evaluating how aesthetically pleasing our results are compared to other methods.

User study 1 - Interactivity: In this part of the study, initially, we showed the users a demo of our interface and briefed them on controls and related effects. Based on the demo, they were asked to vectorize a sample image to familiarize themselves with the application. Later on, they were asked to randomly pick a few pixel arts and use our interface to vectorize them. Finally, the users were asked to rate the interface based on the following questions:

Q1: How easy is the interface to use?
Q2: How close is the final result compared to what they perceived?
Q3: How easy is it to control the output?
Q4: How time consuming is the vectorization process?
Q5: How predictable are the results (w.r.t. the user input)?

A few generic comments we received during the study are:

- There were only two options - left click to sharpen and right click to smoothen, which were really easy to use
- The head [...] got smoothened exactly how I wanted it to be.
- Once we know, it is very easy to control.
- It was time-consuming when I had to [define] a lot of sharp points.

User study 2 - Editability: The users were asked to edit the final results generated by various methods, and score the methods based on how easy it is to edit to get the final result the user had in mind. The scores were in the range of 1 to 10, where 1 being extremely difficult, and 10 being extremely easy. Comments received during the study included:

- Removing connections in the SVG files is extremely difficult (for both InkScape and Adobe AutoTrace), whereas it was easy to edit it in the image itself (the participant referred to the edge disambiguation of our method).
- There was no layering in Inkscape, making it difficult to resize the components.
- Adobe AutoTrace had large boundaries (most of them occluded by other parts) making it non-intuitive to manipulate.
- There are a lot of points in the output making it extremely difficult to move (referring to our unoptimized result).
- Having only a few control points and editing them was very easy (referring to our optimized results using Bezier curves).

User study 3 - Visual quality: To compare the aesthetic beauty of various methods, we showed the users-generated results side by side involving different methods (the order was randomly shuffled in each row to avoid bias). In other words, they could see all results at once without knowing which method produced which result and were asked to grade them twice (on a scale of 1-10) with respect to:

- Visual Aesthetics: How good are the results visually?
- Perceptual Resemblance: How well do the results perceptually match the input?

Figure 21 summarizes the result of the evaluation. As can be seen, our interface is easy to use, and it enabled the generation of results that match the user's expectations. Moreover, though slightly worse, our automatic algorithm performs comparable to the state-of-the-art methods and leads to better results after user interaction.

5. Conclusions and Future Work

We introduced a novel interactive vectorization algorithm building upon a spring-simulation framework to vectorize pixel art. While our algorithm can automatically generate results comparable to the state of the art, we designed our approach to include the user in the conversion process. Contrary to automatic solutions that depend on heuristics, a very small effort can steer the conversion towards high-quality results. Users resolved ambiguities, indicated sharp corners and established smoothness, to meet their interpretation of the pixel input. This core idea of using interaction to drive the conversion could be easily incorporated into other systems like DPA [KL11].

In the future, we want to apply user inputs also for the vector generation of natural raster images. Additionally, we would like to extend our method to animation, which can be beneficial for converting entire sprite sheets into games. Finally, we are considering integrating additional user annotations to control boundary definitions with guiding strokes and want to include gradients in the conversion as well.

References

[Ado] ADOBE INC.: Adobe illustrator. URL: https://adobe.com/products/illustrator. 1, 8

[AKB20] ANWAR S., KHAN S., BARNES N.: A deep journey into super-resolution: A survey. *ACM Comput. Surv. 53*, 3 (May 2020). 2

[CCS*14] CUI Z., CHANG H., SHAN S., ZHONG B., CHEN X.: Deep network cascade for image super-resolution. In *Computer Vision – ECCV 2014* (Cham, 2014), Fleet D., Pajdla T., Schiele B., Tuytelaars T., (Eds.), Springer International Publishing, pp. 49–64. 2

[DLHT14] DONG C., LOY C. C., HE K., TANG X.: Learning a deep convolutional network for image super-resolution. In *Computer Vision – ECCV 2014* (Cham, 2014), Fleet D., Pajdla T., Schiele B., Tuytelaars T., (Eds.), Springer International Publishing, pp. 184–199. 2

[DSG*20] DOMINICI E. A., SCHERTLER N., GRIFFIN J., HOSHYARI S., SIGAL L., SHEFFER A.: Polyfit: Perception-aligned vectorization of raster clip-art via intermediate polygonal fitting. *ACM Trans. Graph. 39*, 4 (July 2020). 1, 2, 8

[eag] Eagle. http://everything2.com/index.pl?node_id=1859453. 1997. 2

[EPD09] EISEMANN E., PARIS S., DURAND F.: A visibility algorithm for converting 3d meshes into editable 2d vector graphics. *ACM Trans. Graph. (Proc. of SIGGRAPH) 28* (July 2009), 83:1–83:8. 6

[EWHS08] EISEMANN E., WINNEMOELLER H., HART J. C., SALESIN D.: Stylized vector art from 3d models with region support. *Computer Graphics Forum (Proc. of EGSR) 27*, 4 (June 2008). 2

[Fat07] FATTAL R.: Image upsampling via imposed edge statistics. *ACM Trans. Graph. 26*, 3 (July 2007), 95–es. 2

[FF11] FREEDMAN G., FATTAL R.: Image and video upscaling from local self-examples. *ACM Trans. Graph. 30*, 2 (Apr. 2011). 2

[FLB16] FAVREAU J.-D., LAFARGE F., BOUSSEAU A.: Fidelity vs. simplicity: A global approach to line drawing vectorization. *ACM Trans. Graph. 35*, 4 (jul 2016). 6

[FMS98] FEKRI F., MERSEREAU R., SCHAFER R.: A generalized interpolative vq method for jointly optimal quantization and interpolation of images. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)* (1998), vol. 5, pp. 2657–2660 vol.5. 2

[Fri22] FRISKEN S. F.: Surfacenets for multi-label segmentations with preservation of sharp boundaries. *Journal of Computer Graphics Techniques (JCGT) 11*, 1 (February 2022), 34–54. 2

[GDA*13] GERSTNER T., DECARLO D., ALEXA M., FINKELSTEIN A., GINGOLD Y., NEALEN A.: Pixelated image abstraction with integrated user constraints. *Computers & Graphics 37*, 5 (2013), 333–347. 2

[HDS*18] HOSHYARI S., DOMINICI E. A., SHEFFER A., CARR N., WANG Z., CEYLAN D., SHEN I.-C.: Perception-driven semi-structured boundary vectorization. *ACM Trans. Graph. 37*, 4 (July 2018). 1, 2, 8

[hqx] Maxim stepin. hqx. http://web.archive.org/web/20070717064839/www.hiend3d.com/hq4x.html. 2003. 2

[Ink] INKSCAPE PROJECT: Inkscape. URL: https://inkscape.org. 1, 8

[IVK13] INGLIS T. C., VOGEL D., KAPLAN C. S.: Rasterizing and antialiasing vector line art in the pixel art style. In *International Symposium on Non-Photorealistic Animation and Rendering* (2013), Cole F., Grimm C., (Eds.), ACM. 2

[JCW09] JESCHKE S., CLINE D., WONKA P.: A gpu laplacian solver for diffusion curves and poisson image editing. *ACM Trans. Graph. 28*, 5 (Dec. 2009), 1–8. 2

[Key81] KEYS R.: Cubic convolution interpolation for digital image processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing 29* (1981), 1153–1160. 2

[KL11] KOPF J., LISCHINSKI D.: Depixelizing pixel art. *ACM Trans. Graph. 30*, 4 (July 2011). 1, 2, 3, 8, 9

[Kob12] KOBOUROV S. G.: Spring embedders and force directed graph drawing algorithms. *arXiv preprint arXiv:1201.3011* (2012). 4

[LHM09] LAI Y.-K., HU S.-M., MARTIN R. R.: Automatic and topology-preserving gradient mesh generation for image vectorization. *ACM Trans. Graph. 28* (2009). 2

[LL06] LECOT G., LEVY B.: Ardeco: Automatic Region DEtection and COnversion. In *Symposium on Rendering* (2006), Akenine-Moeller T., Heidrich W., (Eds.), The Eurographics Association. 2

[LLGRK20] LI T.-M., LUKÁČ M., GHARBI M., RAGAN-KELLEY J.: Differentiable vector graphics rasterization for editing and learning. *ACM Transactions on Graphics (TOG) 39*, 6 (2020), 1–15. 1

[OBB*13] ORZAN A., BOUSSEAU A., BARLA P., WINNEMÖLLER H., THOLLOT J., SALESIN D.: Diffusion curves: A vector representation for smooth-shaded images. *Commun. ACM 56*, 7 (July 2013), 101–108. 2

[PCS21] PARAKKAT A. D., CANI M.-P. R., SINGH K.: Color by numbers: Interactive structuring and vectorization of sketch imagery. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2021), CHI '21, Association for Computing Machinery. 6

[RGLM21] REDDY P., GHARBI M., LUKAC M., MITRA N. J.: Im2vec: Synthesizing vector graphics without vector supervision, 2021. arXiv:2102.02798. 2

[RL16] RESHETOV A., LUEBKE D.: Infinite resolution textures. In *Proceedings of High Performance Graphics* (Goslar, DEU, 2016), HPG '16, Eurographics Association, p. 139–150. 2

[SB19] STASIK P. M., BALCEREK J.: Extensible implementation of reliable pixel art interpolation. *Foundations of Computing and Decision Sciences 44*, 2 (2019), 213–239. 2

[SBZ05] SYKORA D., BURIÁNEK J., ZARA J.: Sketching Cartoons by Example. In *Eurographics Workshop on Sketch-Based Interfaces and Modeling* (2005), Jorge J. A. P., Igarashi T., (Eds.), The Eurographics Association. 2

[sca] Andrea mazzoleni. scale2x. http://everything2.com/index.pl?node_id=1859453. 2001. 2

[SEH08] STROILA M., EISEMANN E., HART J. C.: Clip art rendering of smooth isosurfaces. *IEEE Transactions on Visualization and Computer Graphics 14*, 1 (jan 2008), 135–145. 2

[Sil15] SILBER D.: *Pixel Art for Game Developers (1st ed.)*. A K Peters/CRC Press, 2015. 1

[TC04] TUMBLIN J., CHOUDHURY P.: Bixels: Picture samples with sharp embedded boundaries. In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques* (Goslar, DEU, 2004), EGSR'04, Eurographics Association, p. 255–264. 2

[TM96] THURNHOFER S., MITRA S. K.: Edge-enhanced image zooming. *Optical Engineering 35*, 7 (1996), 1862 – 1870. 2

[VPB*22] VINKER Y., PAJOUHESHGAR E., BO J. Y., BACHMANN R. C., BERMANO A. H., COHEN-OR D., ZAMIR A., SHAMIR A.: Clipasso: Semantically-aware object sketching. *ACM Trans. Graph. 41*, 4 (jul 2022). 2

[WCH21] WANG Z., CHEN J., HOI S. H.: Deep learning for image super-resolution: A survey. *IEEE Transactions on Pattern Analysis & Machine Intelligence 43*, 10 (oct 2021), 3365–3387. 2

[Wir] Bryan wirtz: Video games history: From magnavox odyssey to wii, there's no stopping the gaming industry. https://www.gamedesigning.org/gaming/history/. 2020. 1

[WPM*18] WANG Y., PERAZZI F., MCWILLIAMS B., SORKINE-HORNUNG A., SORKINE-HORNUNG O., SCHROERS C.: A fully progressive approach to single-image super-resolution. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops* (2018), pp. 864–873. 2

[YCZ*16] YANG M., CHAO H., ZHANG C., GUO J., YUAN L., SUN J.: Effective clipart image vectorization through direct optimization of bezigons. *IEEE Transactions on Visualization and Computer Graphics 22*, 2 (Feb. 2016), 1063–1075. 2

[ZCZ*09] ZHANG S.-H., CHEN T., ZHANG Y.-F., HU S.-M., MARTIN R. R.: Vectorizing cartoon animations. *IEEE Transactions on Visualization and Computer Graphics 15*, 4 (2009), 618–629. 2