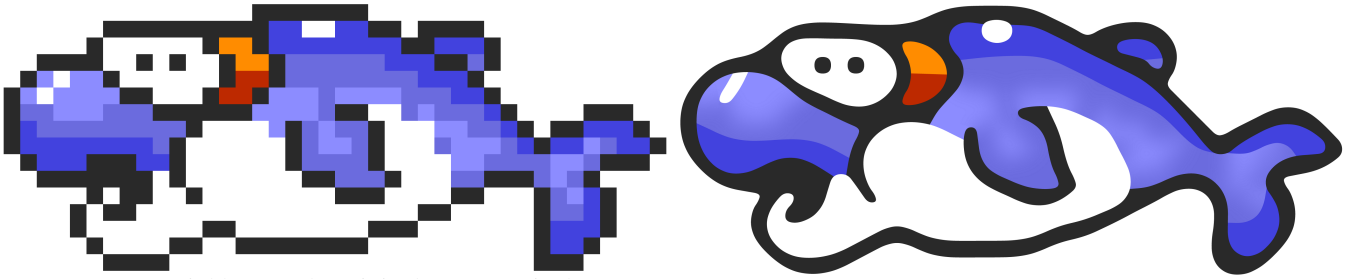
Depixelizing Pixel Art

Johannes Kopf
Microsoft Research

Dani Lischinski
The Hebrew University



Nearest-neighbor result (original: 40×16 pixels) Our result

Figure 1: *Naïve upsampling of pixel art images leads to unsatisfactory results. Our algorithm extracts a smooth, resolution-independent vector representation from the image, which is suitable for high-resolution display devices. (Input image © Nintendo Co., Ltd.).*

Abstract

We describe a novel algorithm for extracting a resolution-independent vector representation from *pixel art* images, which enables magnifying the results by an arbitrary amount without image degradation. Our algorithm resolves pixel-scale features in the input and converts them into regions with smoothly varying shading that are crisply separated by piecewise-smooth contour curves. In the original image, pixels are represented on a square pixel lattice, where diagonal neighbors are only connected through a single point. This causes thin features to become visually disconnected under magnification by conventional means, and creates ambiguities in the connectedness and separation of diagonal neighbors. The key to our algorithm is in resolving these ambiguities. This enables us to reshape the pixel cells so that neighboring pixels belonging to the same feature are connected through edges, thereby preserving the feature connectivity under magnification. We reduce pixel aliasing artifacts and improve smoothness by fitting spline curves to contours in the image and optimizing their control points.

Keywords: pixel art, upscaling, vectorization

Links: DL PDF WEB

1 Introduction

Pixel art is a form of digital art where the details in the image are represented at the pixel level. The graphics in practically all computer and video games before the mid-1990s consist mostly of pixel art. Other examples include icons in older desktop environments, as well as in small-screen devices, such as mobile phones. Because of the hardware constraints at the time, artists where forced to work with only a small indexed palette of colors and meticulously arrange *every pixel* by hand, rather than mechanically downscaling higher resolution artwork. For this reason, classical pixel art is usually marked by an economy of means, minimalism, and inherent modesty, which some say is lost in modern computer graphics. The best pixel art from the golden age of video games are masterpieces, many of which have become cultural icons that are instantly recognized by a whole generation, e.g. "Space Invaders" or the 3-color Super Mario Bros. sprite. These video games continue to be enjoyed today, thanks to numerous emulators that were developed to replace hardware that has long become extinct.

In this paper, we examine an interesting challenge: is it possible to take a small sprite extracted from an old video game, or an entire output frame from an emulator, and convert it into a resolution-independent vector representation? The fact that every pixel was manually placed causes pixel art to carry a maximum of expression and meaning per pixel. This allows us to infer enough information from the sprites to produce vector art that is suitable even for significant magnification. While the quantized nature of pixel art provides for a certain aesthetic in its own right, we believe that our method produces compelling vector art that manages to capture some of the charm of the original (see Figure 1).

Previous vectorization techniques were designed for natural images and are based on segmentation and edge detection filters that do not resolve well the tiny features present in pixel art. These methods typically group many pixels into regions, and convert the regions' boundaries into smooth curves. However, in pixel art, every single pixel can be a feature on its own or carry important meaning. As a result, previous vectorization algorithms typically suffer from detail loss when applied to pixel art inputs (see Figure 2).

A number of specialized pixel art upscaling methods have been developed in the previous decade, which we review in the next section. These techniques are often able to produce commendable results. However, due to their local nature, the results suffer from staircasing artifacts, and the algorithms are often unable to correctly resolve locally-ambiguous pixel configurations. Furthermore, the magnification factor in all these methods is fixed to $2\times$, $3\times$, or $4\times$.

In this work, we introduce a novel approach that is well suited for pixel art graphics with features at the scale of a single pixel. We first resolve all separation/connectedness ambiguities of the original pixel grid, and then reshape the pixel cells, such that connected neighboring pixels (whether in cardinal or diagonal direction) share an edge. We then fit spline curves to visually significant edges and optimize their control points to maximize smoothness and reduce staircasing artifacts. The resulting vector representation can be rendered at any resolution.

ACM Reference Format
Kopf, J., Lischinski, D. 2011. Depixelizing Pixel Art. *ACM Trans. Graph. 30*, 4, Article 99 (July 2011), 8 pages. DOI = 10.1145/1964921.1964994 http://doi.acm.org/10.1145/1964921.1964994.

PhotoZoom 4 (General Image Upsampling) hq4x (Specialized Pixel Art Upscaling) Adobe Live Trace (Vectorization)

Figure 2: *Results achieved with representatives of different categories of algorithms. Compare these results to ours shown in Figure 1.*

We successfully applied our algorithm to an extensive set of pixel art images extracted from vintage video games and desktop icons, as well as to complete frames produced by a Super Nintendo emulator. We compare our results to various alternative upscaling methods, ranging from vectorization to general and pixel-art-specialized image upscaling methods. In addition to the examples included in this paper, all of our results and comparisons are included in the supplementary material to this paper.

2 Previous Work

The previous work related to our paper can be classified into three categories. In Figures 2 and 9, and, more extensively, in the supplementary materials, we compare our algorithm to various representatives from each category.

General Image Upsampling

The "classical" approach to image upsampling is to apply linear filters derived either from analytical interpolation or from signal processing theory. Examples include filters such as Nearest-Neighbor, Bicubic, and Lancosz [Wolberg 1990]. These filters make no assumptions about the underlying data, other than that it is essentially band-limited. As a consequence, images upsampled in this manner typically suffer from blurring of sharp edges and ringing artifacts.

In the last decade, many sophisticated algorithms have appeared which make stronger assumptions about the image, e.g., assuming natural image statistics [Fattal 2007] or self-similarity [Glasner et al. 2009]. A comprehensive review of all these methods is well beyond the scope of this paper. However, in most cases, these (natural) image assumptions do not hold for color-quantized, tiny pixel art images. For this reason, these methods tend to perform poorly on such inputs.

Pixel Art Upscaling Techniques

A number of specialized pixel art upscaling algorithms have been developed over the years [Wikipedia 2011]. Most of these have their origins in the emulation community. None has been published in a scientific venue; however, open source implementations are available for most. All of these algorithms are pixel-based and upscale the image by a fixed integer factor.

The first algorithm of this type that we are aware of is EPX, which was developed by Eric Johnston in 1992 to port LucasArts games to early color Macintosh computers, which had then about double the resolution than the original platform [Wikipedia 2011]. The algorithm doubles the resolution of an image using a simple logic: every pixel is initially replaced by 2×2 block of the same color; however, if the left and upper neighbors in the original image had the same color, that color would replace the top left pixel in the 2×2 block, and analogously for the other corners.

The algorithm is simple enough to be applied in real-time and often achieves good results. However, the direction of edges is quantized to only 12 distinct directions which can cause the results to appear blocky. Another limitation is in the strict local nature of the algorithm which prevents it from correctly resolving ambiguous connectedness of diagonal pixels. Both of these limitations are demonstrated in Figure 9 (bottom right).

Several later algorithms are based on the same idea, but use more sophisticated logic to determine the colors of the 2×2 block. The best known ones are Eagle (by Dirk Stevens), 2xSaI [Liauw Kie Fa 2001], and Scale2x [Mazzoleni 2001], which use larger causal neighborhoods and blend colors. Several slightly different implementations exist under different names, such as SuperEagle and Super2xSaI. An inherent limitation of all these algorithms is that they only allow upscaling by a factor of two. Larger magnification can be achieved by applying the algorithm multiple times, each time doubling the resolution. This strategy, however, significantly reduces quality at larger upscaling factors, because the methods assume non-antialiased input, while producing antialiased output.

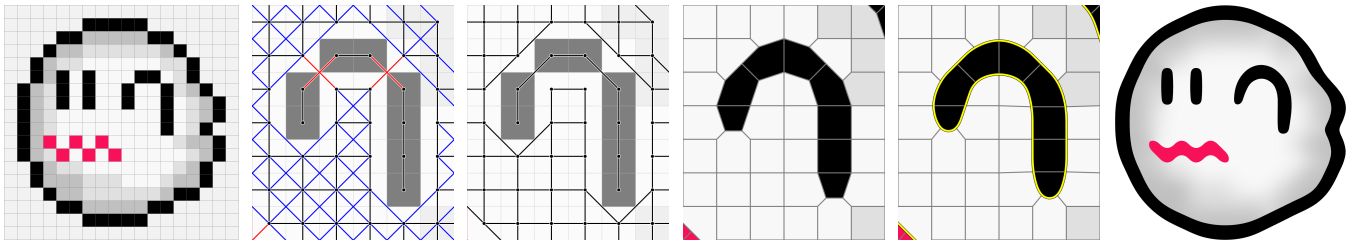
The latest and most sophisticated evolution of this type of algorithms is the hqx family [Stepin 2003]. This algorithm examines 3×3 pixel blocks at a time and compares the center pixel to its 8 neighbors. Each neighbor is classified as being either similar or dissimilar in color, which leads to 256 possible combinations. The algorithm uses a lookup table to apply a custom interpolation scheme for each combination. This enables it to produce various shapes, such as sharp corners, etc. The quality of the results is high. However, due to its strictly local nature, the algorithm cannot resolve certain ambiguous patterns and is still prone to produce staircasing artifacts. Lookup tables exist only for $2\times$, $3\times$, and $4\times$ magnification factors.

Image Vectorization

A large body of work deals with the automatic extraction of vector representations from images. These methods share a similar goal with our algorithm. However, most are designed with larger natural images in mind. At their core, vectorization methods rely on segmentation or edge detection algorithms to cluster many pixels into larger regions, to which vector curves and region primitives are fit. These clustering tools do not perform well on pixel art images, because features are tiny, all edges are step edges, and there are no gradients that can be followed downhill. For these reasons, all algorithms examined in this section tend to lose the small features characteristic to pixel art.

Another challenge for these algorithms is dealing with 8-connected pixels; many pixel art images contain thin features that are only connected through pixel corners. General image vectorization tools are not designed to handle this situation and the ambiguities that arise from it and consequently tend to break the connectivity of these features. Below, we mention only a few representative vectorization approaches.

Selinger [2003] describes an algorithm dubbed *Potrace* for tracing binary images and presents results on very tiny images. This method, however, cannot handle color images. These have to be quantized and decomposed into separate binary channels first, which are then traced separately. This results in inter-penetrating shapes.



(a) (b) (c) (d) (e) (f)

Figure 3: *Overview of our algorithm. (a) Input Image (16×16 pixels). (b) Initial similarity graph with crossing edges. The blue edges lie inside flat shaded regions, so they can be safely removed. In case of the red lines, removing one or the other changes the result. (c) Crossing edges resolved. (d) Reshaped pixel cells reflecting the connections in the resolved similarity graph. (e) Splines fit to visible edges. (f) Final result with splines optimized to reduce staircasing (Input image © Nintendo Co., Ltd.).*

Lecot and Lévy [2006] present a system ("ARDECO") for vectorizing raster images. Their algorithm computes a set of vector primitives and first- or second-order gradients that best approximates the image. This decomposition is based on a segmentation algorithm, which is unlikely to work satisfactorily on pixel art images. Lai *et al.* [2009] present an algorithm for automatic extraction of gradient meshes from raster images. This algorithm also relies on segmentation and is for the same reason unlikely to perform well on pixel art images.

Orzan *et al.* [2008] introduce image partitioning *diffusion curves*, which diffuse different colors on both sides of the curve. In their work, they also describe an algorithm for automatically extracting this representation from a raster image. Their formulation, however, relies on Canny edge detection. These filters do not work well on pixel art input, most likely due to the small image size, since edge detectors have a finite support. Xia *et al.* [2009] describe a technique that operates on a pixel level triangulation of the raster image. However, it also relies on Canny edge detection.

Various commercial tools, such as Adobe Live Trace [Adobe, Inc. 2010] and Vector Magic [Vector Magic, Inc. 2010], perform automatic vectorization of raster images. The exact nature of the underlying algorithms is not disclosed, however, they generally do not perform well on pixel art images, as is evidenced by the comparisons in this paper and in our supplementary material.

3 Algorithm

Our goal in this work is to convert pixel art images to a resolution-independent vector representation, where regions with smoothly varying shading are crisply separated by piecewise-smooth contour curves. While this is also the goal of general image vectorization algorithms, the unique nature of pixel art images poses some non-trivial challenges:

1. *Every pixel matters*. For example, a single pixel whose color is sufficiently different from its surrounding neighborhood is typically a feature that must be preserved (e.g., a character's eye).

2. *Pixel-wide 8-connected lines and curves*, such as the black outline of the ghost in Figure 3a. These features appear visually connected at the original small scale, but become visually disconnected under magnification.

3. *Locally ambiguous configurations*: for example, when considering a 2×2 checkerboard pattern with two different colors, it is unclear which of the two diagonals should be connected as part of a continuous feature line (see the mouth and the ear of the ghost in Figure 3a). This problem has been studied in the context of foreground / background separation in binary images, and simple solutions have been proposed [Kong and Rosenfeld 1996]. The challenge, however, is more intricate in the presence of multiple colors.

4. The jaggies in pixel art are of a large scale compared to the size of the image, making it *difficult to distinguish between features and pixelization artifacts*. For example, in Figure 3a, how does one know that the mouth should remain wiggly, while the ghost outline should become smooth?

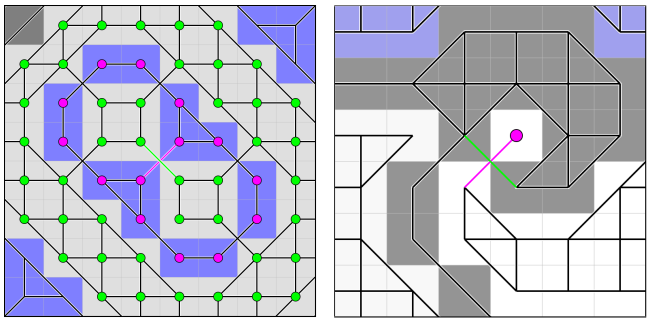
3.1 Overview

The main primitive in our vector representation are quadratic B-spline curves, which define the piecewise smooth contours between regions. Once the curves are computed, an image can be rendered using standard tools [Nehab and Hoppe 2008; Jeschke et al. 2009]. Thus, our main computational task is to determine the location and the precise geometry of these contours. Similarly to other vectorization algorithms, this boils down to detecting the edges in the input pixel art image and fitting curves of the proper shape to them. However, this process is complicated by the reasons listed earlier.

Because of the small scale of pixel art images and the use of a limited color palette, localizing the edges is typically easy: any two adjacent pixels with sufficiently different colors should be separated by a contour in the vectorized result. However, the challenge lies in connecting these edge segments together, while correctly handling 8-connected pixels and resolving local ambiguities.

Consider a square lattice graph with $(w+1)\times(h+1)$ nodes, representing a $w\times h$ image. Each pixel corresponds to a closed cell in this graph. Horizontal and vertical neighbor cells share an edge in this graph, while diagonal neighbors share only a vertex. These diagonal neighbors become visually disconnected when the lattice is magnified, while the neighbors that share an edge remain visually connected. Thus, the first step of our approach is to reshape the original square pixel cells so that every pair of neighboring pixels along a thin feature, which should remain connected in the vectorized result, correspond to cells that share an edge. In this process, described in Section 3.2, we employ a few carefully designed heuristics to resolve locally ambiguous diagonal configurations.

Having reshaped the graph, we identify edges where the meeting pixels have significantly different colors. We refer to these edges as *visible* because they form the basis for the visible contours in our final vector representation, whereas the remaining edges will not be directly visible as they will lie within smoothly shaded regions. To produce smooth contours, we fit quadratic B-spline curves to sequences of visible edges, as described in Section 3.3. However, because the locations of the curve control points are highly quantized due to the low-resolution underlying pixel grid, the results might still exhibit staircasing. We therefore optimize the curve shapes to reduce the staircasing effects, while preserving intentional high-curvature features along each contour, as described in Section 3.4.



(a) Sparse pixels heuristic (b) Islands heuristic

Figure 4: *Heuristics for resolving crossing edges in the similarity graph:* **Curves:** *not shown here, see Figure 3b.* **Sparse pixels:** *the magneta component is sparser than the green one. This heuristic supports keeping the magenta edge connected.* **Islands:** *In this case the heuristic supports keeping the magenta edge connected, because otherwise a single pixel "island" would be created.*

Finally, we render an image by interpolating colors using radial basis functions. This is done in an edge-aware manner, so that the influence of each pixel color does not propagate across the contour lines.

3.2 Reshaping the pixel cells

The goal of this first stage is to reshape the pixel cells, so that neighboring pixels that have similar colors and belong to the same feature share an edge. To determine which pixels should be connected in this way, we create a similarity graph with a node for each pixel. Initially, each node is connected to all eight of its neighbors. Next, we remove from this graph all of the edges that connect pixels with dissimilar colors. Following the criteria used in the hqx algorithm [Stepin 2003], we compare the YUV channels of the connected pixels, and consider them to be dissimilar if the difference in Y, U, V is larger than $\frac{48}{255}$, $\frac{7}{255}$, or $\frac{6}{255}$ respectively.

Figure 3b shows the similarity graph that was processed in this manner. This graph typically contains many crossing diagonal connections. Our goal is now to eliminate all of these edge crossing in order to make the graph planar (Figure 3c). The dual of the resulting planar graph will have the desired property of connected neighboring pixel cells sharing an edge (Figure 3d).

We distinguish between two cases:

1. If a 2×2 block is fully connected, it is part of a continuously shaded region. In this case the two diagonal connections can be safely removed without affecting the final result. Such connections are shown in blue in Figure 3b.
2. If a 2×2 block only contains diagonal connections, but no horizontal and vertical connections, it means that removing one connection or the other will affect the final result. In this case we have to carefully choose which of the connections to remove. Such connections are shown in red in Figure 3b.

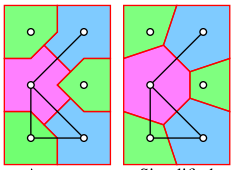
It is not possible to make this decision locally. If one only examines the 2×2 blocks with the red connections in Figure 3b, there is no way to determine whether the dark or the light pixels should remain connected. However, by examining a larger neighborhood it becomes apparent that the dark pixels form a long linear feature, and therefore should be connected, while the light pixels are part of the background.

Determining which connections to keep is related to Gestalt laws, and essentially aims to emulate how a human would perceive the figure. This is a very difficult task; however, we have developed three simple heuristics that, combined, properly resolve the connectivity issue in a surprisingly large number of cases, as evidenced by the large number of examples in our supplementary material. We compute an associated weight for each heuristic, and in the end choose to keep the connection that has aggregated the most weight. In case of a tie, both connections are removed. These heuristics are explained below:

Curves If two pixels are part of a long curve feature, they should be connected. A curve is a sequence of edges in the similarity graph that only connects valence-2 nodes (i.e., it does not contain junctions). We compute the length of the two curves that each of the diagonals is part of. The shortest possible length is 1, if neither end of the diagonal has valence of 2. This heuristic votes for keeping the longer curve of the two connected, with the weight of the vote defined as the difference between the curve lengths. Figure 3b shows two examples for this heuristic: the black pixels are part of a curve of length 7, whereas the white pixels are not part of a curve (i.e., length 1). Therefore, the heuristic votes for connecting the black pixels with a weight of 6.

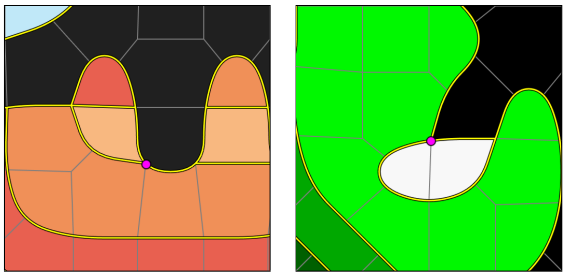
Sparse pixels in two-colored drawings, humans tend to perceive the sparser color as foreground and the other color as background. In this case we perceive the foreground pixels as connected (e.g., think of a dotted pencil line). We turn this into a heuristic by measuring the size of the component connected to the diagonals. We only consider an 8×8 window centered around the diagonals in question. This heuristic votes for connecting the pixels with the smaller connected component. The weight is the difference between the sizes of the components. This case is illustrated in Figure 4a.

Islands we attempt to avoid fragmentation of the figure into too many small components. Therefore, we avoid creating small disconnected islands. If one of the two diagonals has a valence-1 node, it means cutting this connection would create a single disconnected pixel. We prefer this not to happen, and therefore vote for keeping this connection with a fixed weight, with an empirically determined value of 5. This case is illustrated in Figure 4b.



Now that we have resolved the connectivities in the similarity graph, the graph is planar, and we can proceed to extracting the reshaped pixel cell graph. This can be done as follows: cut each edge in the similarity graph into two halves and assign each half to the node it is connected to. Then, the reshaped cell graph can be computed as a generalized Voronoi diagram, where each Voronoi cell contains the points that are closest to the union of a node and its half-edges. We can further simplify this graph through collapse of all valence-2 nodes. The inset figure shows the generalized Voronoi diagram corresponding to a simple similarity graph alongside its simplified version. Figure 3d shows the simplified Voronoi diagram for the similarity graph in Figure 3c. Note that the node positions are quantized in both dimensions to multiples of a quarter pixel. We will make use of this fact later when we match specific patterns in the graph.

The shape of a Voronoi cell is fully determined by its local neighborhood in the similarity graph. The possible distinct shapes are easy to enumerate, enabling an extremely efficient algorithm, which walks in scanline order over the similarity graph, matches specific edge configurations in a 3×3 block a time, and pastes together the corresponding cell templates. We directly compute the simplified Voronoi diagram in this manner, without constructing the exact one.



(a) (b)

Figure 5: *Resolving T-junctions at the marked nodes. Spline curves are shown in yellow. (a) In this case the left edge is a shading edge, therefore the other two edges are combined into a single spline curve. (b) Here, three differently colored regions meet at a point. The edges which form the straightest angle are combined into a single spline.*

3.3 Extracting Spline Curves

The reshaped cell graph resolves all connectivity issues and encodes the rough shape of the object. However, it contains sharp corners and may look "blocky" due to the quantized locations of the nodes. We resolve these issues by identifying the *visible* edges, where significantly different colors meet. Connected sequences of visible edges that contain only valence-2 nodes are converted into quadratic B-spline curves [de Boor 1978]. Here, we only count visible edges to determine the valence of a node. The control points of the B-splines are initialized to the node locations.

When three splines end at a single common node, we can choose to smoothly connect two of the splines into one, creating a T-junction. This is preferable because it leads to a simpler and smoother figure. The question is: which two out of three splines to connect?

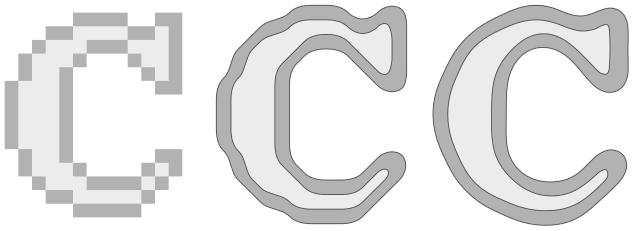
We first categorize each of the three visible edges meeting at the node as either a shading edge or a contour edge. Shading edges separate cells that have similar colors (which were nevertheless considered different enough to classify the edge as visible to begin with). Contour edges separate cells with strongly dissimilar colors. Specifically, in our implementation we classify an edge as a shading edge if the two cells have a YUV distance of at most $\frac{100}{255}$. Now, if at a 3-way junction we have one shading edge and two contour edges, we always choose to connect the contour edges. This situation is demonstrated in Figure 5a. If this heuristic does not resolve the situation, we simply measure the angles between the edges and connect the pair with the angle closest to 180 degrees. This situation is shown in Figure 5b.

One minor issue arises from the fact that B-spline curves are only approximating their control points, but do not interpolate them in general. For this reason, we have to adjust the endpoint of a curve that ends at a T-junction to properly lie on the curve that continues through the T-junction.

3.4 Optimizing the Curves

Fitting B-spline curves greatly improves the smoothness of our results; however, they still suffer from staircasing artifacts (Figure 6b). Therefore, we further improve the smoothness of the curves by optimizing the locations of their control points. The optimization seeks the minimum of a sum of per-node energy terms:

$$\arg\min_{\{p_i\}} \sum_i E^{(i)}, \tag{1}$$



(a) Input (13×15 pixels) (b) Splines Initialized (c) Splines Optimized

Figure 6: *Removing staircasing artifacts by minimizing spline curvature.*

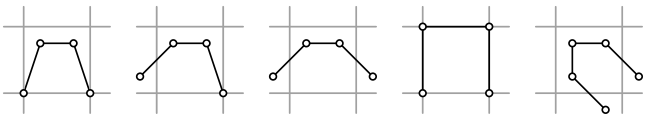


Figure 7: *Corner patterns our algorithm detects. The original square pixel grid is shown in gray. Detecting these patterns is straight forward because node locations are quantized to multiples of a quarter pixel in both dimensions due to the graph construction.*

where p_i is the location of the i-th node. The energy of a node is defined as the sum of smoothness and positional terms:

$$E^{(i)} = E_s^{(i)} + E_p^{(i)} \tag{2}$$

The two terms have equal contribution to the energy. Smoothness is measured as the absence of curvature. Therefore, we define the smoothness energy as

$$E_s^{(i)} = \int_{s \in r(i)} |\kappa(s)| \, \mathrm{d}s, \tag{3}$$

where $r(i)$ is the region of the curve that is influenced by p_i, and $\kappa(s)$ is the curvature at point s. We compute the integral numerically by sampling the curve at fixed intervals.

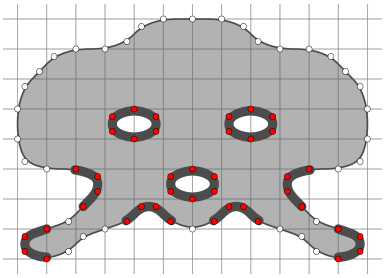
In order to prevent objects from changing too much, we need to further constrain the positions of the control points. We define a positional energy term as follows:

$$E_p^{(i)} = \|p_i - \hat{p}_i\|^4, \tag{4}$$

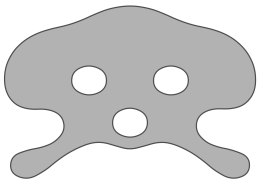
where, $\hat{p}_i$ is the initial location of the i-th node. Raising the term to the fourth power allows nodes to move relatively freely within a small region around their original location while sharply penalizing larger deviations.

Note that the energy function as defined above does not distinguish between staircasing artifacts and intentional sharp features, such as corners. In the former case, smoothing is desired, however, in the latter, it needs to be avoided. We correct this behavior by detecting sharp features in the model and excluding the regions around these features from the integration in Equation 3. Due to the quantized nature of the reshaped cell graph, sharp features can only take on a limited number of specific patterns, shown in Figure 7. Thus, we simply look for these patterns (including their rotations and reflections) in the reshaped cell graph. Having detected a pattern, we exclude the part of the spline curve between the nodes of the pattern from the integration (3). Figure 8 shows the detected patterns on an example sprite and highlights the parts of the curve which are excluded from the integration.

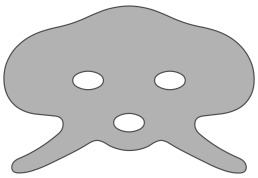
While our energy function is non-linear, it defines a very smooth potential surface, and therefore can be optimized using a simple



(a) Detected corner patterns



(b) Without corner detection



(c) With corner detection

Figure 8: *Corner detection: (a) Nodes that belong to a detected corner pattern are shown in red. The curve segments excluded from the evaluation of the smoothness term are highlighted in black.*

relaxation procedure. At each iteration, we do a random walk over the nodes and optimize each one locally. For each node, we try several random new offset positions within a small radius around its current location, and keep the one that minimizes the node's energy term.

After optimizing the locations of the spline nodes, the shape of the pixel cells around the splines might have changed significantly. We therefore compute new locations for all nodes that were not constrained in the optimization described above (and do not lie on the boundary of the image) using harmonic maps [Eck et al. 1995]. This method minimizes the cell distortion, and boils down to solving a simple sparse linear system (see Hormann [2001] for the exact form and simple explanation).

3.5 Rendering

Our vector representation can be rendered using standard vector graphics rendering techniques, e.g. the system described by Nehab and Hoppe [2008]. Diffusion solvers can also be used to render the vectors. Here, one would place color diffusion sources at the centroids of the cells and prevent diffusion across spline curves. Jeschke *et al.* [2009] describe a system that can render such diffusion systems in real time. The results in this paper were rendered using a slow but simple implementation, where we place truncated Gaussian influence functions ($\sigma = 1$, radius 2 pixels) at the cell centroids and set their support to zero outside the region visible from the cell centroid. The final color at a point is computed as the weighted average of all pixel colors according to their respective influence.

4 Results

We applied our algorithm to a wide variety of inputs from video games and other software. Figure 9 shows representative results of our algorithm and compares them to various alternative upscaling techniques. In the supplementary materials, we provide an extensive set of additional results and comparisons.

The performance of our algorithm depends on the size and the number of curves extracted from the input. While we did not invest much time into optimizing our algorithm, it already performs quite well. The following table summarizes the timings for the 54 examples included in the supplementary material, computed on a single core of a 2.4 GHz CPU:

	Median	Average	Min	Max
Similarity Graph Construction	**0.01s**	0.01s	0.00s	0.07s
Spline Extraction	**0.02s**	0.07s	0.00s	1.95s
Spline Optimization	**0.60s**	0.71s	0.01s	1.93s
Total	**0.62s**	0.79s	0.01s	3.06s

While our current focus was not on achieving real-time speed, we were still interested in how our algorithm would perform on animated inputs, e.g. in a video game. For this experiment, we dumped frames from a video game emulator to disk and applied our technique to the full frames. The two stages of our algorithm that are most critical for temporal coherency are the decisions made when connecting pixels in the similarity graph and the node location optimization. On the sequences we tried, our heuristics performed robustly in the sense that they made similar decisions on sprite features, even when they changed slightly throughout the animation phases. In the optimization, the node locations are actually quite constrained due to the large power in the positional energy term. For this reason, our results can never deviate much from the input and are therefore as temporally consistent as the input. Figure 10 shows a video game frame upscaled using our method. In the supplementary materials, we provide high resolution videos for a long sequence and compare these to other techniques.

4.1 Limitations

Our algorithm is designed specifically for hand-crafted pixel art images. Starting in the mid-nineties, video game consoles and computers were able to display more than just a handful of colors. On these systems, designers would start from high resolution multicolor images, or even photos, and then downsample them to the actual in-game resolution. This results in very anti-aliased sprites, which are in some sense closer to natural images than to the type of input our algorithm was designed for. The hard edges that we produce do not always seem suitable for representing such figures. Figure 11 shows an example of a sprite that was designed in this manner.

Another limitation is that our splines sometimes smooth certain features too much, e.g. the corners of the "386" chip in Figure 9. Our corner detection patterns are based on heuristics and might not always agree with human perception. One possible future extension is to allow increasing the multiplicity of the B-spline knot vector to create sharp features in the vector representation, for example in places where long straight lines meet at an angle.

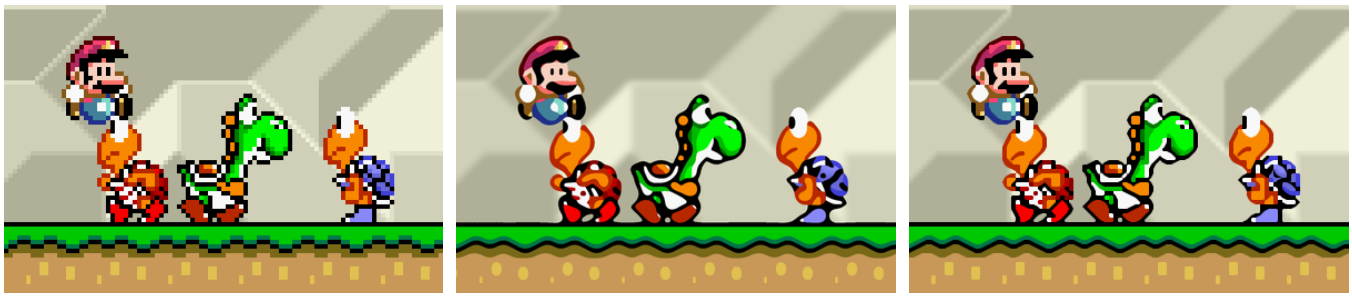
While many of our inputs use some form of anti-aliasing around edges, we have not experimented with pixel art that use strong dithering patterns, such as checker board patterns, to create the impression of additional shades. Our algorithm might need to be adapted to handle such inputs well.

5 Conclusions

We have presented an algorithm for extracting a resolution-independent vector representation from pixel art images. Our algorithm resolves separation/connectedness ambiguities in the square pixel lattice and then reshapes the pixel cells so that both cardinal and diagonal neighbors of a pixel are connected through edges. We extract regions with smoothly varying shading and separate them crisply through piecewise smooth contour curves. We have demonstrated that conventional image upsampling and vectorization algorithms cannot handle pixel art images well, while our algorithm produces good results on a wide variety of inputs.

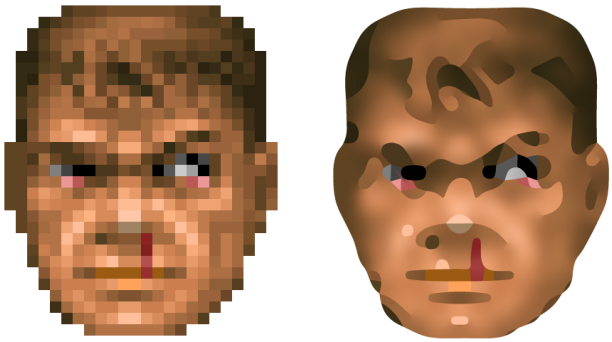
There are many avenues for future work. Obviously, it would be nice to optimize the performance of the algorithm so that it can be applied in real-time in an emulator. Some of the ideas presented in

Figure 9: *Some results created with algorithm and comparison to various competing techniques. Please zoom into the PDF to see details, and see the supplementary materials for a large number of additional results and comparisons (Input images: Keyboard, 386 © Microsoft Corp.; Help, Yoshi, Toad © Nintendo Co., Ltd.; Bomberman © Hudson Soft Co., Ltd.; Axe Battler © Sega Corp.; Invaders © Taito Corp.).*



Nearest Neighbor Result | Our Result | hq4x Result

Figure 10: *Applying pixel art upscaling in a dynamic setting. The output frame of an emulator is magnified by 4× (crop shown). Please zoom into the PDF to see details (Input image © Nintendo Co., Ltd.).*



Input (24×29 pixels) | Our Result

Figure 11: *A less successful case. Anti-aliased inputs are difficult to handle for our algorithm. "We are doomed..." (Input image © id Software)*

this work can potentially benefit general image vectorization techniques. Another interesting direction would be to improve the handling of anti-aliased input images. This might by done by rendering some curves as soft edges rather than sharp contours. A whole new interesting topic would be to look into *temporal* upsampling of animated pixel art images. If we magnify from a tiny input to HD resolution, locations are quite quantized which might result in "jumpy" animations. Since many modern display devices are operating at a much higher refresh rate than earlier hardware, one could improve the animations by generating intermediate frames.

Acknowledgements

We thank Holger Winnemöller for useful comments and help in creating some of the comparisons. This work was supported in part by the Israel Science Foundation founded by the Israel Academy of Sciences and Humanities.

References

ADOBE, INC., 2010. Adobe Illustrator CS5. http://www.adobe.com/products/illustrator/.

DE BOOR, C. 1978. *A Practical Guide to Splines*. Springer-Verlag.

ECK, M., DEROSE, T., DUCHAMP, T., HOPPE, H., LOUNSBERY, M., AND STUETZLE, W. 1995. Multiresolution analysis of arbitrary meshes. *Proceedings of SIGGRAPH '95*, 173–182.

FATTAL, R. 2007. Image upsampling via imposed edge statistics. *ACM Trans. Graph. 26*, 3, 95:1–95:8.

GLASNER, D., BAGON, S., AND IRANI, M. 2009. Super-resolution from a single image. In *Proc. ICCV*, IEEE.

HORMANN, K. 2001. *Theory and Applications of Parameterizing Triangulations*. PhD thesis, University of Erlangen.

JESCHKE, S., CLINE, D., AND WONKA, P. 2009. A GPU Laplacian solver for diffusion curves and Poisson image editing. *ACM Trans. Graph. 28*, 5, 116:1–116:8.

KONG, T. Y., AND ROSENFELD, A., Eds. 1996. *Topological Algorithms for Digital Image Processing*. Elsevier Science Inc., New York, NY, USA.

LAI, Y.-K., HU, S.-M., AND MARTIN, R. R. 2009. Automatic and topology-preserving gradient mesh generation for image vectorization. *ACM Trans. Graph. 28*, 3, 85:1–85:8.

LECOT, G., AND LEVY, B. 2006. ARDECO: Automatic region detection and conversion. In *Proc. EGSR 2006*, 349–360.

LIAUW KIE FA, D., 2001. 2xSaI: The advanced 2x Scale and Interpolation engine. http://www.xs4all.nl/~vdnoort/emulation/2xsai/, retrieved May 2011.

MAZZOLENI, A., 2001. Scale2x. http://scale2x.sourceforge.net, retrieved May 2011.

NEHAB, D., AND HOPPE, H. 2008. Random-access rendering of general vector graphics. *ACM Trans. Graph. 27*, 5, 135:1–135:10.

ORZAN, A., BOUSSEAU, A., WINNEMÖLLER, H., BARLA, P., THOLLOT, J., AND SALESIN, D. 2008. Diffusion curves: a vector representation for smooth-shaded images. *ACM Trans. Graph. 27*, 3, 92:1–92:8.

SELINGER, P., 2003. Potrace: a polygon-based tracing algorithm. http://potrace.sourceforge.net, retrieved May 2011.

STEPIN, M., 2003. Demos & Docs – hq2x/hq3x/hq4x Magnification Filter. http://www.hiend3d.com/demos.html, retrieved May 2011.

VECTOR MAGIC, INC., 2010. Vector Magic. http://vectormagic.com.

WIKIPEDIA, 2011. Pixel art scaling algorithms. http://en.wikipedia.org/wiki/Pixel_art_scaling_algorithms, 15 April 2011.

WOLBERG, G. 1990. *Digital Image Warping*, 1st ed. IEEE Computer Society Press, Los Alamitos, CA, USA.

XIA, T., LIAO, B., AND YU, Y. 2009. Patch-based image vectorization with automatic curvilinear feature alignment. *ACM Trans. Graph. 28*, 5, 115:1–115:10.