User-Guided Vectorisation of Pixel Art through Spring Simulation

Marko Matušovič[1] **Amal D. Parakkat[1]** **Elmar Eisemann[1]**

[1]TU Delft

Abstract

This research builds upon a previous method of vectorisation of pixel art by pixel neighbour connecting and boundary energy minimisation. The downside of the previous method is the lack of user input throughout the process and possible divergence between the results and the artist's vision. The proposed method uses the first part of previous method to connect neighbour pixels and continues with a spring simulation of the boundaries. A process that can be heavily user-guided with adjusting the stiffness of the springs. Results show the proposed method to achieve higher variance in the results, possibly resulting in output images closer to the artist's vision. However, introduce lengthy process for the user that can be removed by providing higher level GUI.

1 Introduction

Sometimes seeing pixels is welcomed; So-called "Pixel art" is a popular type of digital art. It originated in early video games. The first sprites appeared in video games in the 1970s. The game *Space Invaders* contained black and white sprites and was released in 1978 [1]. Shortly after, the game *Super Mario Bros.* was one of the first games to feature colour sprites, it was released in 1985 [2]. Since then computers gained the capability to display high-resolution images, but pixel art is still used as an art form. A benefit of pixel art is that it can be produced quickly and only with few resources [3]. Pixel art has become a distinct style and is still relevant in the present.

Raster images are described by a two-dimensional grid with pre-defined dimensions. Every cell in the grid holds information about the colour in that area, this cell is also called a pixel and is usually a square. In most cases the pixels of an image are not visible, however, when a raster image is zoomed in substantially, the individual pixels can become visible.

The counterpart to raster images are vector images. Vector images are described by a set of shapes and curves. Unlike raster images, they have no pre-defined resolution. Vector images can be shown at any resolution, and because a mathematical curve can be zoomed in infinitely, vector images remain sharp at all scales. However, making vector art requires more skill and effort.

The two premises: simplicity to create pixel art and beauty of vector art, creates a need for a method of a fast and low effort conversion of raster images into vector images. There are many previous methods of pixel art vectorisation, that perform well on their own. However, methods for vectorisation of pixel art accept no user input throughout the conversion. Given the base of art, this vector output might not be equal to the one the artist envisioned. If that is the case, the artist must then edit the final vector image to achieve a fully satisfactory image. Therefore, there is a need for a pixel art vectorisation method, that can be guided by the user.

The inspiration for smoothening out the sharp edges of pixel art came from soft body simulation. Soft bodies tend to be round. The inner forces pull on the surface and smoothen it. If the shapes in pixel art were modelled as soft bodies and simulated, they would also smooth out and create natural-looking curves.

The focus of this paper is to propose a new semi-automatic method for the vectorisation of pixel art, based on a previous method [4] and soft body simulation. The research question is:

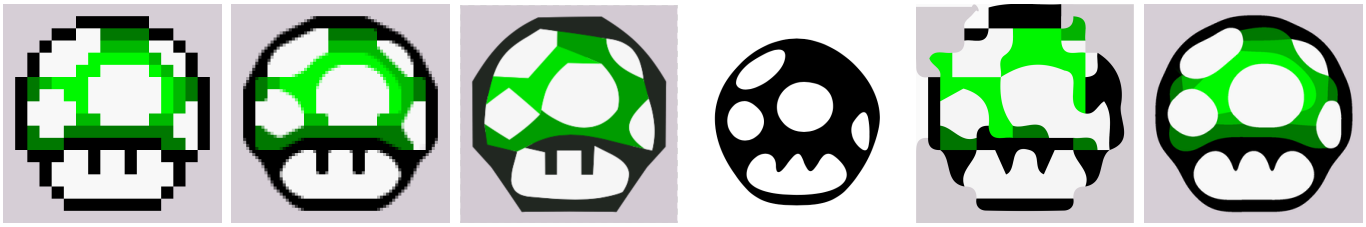
Can guided vectorisation of pixel art through spring simulation produce high-quality vector art?

High-quality vector art is defined as an image in vector format, whose curves appears smooth, but keeps sharp corners. The research question will be answered through a series of related sub-questions. How to structure the spring architecture? How satisfactory are the results with no user input? And what controls help artists achieve their results?

2 Related Work

Many previous methods achieve sharper details of pixel art images. Vectorisation is not the only way of achieving this result, the earlier methods include upscaling, or super-resolution. A process of increasing the resolution of raster images. This section will go over some of the most successful methods of upscaling and vectorisation.

Image Resampling The most usual method of achieving a different resolution of an original image is resampling. In this case, the focus is on higher resolution.



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

Figure 1: Showcase of previous methods. (a) Nearest neighbour (b) hq4x (c) vector magic (d) vectorization.org (f) Adobe Illustrator 2020 auto-trace (g) Depixelizing Pixel Art

The three most common approaches are presented here: nearest neighbour, bilinear interpolation, bicubic interpolation. Nearest neighbour looks up the nearest pixel from the original image and takes its colour value. This method keeps the colour palette intact, on the contrary, interpolation methods might result in pixels with colours not present in the original image. These methods interpolate between four adjacent pixels to produce an estimate of what a new pixel might be at that location. Bilinear interpolation uses a linear transition, bicubic interpolation uses 2nd order polynomial to calculate the new colour. In practice, interpolation is usually not used for pixel art as it results in blurry images, which is not desired appeal of pixel art. Nearest neighbour is the only of these methods used for pixel art as it preserves the hard edges.

hq4x In contrast to the methods in the previous paragraph, there are also upsampling algorithms developed for the sole purpose of upsampling pixel art. Among others, `hqx` [5] is a family of upsampling algorithms: `hq2x`, `hq3x` and `hq4x`. For every pixel, the similarity is determined with all 8 adjacent pixels. Then, by pre-determined rules, a pattern is applied to the pixel. Since this algorithm only looks at a `3x3` neighbourhood at most, it does not take into account what the image represents at a bigger scale. Moreover, `hqx` [5] requires aliased input and produces anti-aliased output. Therefore, it cannot be applied multiple times in succession and the maximum upscaling factor is `4x`.

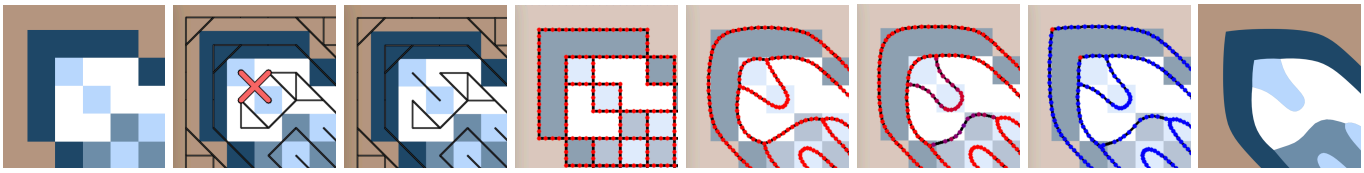
Commercial methods There are multiple commercial solutions for vectorisation. The source code is not available. However, the results can be obtained and compared. The website vectorization.org [6] provides free vectorisation tool. The downside is that it only works with black and white images. Adobe Illustrator has a built-in vectorisation feature: auto-trace [7]. It supports multicolour images and has a broad use. Another solution is Vector magic [8], it is not optimised for pixel art. None of the commercial solutions are optimised for pixel art and therefore usually underperform.

Depixelizing Pixel Art A novel method of vectorisation of pixel art was proposed in the paper *Depixelizing Pixel Art* [4]. Similarly to `hqx` [5] it compares all pixels to their neighbours and determines connections, according to these connections the pixels are reshaped. The boundaries are determined and through energy minimisation, the curves are smoothened. The method also finds automatically areas that should be sharper and areas where round edges are desired. Moreover, it also focuses on deciding which lines appear continuous in `T` and `X` crossings. Another feature of this method is that when similar colours appear in the image, the method tries to approximate a gradient through the region so that it appears smoother and can be merged into one shape.

Perception-driven semi-structured boundary vectorization This more recent work [9] is aimed at both low resolution and high-resolution clip art. Pixel art is therefore only a subset of the target group. This algorithm uses an ML classifier to detect key points in the input raster image, and then approximates a curve to connect them. The highlight of this method is the regularisation of the curves, this step replaces multiple similar shapes with one shape, such that the output vector image appears simple and visually pleasing.

Polyfit This method [10] builds upon the *Perception-driven semi-structured boundary vectorization* [9] algorithm. It is also a boundary vectorisation with the use of an ML classifier to detect edges. This method focuses on accuracy, simplicity, continuity and regularity of images to produce an output that is visually pleasing. In a user study they evaluated that *Polyfit*[10] better aligns with user expectations compared to both *Perception-driven semi-structured boundary vectorization* [9] and *Depixelizing Pixel Art* [4].

All algorithms optimised for pixel art already perform well, however, they allow for no user guidance through the process, this can be a downside as the result might not fully match the artist's vision. While both the *Perception-driven semi-structured boundary vectorization* [9] and *Polyfit* [10] usually outperform *Depixelizing Pixel Art* [4] both methods use machine learning in their processes. For the scope of this paper our method is based and compared directly to the method of *Depixelizing Pixel Art* [4].



(a) input (b) find links (c) de-cross (d) make nodes (e) simulate (f) smoothen (g) sharpen (h) export

Figure 2: Overview showing individual phases in the process. Phases (c), (f) and (g) accept user input.

3 Method

Our algorithm aims to convert a raster image of pixel art to a vector representation. The process can be split into four steps. The first step is called *Clustering Neighbours* and is shown in Figure 2b and 2c. In this step, the pixels are grouped into areas that seem like one shape and will be represented by one colour.It is based upon a previous method *Depixelizing Pixel Art* [4] mentioned in section 2. The second step is called *Generating Nodes* and is shown in Figure 2d. In this step, the borders around the grouped pixels are converted to nodes. The third step is called *Simulating Springs* and is shown in Figure 2e, 2f and 2g. In this step the nodes are simulated with their assigned spring stiffness, the user has an option to change the stiffness of springs and change how many iterations occur. The fourth step is called *Exporting to SVG* and is shown in Figure 2h. In this step, the nodes are exported to an SVG file as individual shapes.

3.1 Clustering Neighbours

In the first step of the method, pixels are segmented into groups. Every two adjacent pixels, even diagonally adjacent, are compared to each other and if their colour is evaluated to be similar, a link between them is stored. A link is defined as a set of two adjacent pixels. Links can be directly adjacent, (horizontal and vertical) and indirectly adjacent (diagonal). A problem arises when two diagonal links cross through each other. A cross is defined as a set of two links that intersect. All crosses must be disambiguated and one of the links must be removed. The user can solve the crosses automatically or they can decide which link to remove manually. After all crosses are simplified to one diagonal link, groups are formed. A group is defined as a set of pixels connected by links. Two groups are mutually exclusive, they do not share any pixels, and there are no links between them. This is the reason why all crosses must be removed. The algorithm advances to the next step only if there are no crosses present.

Colour comparison To avoid extensively user input, the algorithm determines which pixels are initially connected based on the similarity of the colour of adjacent pixels. This comparison is evaluated as a distance between the colours against a threshold. The distance metric is evaluated in HSV colour space. RGB (Red, Green, Blue) colour space is the most common space, it is often used when displaying images on computer screens, as pixels on screens are usually constructed from red, green and blue channels. In contrast, HSV (Hue, Saturation, Value) is closer to how people perceive colour [11]. In comparison to RGB, differences in HSV are perceived as linear by human observers [12].

Reconnecting Neighbours After the initial links are created with colour matching, the user is asked to remove all crosses, if any are present, and optionally create new links or remove existing links. This step allows the user to determine which pixel groups are connected, and how exactly they connect. The user interface consists of two windows. The main window, shown in Figure 3, contains the image with the links and crosses. By using the left and right mouse button to click on two adjacent pixels, the user can create or remove links. The secondary window, shown in Figure 4, contains a guide. If there are no crosses the user can choose to advance to the next step.

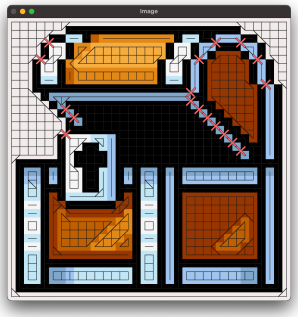


Figure 3: The main user interface windows for the first step. Links are shown black, crosses are shown red.

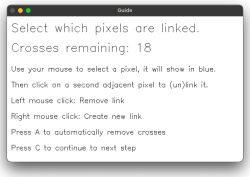


Figure 4: The helper window for the first step.

Automatic Cross Removal To alleviate some work from the user, the automatic cross disambiguation re-

moves most crosses. The algorithm is based on background - foreground segmentation. The foreground element occurs more rarely and the background is dominant [13]. In a given region around the cross, the pixels are bucketed into different groups. Buckets with fewer pixels represent the foreground. The link representing background is removed as it is preferred to keep foreground elements connected [4].

3.2 Generating Nodes

The second step generates nodes that are the base of the architecture that will be simulated. First, the borders of every group are determined. A border segment is defined as an edge of two directly adjacent pixels from two different groups. This is done by checking if two directly adjacent pixels are in the same group, if not, a border segment between them is created. Nodes are placed on each border segment in uniform distances, starting and ending at the corners. The number of nodes along a border is determined by a global constant (`RESOLUTION`). More nodes increase the resolution of the output vector image, fewer nodes shorten the runtime. A node is the smallest element that interacts with other nodes through springs, this behaviour is described in Subsection 3.3.

Figure 5: A group of a single pixel with nodes around it. Edge nodes are blue, corner nodes are red.

Connecting Nodes The nodes generated along a border lie either on a corner of a pixel or on an edge of a pixel. The two types are shown in Figure 5. The nodes along the edge of a pixel are surrounded by two pixels. The nodes are linked to each other and make a chain. Nodes placed on the corner of a pixel are surrounded by four pixels. Depending on how these pixels are linked and grouped, nodes can be connected to two, three or four other nodes, alternatively there can be two nodes in the same position each connected to two other nodes. All situations are shown in Figure 6.

If all four surrounding pixels belong to different groups, a corner node is connected to all four edge nodes, see Figure 6.a.

If the surrounding pixels belong to two groups, the corner node is connected to only two edge nodes, see Figure 6.b for a situation with two pixels in each group. Note that they must always be horizontally or vertically continuous, as two diagonally connected groups would make a cross and all crosses were removed in the previous step.

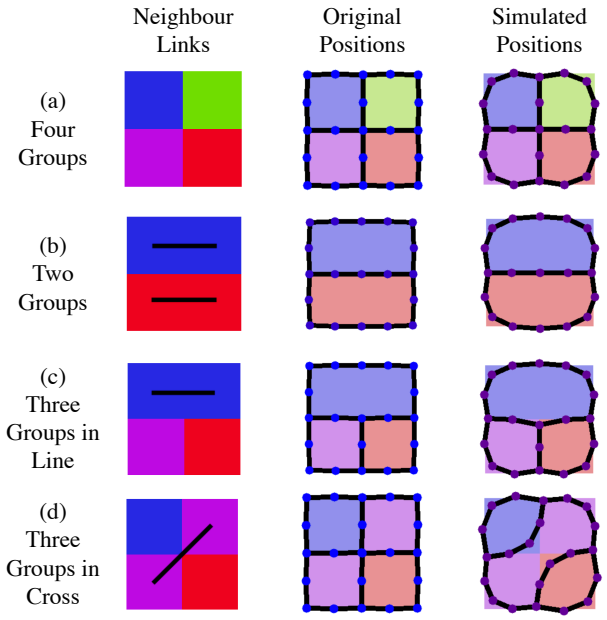


Figure 6: Different ways of connecting edge nodes to corner nodes by the number of groups.

If the surrounding pixels belong to three groups, two situations can occur, either the link does not go through the corner node, in which case the link is vertical or horizontal (see Figure 6.c), or the link goes through the corner node and the link is diagonal (see Figure 6.d). In the latter case, the two diagonally connected pixels must share some area and there cannot be any border between them. This is solved by placing another corner node at the same position. Both of them are connected to two edge nodes forming an `L` shape. The nodes are connected in such a way that the connections do not intersect with the pixel link.

At the end of this step, every group is enclosed by a path of nodes. The model contains information about all nodes, how they are connected and which nodes are associated with which groups.

3.3 Simulating Springs

In this step, the nodes are simulated, they undergo a transition to a minimum energy state, in which the paths appear smooth. A node is connected by a spring to all its neighbours, and also to its original position. There are three types of forces acting on a node, spring force from all its neighbours, spring force from the origin, and an area force. A node contains information about its original position, its current position, neighbouring nodes, associated area, neighbour spring stiffness and origin spring stiffness. This section explains the individual forces, their importance and interaction.

Forces Explained The springs between a node and its neighbouring nodes act as a pulling force. Tension in a path contract it to the shortest path, this makes it straighten and results in a smooth curve. The spring from a node to its origin is there to prevent the node from deviating too much from its origin. Additionally, every pixel group encloses an area. This area is calculated every iteration and a force is applied to all nodes of the path surrounding it trying to keep it at the same volume over time.

Assigning Spring Stiffness All nodes are initialised with preset spring stiffness. The neighbour spring stiffness is the same for all nodes and is determined by a global value (`NEIGHBOUR_SPRING_STIFFNESS`). The origin spring stiffness varies along the border. It is the highest at the centre of the edge and gets lower linearly towards the corner.

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

Equation 1 shows the formula for calculating the origin spring stiffness K_O. Where K_{OE} is the stiffness at the centre of an edge (`ORIGIN_SPRING_STIFFNESS_EDGE`), K_{OC} is the stiffness at the corner (`ORIGIN_SPRING_STIFFNESS_CORNER`), i is the index position of a node along a border, N is the number of nodes along a border (`RESOLUTION`). Figure 7 shows the plot of this function for $K_{OC} < 0$. Consider a diagonal line made of pixels, a strong force on the corners causes a staircase effect to show. This function was chosen as it lowers the force on the corners of pixels, hence reducing the staircase effect.

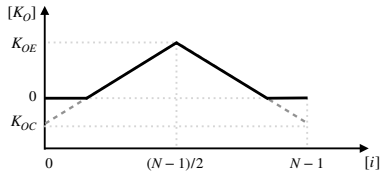


Figure 7: Plot of the origin spring stiffness function.

Calculating Area Force The area is represented as a closed path around a pixel group. Every group has always one area associated with it. This area is essentially an irregular polygon with known vertex positions. The area is calculated with Gauss's area formula, also called shoelace formula [14]. The initial area is stored and every iteration the current area is calculated. Moreover, every area also contains information about its centre position. This position is calculated as a centre of mass, an average of all node positions.

$$\vec{F}_A = \left(\vec{P}_N - \vec{P}_{AC}\right) \cdot \left(1 - \sqrt{\frac{A}{A_0}}\right) \tag{2}$$

The area force acting upon a node is determined by a vector from the centre of the area to the current node position scaled by a value correlating to the fraction of the area. This equation is shown in Equation 2, where for a given node and an area, $\vec{P}_N$ is the current position of the node, $\vec{P}_{AC}$ is the position of the area centre, A is the current area, A_0 is the initial area and $\vec{F}_A$ is the resultant force to be applied to the node. The square root is present to compensate for the correlation between an area and its radius. ($r^2 \sim A$) In this formula, the vector $(\vec{P}_N - \vec{P}_{AC})$ is similar to radius r. The resultant force $\vec{F}_A$ is added to the current position of the node $\vec{P}_N$.

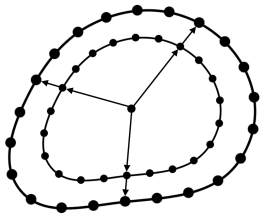


Figure 8: Area with a path at 80% scale that needs to be push back at 100%.

Calculating Neighbour Force Equation 3 shows the formula for calculating the neighbour spring force $\vec{F}_{Ni}$ for a given node N and its neighbour N_i.

$$\vec{F}_{Ni} = \left(\vec{P}_{Ni} - \vec{P}_N\right) \cdot K_{Ni} \tag{3}$$

In this formula, $\vec{P}_{Ni}$ is the position of the neighbour node and K_{ni} is the neighbour spring stiffness of the neighbour node. The difference between the position vectors results in a vector from the current node to the neighbour node, this vector is multiplied by the spring stiffness of the neighbour node to get the neighbour force acting on the node.

Calculating Origin Force Equation 4 shows the formula for calculating the origin spring force $\vec{F}_O$ for a given node N.

$$\vec{F}_O = \left(\vec{P}_O - \vec{P}_N\right) \cdot \left|\vec{P}_O - \vec{P}_N\right| \cdot K_O \tag{4}$$

In this formula $\vec{P}_O$ is the origin of the node. The difference between the position vectors results in a direction vector from the current position of the node to its origin, this vector is multiplied by its magnitude and the origin spring stiffness. The reason for multiplying by its magnitude is that the resultant force is relatively small for nodes close to their origin, however, it is large for nodes far from their origin. This is introduced to further reduce the staircase effect as discussed in *Assigning Spring Stiffness*.

One iteration Every iteration all forces for each node are calculated, scaled by a step size, and added to the current position of the node.

$$\vec{F} = \vec{F}_O + \sum_{i \in Ns} \vec{F}_{Ni} + \sum_{i \in As} \vec{F}_{Ai} \tag{5}$$

Equation 5 shows a formula for summing together all forces, where Ns is a set of all neighbour nodes and As is a set of all areas related to the node. An example of all forces acting on a node can be seen in Figure 9.

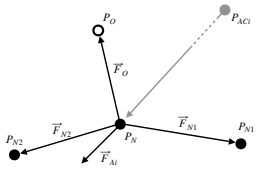
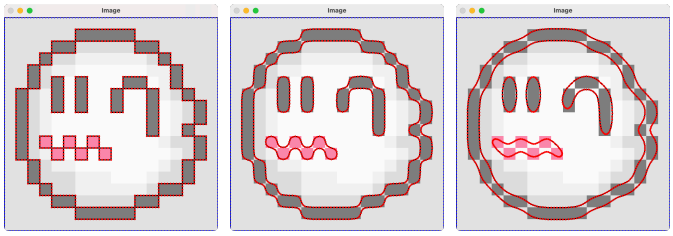


Figure 9: A node with two neighbours, showing all forces acting on it.

Simulation The simulation consists of many consecutive iterations. The step size of one iteration is determined by the magnitude of the largest force $\vec{F}_{\max}$. Equation 6 shows the formula for calculating the step size S. From the relation, the step size will always be at most $S_{\max}$ (`MAX_STEP_SIZE`) and in a case the magnitude of the largest force is high, the step size will be smaller.

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

The simulation has two progress modes, simulating for a fixed number of iteration steps and simulating until a minimal energy state is reached. An iteration step (`ITERATION_STEP`) is a value that dictates how many iterations of which step size can be executed. Every iteration the step size is summed. When the sum reaches the iteration step, the simulation ends. Figure 10 shows the difference between various iteration steps. The second progress mode compares the magnitude of $\vec{F}_{\max}$ to a threshold (`ITERATION_THRESHOLD`). The simulation reaches an end if the magnitude is below this threshold. It is up to the user to decide if they want the automatic approach, or if they find more or fewer iteration steps necessary.

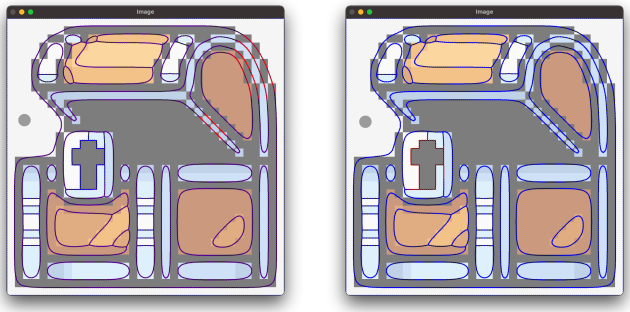


(a) I=0 (b) I=2 (c) I=20

Figure 10: Nodes after being simulated for various number of iteration steps

Adjusting Springs The user can adjust the individual spring stiffness to obtain a result more in line with their vision. The user can switch between editing the neighbour spring stiffness and the origin spring stiffness. The nodes are coloured by the spring stiffness, as seen in Figure 11.

A helper window, shown in Figure 12, is shown with instructions and information about the brush type, size and strength.



(a) neighbour spring stiffness (b) origin spring stiffness

Figure 11: The main user interface window for the second step. The two figures show various types of springs.

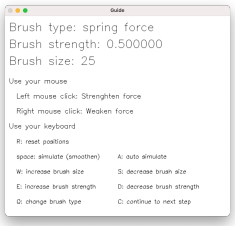


Figure 12: The helper window for the second step.

3.4 Exporting to SVG

In the last step, the simulated paths are stored in an SVG format as n-sided polygon, where n is the number of nodes enclosing an area. Areas are coloured by the average colour of a group.

Evaluating Z Order Most of the areas are non-intersecting, however, if an area is fully enclosed by another area, the outer area will cover the inner area. To prevent this from happening, the outer area is drawn first, then the inner area is drawn on top of it. The order of drawing is determined by the size of the areas, as an inner area is always strictly smaller than the outer area.

Colour Averaging Since the user gets a choice to connect pixels that might have different colours, the colour of the resulting group is not guaranteed to be the same. The resulting colour of the group is calculated as an average of colours of all pixels in the group. Each colour channel is averaged individually over all pixels in a group. The RGB colour space is used, as this is the input colour space and using other colour spaces does not provide a significant advantage. The result is a colour best estimating all pixels in a group.

4 Implementation

The method was implemented in C++ using OpenCV to read the image and provide GUI. Table 1 shows the time measured for Figure 13.a when skipping all user input. The algorithm was run on MacBook Pro 13" 2018 with 2.3 GHz Quad-Core Intel Core i5 CPU, Intel Iris Plus Graphics 655 1536 MB GPU and 16 GB 2133 MHz LPDDR3 RAM. Table 2 shows the list of all parameters used during the implementation.

Segment	Time	Percentage
Determine Neighbours	0.2 ms	0.03%
Find Crosses	1.5 ms	0.22%
Make Groups	0.5 ms	0.08%
Find Borders	0.7 ms	0.11%
Connect Paths	16.2 ms	2.45%
Find Areas	366.6 ms	55.57%
Simulate	272.5 ms	41.31%
Export to SVG	1.6 ms	0.24%
Total	287.5 ms	100.00%

Table 1: Timing of the algorithm on Figure 13.a, per individual segment, with user input ignored.

Parameter	Value
`RESOLUTION`	5
`NEIGHBOUR_SPRING_STIFFNESS`	1.5
`ORIGIN_SPRING_STIFFNESS_EDGE`	0.2
`ORIGIN_SPRING_STIFFNESS_CORNER`	-0.1
`MAX_STEP_SIZE`	0.1
`ITERATION_STEP`	0.5
`ITERATION_THRESHOLD`	0.03

Table 2: Parameters used in implementation.

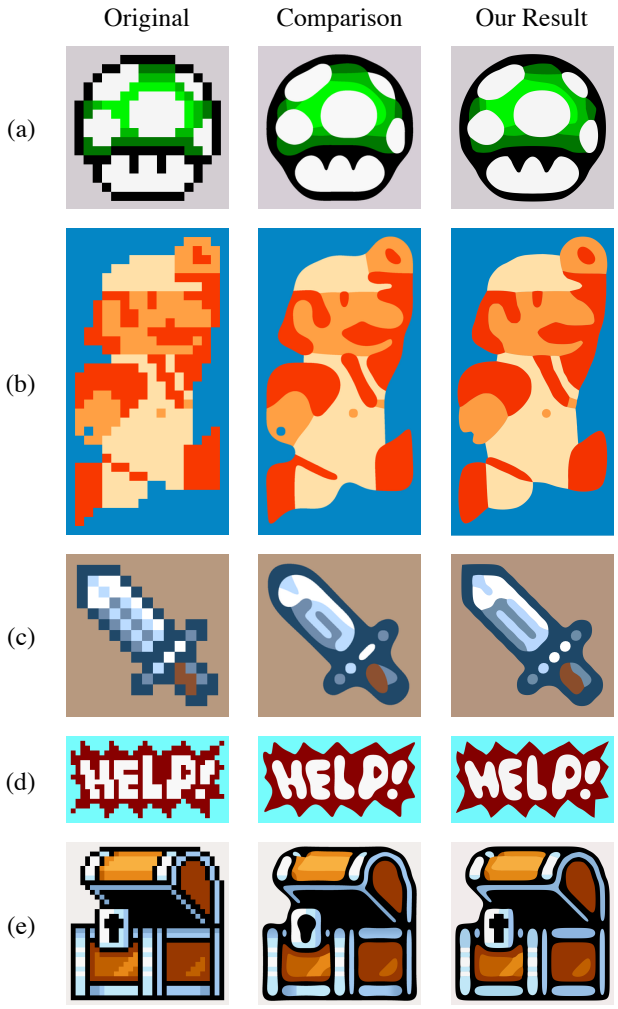


Figure 13: Our results with comparison to the original and results from *Depixelizing Pixel Art* [4].

5 Results

Our algorithm was applied to a selected batch of images from *Depixelizing Pixel Art* [4] and compared to their results. Figure 13 and Figure 14 show the original input, comparison results from *Depixelizing Pixel Art* [4] and our results. Additionally, Figure 14 also shows a second variation of our results, the difference was achieved with various user input. More results can be found in the appendix. This section also highlights the improvements of our method.

Match to Base Method Our method is based on that of *Depixelizing Pixel Art* [4], therefore it is critical for our method to produce no worse results. Consider Figure 13, our results look similar to the comparison results. It shows that with our method artist can achieve almost identical results. The only difference being a gradient through an area of similar colour, this feature was omitted in this paper.

More Options Artists using our method can achieve various results from the same input. Since our method works with user input, artists can decide and guide the process until the resulting image better represents their vision. In Figure 13.a in the comparison image, most white areas appear round except one on the top left. In our result, this area is round too. In Figure 13.b the comparison result has a few features which might not appear natural, in our result these were changed: the upper hand is not connected with the hat but rather with the sleeve, the lower hand does not have a hole in it, the shoe is disconnected from the red area on the pants.

Sharp Corners Our method can produce results with sharp corners. Consider Figure 13.c, in our result the tip of the sword is pointy. In Figure 13.e the keyhole on the chest keeps its shape and its sharp corners.

Different Styles Our method provides the possibility for artists to come up with various styles. Figure 14 shows

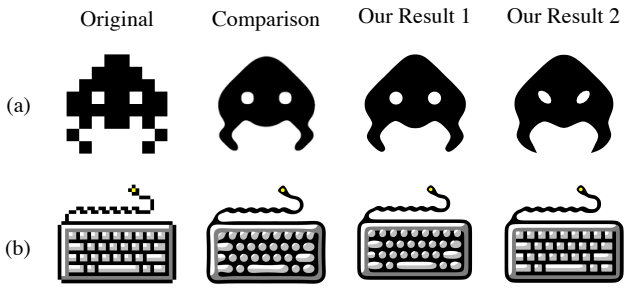


Figure 14: Two styles of our results with comparison to the original and results from *Depixelizing Pixel Art* [4].

an alternative styles of results. Figure 14.a shows an invader with prolonged eyes and sharp tentacles. Figure 14 shows a keyboard with square keys and round shades inside of them. Note that the comparison result for the keyboard mixes square keys in the top row with round keys in the middle rows. Our results can achieve both square and round keys and give the choice to the user.

6 Responsible Research

This section will go over the main aspects of responsible research related to this paper, namely: reproducibility and disclosure of data.

Reproducibility The method was implemented in C++, the code is available by request to the responsible professor. Results with no user input will be identical. Since the application works with user input throughout the process, the results can differ. To reproduce follow the instructions for human interference as described in section 3 and the user annotations in the appendix.

Disclosure of Data The method proposed in this paper builds upon a previous method, therefore the results are compared to the previous method. A selected batch is shown in the results section. A full comparison of all results can be found in the appendix. The intended use of the application is for artists to quickly change pixel art to a sharper representation in vector format. Additional purpose can be for computer games, however as this method works best with user input, it must be pre-processed.

7 Discussion

This section will discuss the relevance of this paper to the present research and its contributions. It will also address the limitations and outline the potential future work to be researched.

Relevance and Contribution A semi-automatic vectorisation of pixel art with the ability to guide the method can be a great addition to any digital artist's tool. Another use is for computer games, there are many computer games with pixel art. The sprites could be converted to vector art to give a game a modern feel. Moreover, while it is not the intended use, there are many old galleries with low-resolution raster images which would benefit from vectorisation as the converted vector images could be used where higher resolution is required. A new feature our method can achieve is a morph animation from the original pixel art to the output vector image. Simulating the nodes from the origin and taking a snapshot of each iteration, produces an animation.

Limitations Due to the time restriction of the research project this paper was a part of, not all features were pursued. The nodes in the output SVG image are connected by straight lines and are therefore visible. Higher `RESOLUTION` parameter helps hide them. However, there are different downsides to high node count. Since the SVG stores positions of all nodes, the file size tends to be quite large. A significant limitation is the lack of the gradient feature of the previous method [4], our method compensates by averaging the colours. Another limitation is the lack of advanced GUI. Our GUI provides the ability to edit the spring stiffness of individual nodes, but sometimes the process is lengthy. Moreover, in some cases our method relies on some user input and cannot be fully automatic, this makes the process even longer.

Future work The next steps for this research are to convert the high number of nodes into a smaller number of curves, such as Bézier curves or B-splines. Moreover, the curves can be replaced with standard shapes such as circles or boxes. Additionally, the method would benefit from more automatic features, which would pre-process the spring stiffness, further simplifying the artists work. Further, the UI can be improved to provide editing possibilities for the shapes as a whole, the back-end implementation supports this but the GUI does not.

8 Conclusion

This paper builds upon an already efficient method of vectorisation of vector art and proposes an alternative solution to smoothing out the curves. The spring architecture with the neighbour and origin spring forces and area pressure force proves to be a reliable back-end with potential for further research. Our method accepts various user input and demonstrates the possibility to replicate the results of *Depixelizing Pixel Art* [4]. However, the implementation is lacking features to be able to operate fully automatic. With user input the results can better fit the artist's vision as our method provides more options to edit the shapes, resulting in a larger pool of attainable results. A user-guided vectorisation through spring simulation can produce high-quality pixel art.

References

[1] B. Wirtz, "Video games history: From magnavox odyssey to wii, there's no stopping the gaming industry." `https://web.archive.org/web/`

20210426234123/https://www.gamedesigning.org/gaming/history/, 2020. [Online; accessed 3-June-2021].

[2] C. Plante, "When was super mario bros. released in the us? nobody knows!." http://web.archive.org/web/20201109022733/https://www.theverge.com/2015/9/14/9324833/super-mario-brothers-30th-anniversary-date, 2015. [Online; accessed 3-June-2021].

[3] D. Silber, *Pixel Art for Game Developers*. Boca Raton: CRC Press, 2015.

[4] J. Kopf and D. Lischinski, "Depixelizing pixel art," *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011)*, vol. 30, no. 4, pp. 99:1 – 99:8, 2011.

[5] M. Stepin, "Hqx." http://web.archive.org/web/20070717064839/www.hiend3d.com/hq4x.html, 2003. [Online; accessed 3-June-2021].

[6] Spikerog, "Vectorization.org." https://www.vectorization.org. [Online; accessed 3-June-2021].

[7] Adobe Inc., "Adobe illustrator." https://adobe.com/products/illustrator. version CC 2020 (24.1).

[8] C. L. Ventures, "Vector magic." http://vectormagic.com/. [Online; accessed 3-June-2021].

[9] S. Hoshyari, E. Alberto Dominici, A. Sheffer, N. Carr, D. Ceylan, Z. Wang, and I.-C. Shen, "Perception-driven semi-structured boundary vectorization," *ACM Transaction on Graphics*, vol. 37, no. 4, 2018.

[10] E. Alberto Dominici, N. Schertler, J. Griffin, S. Hoshyari, L. Sigal, and A. Sheffer, "Polyfit: Perception-aligned vectorization of raster clip-art via intermediate polygonal fitting," *ACM Transaction on Graphics*, vol. 39, no. 4, 2020.

[11] G. G. Marcu and S. Abe, "New HSL and HSV color spaces and applications," in *Imaging Sciences and Display Technologies* (J. Bares, C. T. Bartlett, P. A. Delabastita, J. L. Encarnacao, N. V. Tabiryan, P. E. Trahanias, and A. R. Weeks, eds.), vol. 2949, pp. 252 – 263, International Society for Optics and Photonics, SPIE, 1997.

[12] F. Garcia-Lamont, J. Cervantes, A. López, and L. Rodriguez, "Segmentation of images by color features: A survey," *Neurocomputing*, vol. 292, pp. 1–27, 2018.

[13] H. Garain, Utpal, Paquet, Thierry, "On foreground - background separation in low quality document images," *International Journal of Document Analysis and Recognition (IJDAR)*, vol. 8, p. 47, Feb 2006.

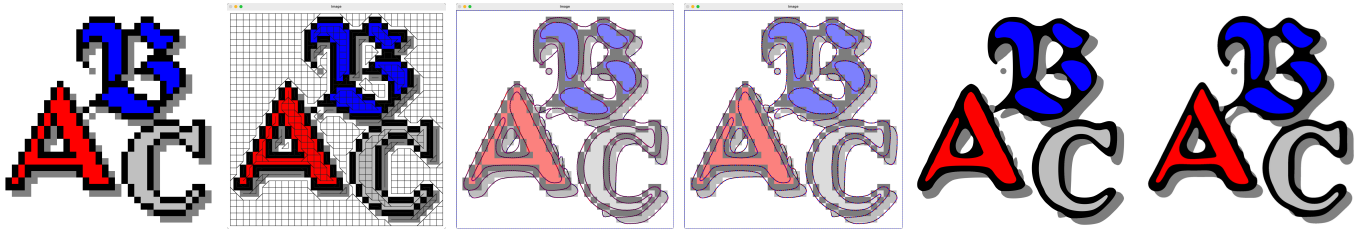
[14] W. Lim, "Shoelace formula: Connecting the area of a polygon and vector cross product," *Mathematics Teacher*, vol. 110, pp. 631–636, 04 2017.

Appendix

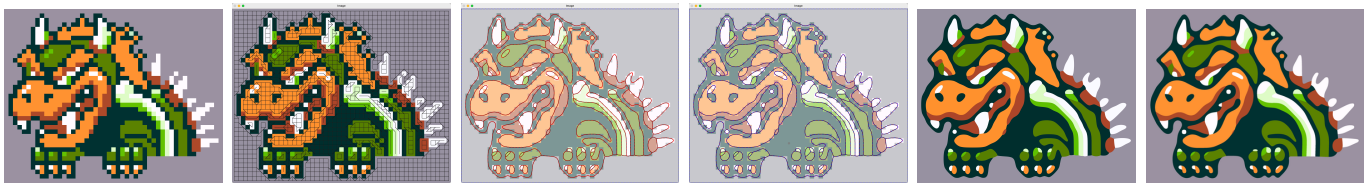
All Results with User Annotations

Input Image	Neighbour Links	Neighbour Spring Strength	Origin Spring Strength	Our Result	Depixelizing Pixel Art Result

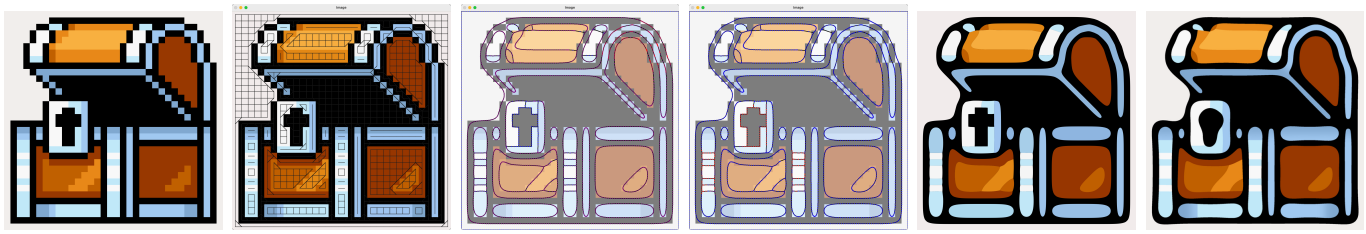
ABC



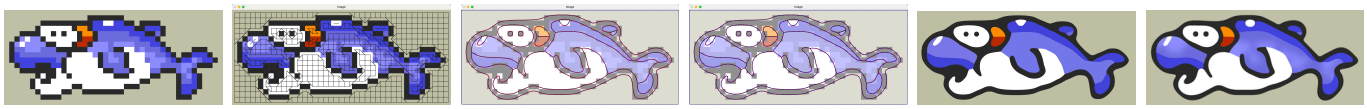
Bowser



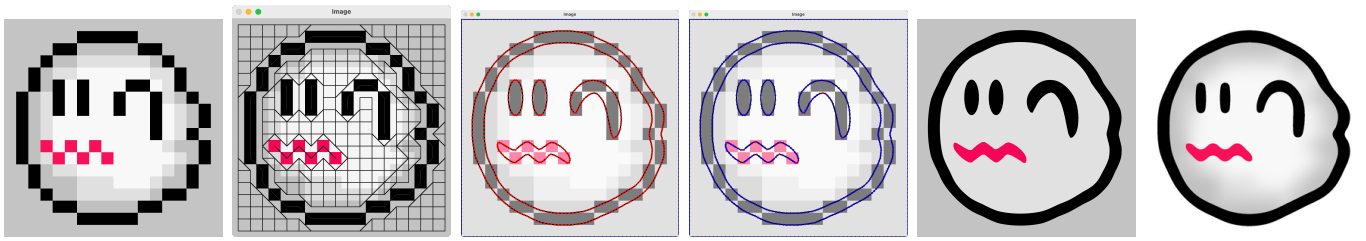
Chest



Dolphin

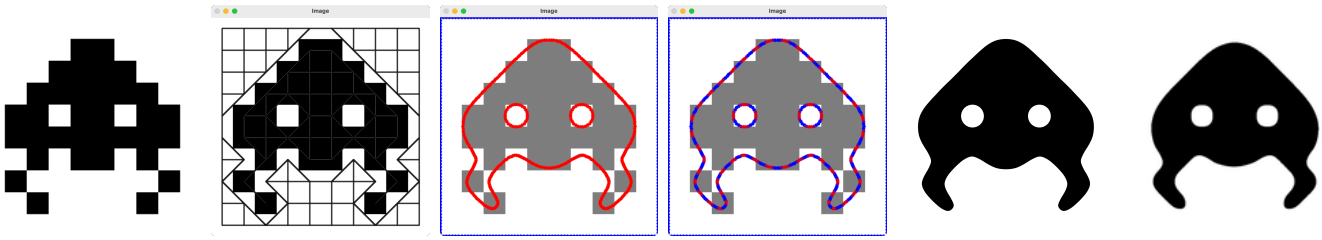


Ghost

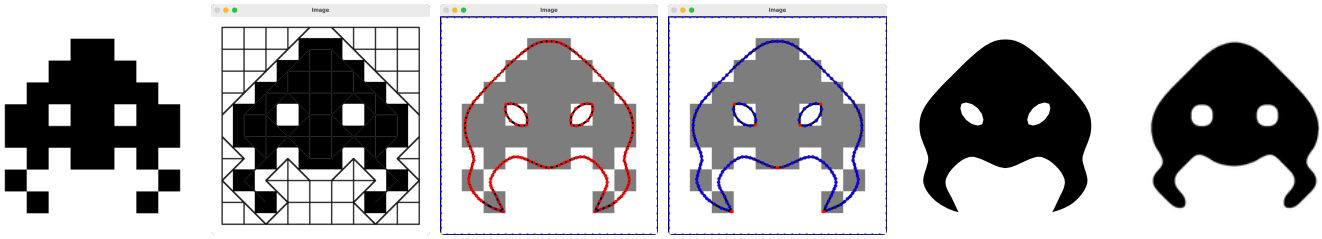


Input Image	Neighbour Links	Neighbour Spring Strength	Origin Spring Strength	Our Result	Depixelizing Pixel Art Result

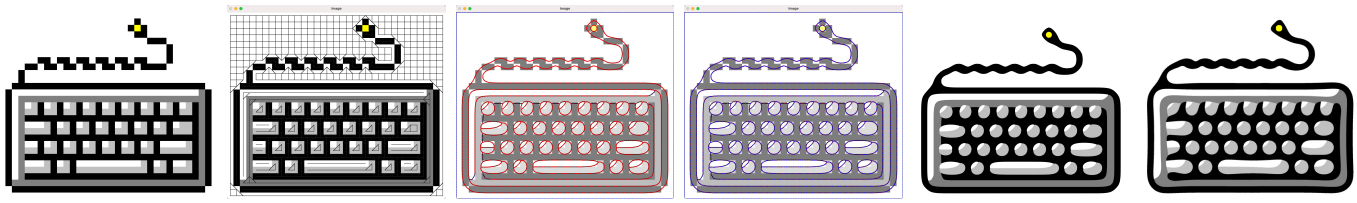
Invader: Style 1



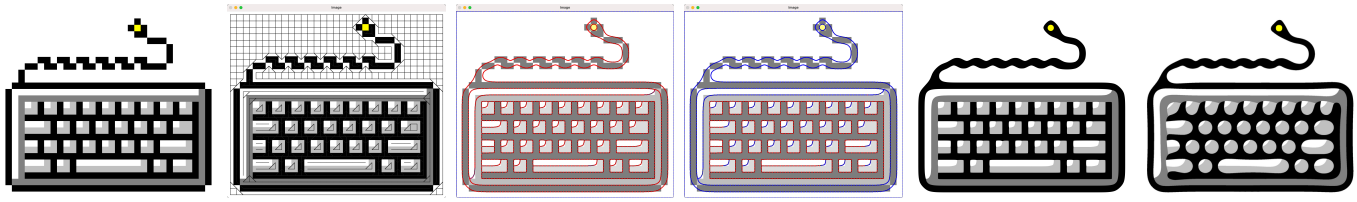
Invader: Style 2



Keyboard: Style 1



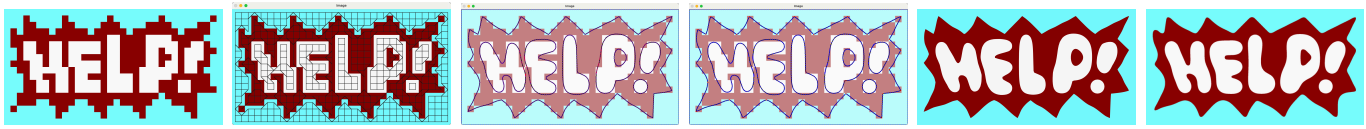
Keyboard: Style 2



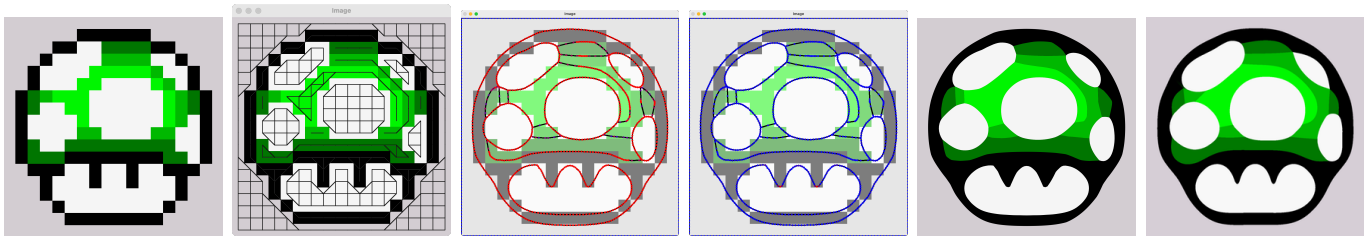
Mario

Input Image	Neighbour Links	Neighbour Spring Strength	Origin Spring Strength	Our Result	Depixelizing Pixel Art Result

Help Sign

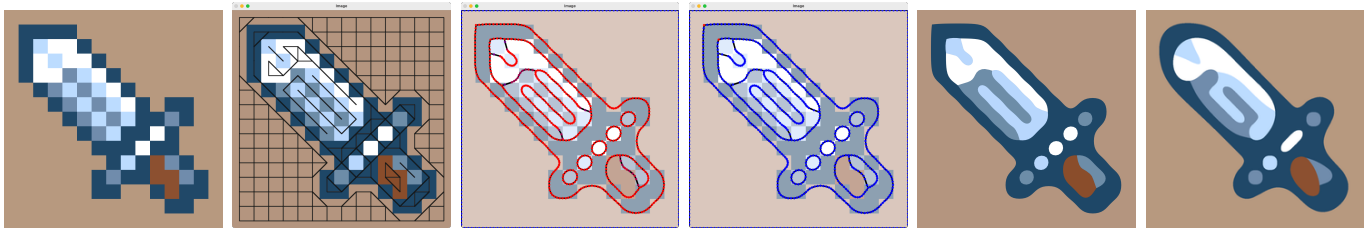


Mushroom



Salamando

Sword



Yoshi