STAG: Smart Tools and Applications in Graphics (2021)
P. Frosini, D. Giorgi, S. Melzi, and E. Rodolà (Editors)

Efficient Image Vectorisation Using Mesh Colours

G. J. Hettinga[1], J. Echevarria[2], J. Kosinka[1]

[1]Bernoulli Institute, University of Groningen, the Netherlands
[2] Adobe Research, San Jose, CA, USA

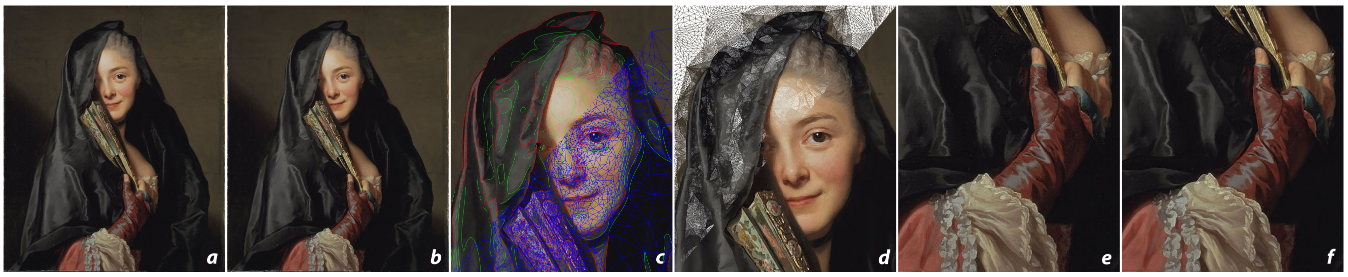

Figure 1: *An example input image (a) and our vectorised result (b). Please, zoom in to see full details. Our method follows an efficient and controllable pipeline where we initially compute hard (red) and soft (green) image features (c, top). These features are then used to build a curved triangular 2D mesh (c, bottom), where each triangle is equipped with mesh colours (d, top) that can be rendered efficiently in real-time (d, bottom). Insets (e, f), coming from (a, b), show how our method keeps the sharpness around hard features (like the folds), while interpolating colour smoothly everywhere else. Results of this quality can be achieved in seconds with the proposed method. Please, see the accompanying video for a deeper look into this example.*

Abstract

Image vectorisation methods proposed in the past have not seen wide adoption due to performance, quality, controllability, and/or generality issues. We present a vectorisation method that uses mesh colours *as a vector primitive for image vectorisation. We show that mesh colours have clear benefits for rendering performance and texture detail. Due to their flexibility, they also enable a simplified and more efficient generation of meshes of curved triangular patches, which are in our case constrained by our image feature extraction algorithm. The proposed method follows a standard pipeline where each step is efficient and controllable, leading to results that compare favourably with those from previous work. We show results over a variety of input images including photos, drawings, paintings, designs, and cartoons and also devise a user-guided vectorisation variant.*

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Image processing;***

1. Introduction

Image vectorisation is the process of converting a bitmap (raster) image into a vector image. Vector images define an image as a collection of primitives, such as lines, curves, or more elaborate geometric objects. Vector graphics are key in many disciplines, such as graphics and web design, or textile and printing industries, due to their ability to display an image at arbitrary resolutions without loss of quality. The manual vectorisation of a raster image is a painstaking process, especially for highly detailed input like photographs, which requires expertise and immense amounts of time [YCZ*16].

Over the years, many approaches to automatic image vectorisation have been devised, using various primitives, the building blocks of vector images. However, they have not seen wide adoption given performance, quality or controlability issues, although popular commercial tools like Adobe's *Live Trace* [Ado19b] have turned their limitations into an artistic style of its own. Creating complex realistic vector graphics thus remains a challenge.

We propose a new image vectorisation method that excels at processing time, detail control, and rendering efficiency, while producing representations that have potential to be included in existing vector graphics authoring tools and are easily edited. Our method performs well over a wide variety of inputs, including natural images and stylised design graphics.

Our approach follows a standard pipeline of three main steps:

DOI: 10.2312/stag.20211484

image feature extraction, 2D mesh generation, and colour fitting/texture transfer. Each step has been designed with quality, performance, and control in mind, leveraging recent advances in mesh generation and texture representation. Additionally, using the proposed rendering method, our vectorised images can be rendered in real-time on a wide range of hardware and offer extra control over the level of detail for more constrained use cases.

We present an intuitive method for extracting image features that preserve sharp and soft details alike. Our mesh generation step faithfully traces and leverages those features in efficient curved triangular meshes. Our new colour fitting step automatically transfers texture information from the image to per-triangle *mesh colour* [YKH10] patches. We show results for a variety of inputs, which compare favourably to previous works in terms of image quality, performance, or both.

In summary, the main contributions of our method are:

- The introduction of mesh colours as an image vectorisation primitive, with an efficient strategy of automatic fitting;
- A novel way to extract texture information and soft features from the input image;
- Efficient automatic and user-guided image vectorisation.

We start with an overview of related work in vector graphics and image vectorisation (Section 2). Then we provide an overview of our vectorisation method (Section 3), which is followed by a detailed description of its stages: image feature extraction (Section 4), mesh generation (Section 5), and texture transfer (Section 6). To demonstrate the utility of our method, we show the results of applying our pipeline on several types of raster images, edits to our vector images, our user-guided vectorisation pipeline and compare with previous works (Section 7). Finally, we discuss our method before concluding the paper (Section 8).

2. Related Work

Solid Colours & Linear Gradients. Early attempts at vectorisation of images automatically partition the image into regions that can be represented reasonably well by flat, linear or quadratic gradients [LL06]. This yields stylised representations of the original image, which may be desirable in some applications. Vectorisation of natural images using only solid colours often requires a preliminary colour quantisation step to simplify detail [Ado19a; LLGR20], which affects the expressiveness and detail preservation of this approach. Moreover, interactive user guidance for such quantisation is often desired to preserve salient semantic boundaries between objects [XWLS17; RLB*14; FLB17]. Early attempts also include adaptively created image triangulations [DDI06].

Diffusion Curves. Following the paradigm of image creation based on its salient edges, diffusion curves [OBW*08] represent an image by a set of curves defining sharp transitions in colour. These colours are then diffused over the rest of the image. When used in vectorisation, image edges are detected and represented with smooth curves, and their colour is deduced from the underlying image. Diffusion curves have been extended in several ways to allow for more expressiveness and user control of the primitive itself, and thus also in the vectorisation process. This includes hierarchical diffusion curves [XSTN14], depth-aware image vectorisation [LJD*19], and other methods [DLS13; ZDZ17]. Although diffusion curves and their generalisations offer a powerful set of primitives, they tend to be expensive to evaluate as this involves solving large linear systems [OBW*08]. Advances in diffusion curves have lead to numerous solvers that try to do this in a smart way [JCW09; JCW11] or with raytracing [BLW11; PJS15]. Although Diffusion curves are an excellent way to represent images they have not seen wide adoption due to the complex nature of the solvers [BBG12].

Parametric Patches. The gradient mesh primitive, originally introduced in Adobe Illustrator [Ado19b], represents an image as a regular grid of bicubic patches [Ado06; SLWS07]. These bicubic patches are the result of interpolation of the colours and colour gradients assigned to the vertices of the (gradient) mesh. Early attempts at vectorisation with this primitive have used it in combination with (pre-)segmented regions with progressive subdivision of patches [PB06] or optimising meshes [LHM09]. A fully automatic pipeline based on a frame-field guided quadrangulation was proposed by Wei et al. [WZG*19].

Later, subdivision surfaces were used over triangular meshes [LHFY12; ZZW14] that were created using an elaborate pipeline of feature extraction, mesh generation as well as colour fitting for a fully automatic image vectorisation solution. However, subdivision surfaces are costly to evaluate and do not in general interpolate colours assigned to mesh vertices [VK18]. Gradient meshes and subdivision surfaces can model smooth regions of an image with high accuracy, but require many patches in regions with high-frequency changes in colour. Recently, other non-standard forms of the gradient mesh have been proposed to alleviate this problem [LKSD17; LJH13; BLHK18; WZG*19; BHEK21].

In [XLY09], thin-plate splines are used in combination with cubic Bézier triangles. The Bézier triangle mesh is generated by simplifying a pixel-dense triangle mesh, which is followed by optimising that mesh into Bézier triangles. Subsequently, thin-plate splines are fit on a per-triangle basis. Chen et al. [CLL*20] combine dense thin-plate splines with coarser gradient meshes that are generated from a rough manual segmentation of the image. However, thin-plate spline evaluation is costly, even though the authors provide an efficient kernel to evaluate the splines. They are also unable to preserve sharp features that are not preserved by the segmentation.

We also create Bézier triangles for our mesh, but directly from detected image edges. In contrast to prior art, we represent the colour inside each Bézier triangle as mesh colours [MSY19]. This allows for representing texture in a detailed way and provides a cheap, yet accurate representation of the original image. It is worth noting that recent work on new primitives for colour manipulation [SKFS20] presents similar triangular subdivisions, but the parametric shape of their colour distributions does not capture spatial texture detail.

Other Vectorisations. Finally, there is a related but different class of vectorisation that focuses on more abstract/stylised imagery such as *pixel art* [KL11; SMC*13], where aliased raster edges can be shape, texture or shading, making that inference the core of the

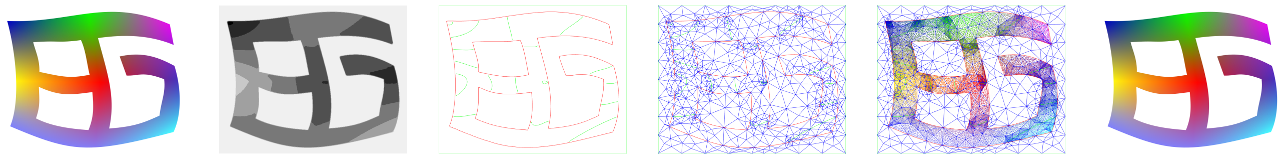

Figure 2: *An overview of the steps in our method. From left to right: The input raster image, banded greyscale image to extract soft edges from, extracted hard (red) and soft (green) image features, generated mesh, mesh colours with colours fitted, and final rasterised result.*

problem. Other recent works focus on perceptually-motivated vectorisations of uniform colour regions separated by sharp transitions [HDS*18; DSG*20]. Similarly, vectorisation of drawings and sketches [FLB16; NS19; EVA*20; SBBB20] focuses on inferring vector lines from latent lines perceived on the raster inputs.

3. Overview

Our method automatically converts an input raster image into a vector image. The image can have various content: we accept anything from natural images to design graphics. Although we want the vector image to accurately represent the input image, we still want the vector representation to be editable, that it can be rendered efficiently, and that it does not contain a large number of primitives. To that end, we extract image features that capture representative geometry, shading and texture. Once we vectorise them using spline curves, they should turn into intuitive handles for high level edits. These image features are also the constraints for our mesh generation step, where we are interested in a curved triangular mesh that follows them faithfully. We look for efficient mesh generation, with enough triangles to obtain a good topology to support detailed mesh colour patches, yet the representation should still be easily editable by the user for more local geometric edits. Next, we equip each triangle with a mesh colour patch. We do not intend to expose the mesh colours to the user for geometric edits. Thus they can be of higher resolution than the mesh triangles to retain as much texture detail as desired. Our new automatic texture transfer process then fits the colours from the input pixels to the mesh colours of the patches. Finally, we rasterise our vector images in real-time through tessellation shaders, whose level of detail can be controlled on the fly for excellent performance for visualisation, but also for detailed offline work.

Figure 2 visually depicts the steps in our pipeline. First, the main features of the image are extracted (Section 4). We then generate a mesh (Section 5), which is followed by colour fitting of mesh colours (Section 6). Figure 1 shows the same intermediate steps for a more complex input.

4. Feature Extraction

The key to a successful image vectorisation is to establish which features to preserve from the input raster image. Given that we aim for a universal method applicable to any input image, we cannot make any assumptions about their content. However, we define two types of features: *hard* and *soft* edges. A similar distinction between hard and soft edges was made earlier on by Lindeberg [Lin96] and Elder [EZ98] who describe differing blur scales to edges which are extracted using a scale-space approach. Their approach was later used in a vectorisation setting by Orzan et al. [OBW*08] to estimate blur scales for edges.

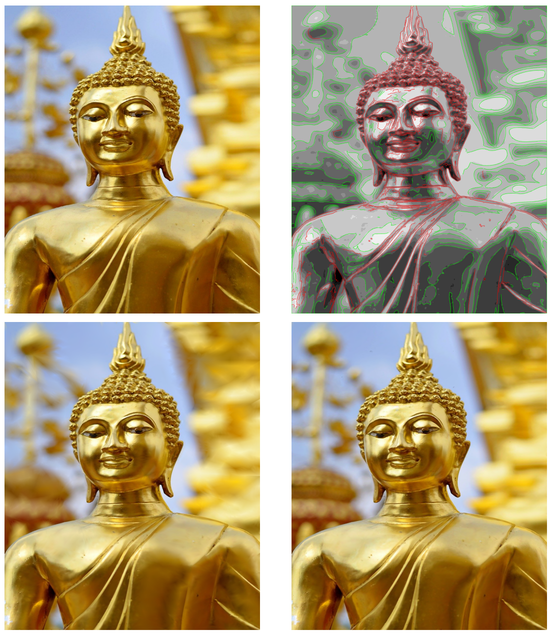

Figure 3: *Top row: Original image and the quantised greyscale image with the extracted hard (red) and soft (green) edges overlaid. Bottom row: Our vectorised version without using soft edges (left) and with soft edges (right). Note that soft edges help to capture more detail and to avoid artifacts on the reflections of the statue and in the background. Please, zoom in for a more detailed view.*

Hard edges come mainly from colour discontinuities that typically capture salient shapes, contours and textures of the elements in the image; and they should remain sharp on the vector image. On the other hand, we use soft edges to model smooth but complex colour transitions (e.g. shading). Edge detection is still an active topic after decades of research [MA09], with recent neural approaches that do an increasingly good job at inferring geometrical edges at object level [XT15; LCH*17; HZY*20]. However, these methods are not that well suited to surfacing progressive texture detail or complex colour transitions.

Edge Extraction. For simplicity and ease of control, and similarly to previous works [XLY09; CLL*20], we use the Canny edge detector [Can86]. Given the performance of the rest of our method, this choice provides interactive and intuitive control over the level of detail of the resulting vectorisation. We typically set the low and high thresholds to 15 and 100 in the range $[0,255]$, respectively. However, while Canny is good at detecting hard edges, it fails to pick up soft edges: lower thresholds lead to too much noise and/or unwanted texture detail. Thus, we propose a new simple procedure to extract soft edges to complement the hard ones found by Canny.

Upon closer inspection of the vectorisations obtained using only hard edges, we noticed that the missing soft edges we were interested in are typically orthogonal to the smooth colour gradients in the image. To expose those features, we quantise a greyscale version of the input image (20 levels by default), and trace the discontinuities from the resulting banding. This is similar in spirit to the iso-contours used to vectorise brushstrokes in [BDF14], but our soft edges go through extra processing before being traced differently.

Edge Filtering. Because hard and soft edges may overlap near areas where the image gradients change quickly, we filter the soft edges based on their distance to neighbouring hard edges, effectively removing them from areas where hard edges were already present. We do this by creating the distance transform of the previously extracted hard edges and using this as a filter for the extracted edges from the banded image. This filtering promotes the creation of sparser geometry later on in Section 5, as it removes bands that overlap with hard edges or are closely parallel to them. Figure 3 shows the result of quantising the greyscale image, and the subsequent soft features extracted from them (top right). These features help us better capture soft image details, such as all the blurry background elements in the input image (bottom row).

5. Mesh Generation

The goal of the mesh generation step is to create a mesh of cubic Bézier triangles that conform to the detected image features. We vectorise the detected edges by converting them to cubic Bézier splines, which are then used to drive the curved triangulation step.

Edge Vectorisation The extracted hard edges are traced and linked into pixel chains to be vectorised as cubic Bézier splines. These splines help us capture curves and remove the aliasing present on the hard edges, and enforce C^0 or G^1 continuity as needed. In the spirit of [Sch90], we progressively fit the splines to the chains by recursively fitting curves to each whole pixel chain. We keep each fit only if the maximum error distance between a polyline approximation of the Bézier curve and the pixel chain is half a pixel. If not, the pixel chain is split at the pixel with the largest distance, and new splines are fitted to the respective halves of the previous pixel chain, until every segment of the pixel chain is converted.

Soft edges are usually noisier and do not represent salient image features that need to be preserved as accurately as the hard edges. Therefore, we found an approximation is sufficient and we do not have to strictly enforce the start and end points of the spline piece to lie over the actual pixel-chain. Figure 3 (top right) shows the vectorised edges obtained from the input image. As can be seen, the hard edges are accurate, whereas the soft edges offer an approximation of the bands from the quantisation, without affecting the quality of the reconstruction. Hard and soft edges are kept separate to be handled differently in later stages of the pipeline.

Curved Triangulation. The soft and hard vectorised edges from the previous step are curved, so direct application of standard linear meshing techniques would cause degeneracies if supporting straight line segments crossed through the curved edges. One option could be to subdivide the curved segments until only accurate-enough linear elements remain, but this would quickly increase the number of faces in the subsequent meshing step. Instead, we choose to create a curved triangular mesh, as this allows us to keep a lower number of generated triangles while directly incorporating the curved edges. [XLY09] also use curved triangular domains, however they are the result of untangling an initial linear mesh through a nonlinear optimisation method. We generate our curved triangular meshes in a single and more efficient pass.

To generate such a mesh we follow the example of Mandad and Campen [MC20] based on guarding triangles to avoid intersections and employ standard constrained Delaunay triangulation. We speed up the mesh optimisation phase by inserting supporting vertices into the triangulation at regular intervals and only when the inserted position is some distance from the nearest feature. Alternatively, TriWild [HSG*19] could also be used to generate the geometry. After the triangulation step, we are left with a non-degenerate curved triangulation. This fixes the topology of the mesh. Next, we determine its geometry and parametrisation.

On each (curved) triangle, we construct a cubic Bézier triangle (see Figure 4, left) by using the control points of the curved edges as edge control points and converting straight supporting segments to cubic Bézier curves, which determines all the blue control points. This ensures that all vectorised (hard and soft) edges are exactly reproduced in the curved triangulation. To fix the parametrisation, we add the central control point (red in the figure) as the centroid of the edge control points of each triangle. We also keep track of which edges represent hard and soft features by tagging them. The straight segments generated by the triangulation step are always deemed to be soft.

6. Texture Transfer

As mentioned before, our goal is to use vector primitives able to handle varying texture detail, while being fast to evaluate. We propose the use of *mesh colours* [MSY19], a first in the context of image vectorisation, when not considering simple vertex colours and standard linear interpolation. Mesh colours [YKH10] are a convenient way of storing colour and texture information in complex meshes. Above all, no texture coordinates are needed, which removes the need for complex UV maps and unwrapping techniques. This makes it robust to further transformations and edits, especially when compared with parametric texture representations from previous vectorisation work. These properties make mesh colours a fitting mechanism for transferring texture information from a raster to a mesh. However, mesh colours have been used in conjunction with 3D painting tools [YLT19], and there are no standard ways of transferring texture information to mesh colours.

Mesh Colours. In our setup, we map triangular mesh colours to the cubic Bézier triangles that were generated in the previous step (Section 5). Figure 4, middle, shows a schematic view of mapped mesh colours. The resolution r handles the number $R = \frac{(r+1)(r+2)}{2}$ of mesh colour vertices $t_{\mathbf{i}}$ per patch, where $\mathbf{i} = (i, j, k)$, $i + j + k = r$ and $i, j, k \geq 0$. We follow the procedure of [MSY19] to interpolate mesh colour values over each patch. For clarity of presentation and

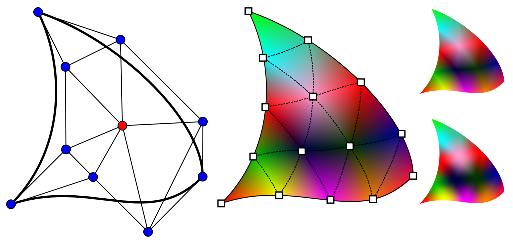

Figure 4: *Left: A cubic Bézier triangle with control points (the central one is in red). Middle: Resolution* 4 *mesh colour texture with* $(4+1)\cdot(4+2)/2 = 15$ *mesh colours mapped on the cubic Bézier triangle. Right: Linear (top) and quartic (bottom) interpolation.*

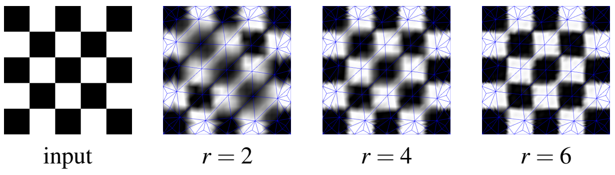

Figure 5: *From left to right: Input image and vectorisations using the same mesh but different resolution r of mesh colours. On purpose, the mesh does not capture the features of the input image correctly. Regardless, increasing r leads to increasingly better approximations of the input image.*

to prepare the ground for our proposed mesh colour fitting scheme (Section 6.1), in the following we detail the steps for evaluating a mesh colour texture on a triangular patch.

The barycentric coordinates $\phi = (u, v, w)$ with respect to a triangle $\triangle$ in the mesh are used to determine the three closest mesh colours: $t_{\mathbf{i}}$, $t_{\mathbf{j}}$, $t_{\mathbf{k}}$. These colours together determine a mesh colour (sub)triangle of $\triangle$. From the coordinates ϕ we determine the local barycentric coordinates $\overline{\phi}$ inside the mesh colour triangle. These local coordinates are subsequently used to either linearly or quartically interpolate between the three colour values of the mesh colour triangle. The former results in piece-wise linear C^0 colour interpolation over $\triangle$, and the latter in piece-wise quartic C^1 colour interpolation at the expense of increased computational cost. We choose to use the C^0 version as there is not a lot of difference (Figure 4, right column), except in cases where mesh colour values vary extremely. The resolution of each mesh colour texture can be changed per triangle [Yuk16], effectively adjusting the amount of texture detail that can be represented and the storage required for it. Figure 5 shows a simple example where a static mesh approximates the same image with different mesh colour resolutions. Higher mesh colour resolutions are able to approximate the input more clearly.

6.1. Mesh Colour Fitting

At this stage, the mesh geometry is already fully defined and mesh colours can be fitted to them. Ideally, the colour fitting process should be implemented as a global least-squares problem over all mesh colours of the entire mesh. This would automatically generate smooth colour transitions over triangle boundaries that are marked as smooth. However, this leads to a large system of equations that needs to be solved and would not be practical both with regards to memory and performance. Our approach simplifies the problem, by fitting each triangle individually and only afterwards smoothing mesh colours where appropriate.

We first fit mesh colours to each cubic Bézier triangle T parametrised over $\triangle$ separately. We sample each T in the mesh uniformly to obtain the pairs (p_i, ϕ_i) for every sample position p_i in the image with ϕ_i its barycentric coordinates in $\triangle$, i.e., $T(\phi_i) = p_i$. To effectively fit mesh colours, we need to ensure that the number of samples m satisfies $m > R$. Using the image position p_i of each evaluated pair, we look up the bilinearly interpolated colour value $I(p_i) = c_i$ in the input raster image I. Using ϕ_i, we determine the local barycentric coordinates $\overline{\phi}_i$ of p_i in its mesh colour (sub)triangle. We then minimise the following function on a per-triangle basis, used once per colour channel:

$$\min \sum_i \left(T^c(\overline{\phi}_i) - c_i\right)^2,$$

where T^c evaluates the colour corresponding to T. This is a standard least squares problem that we solve for the mesh colours of T. The matrix of the system can be reused for fitting triangles with the same mesh colour resolution r, since it is independent of the actual image positions and it uses only parametric positions (expressed in terms of ϕ), which are generated uniformly for each triangle. We efficiently perform this sample pair creation process in parallel for each triangle using GPU compute shaders.

To increase the efficiency of this colour fitting step even further, we adaptively use different mesh colour resolutions based on the size of each triangle with respect to the original image, as follows. We assume that during the feature extraction step most regions of highly varying colour are split into smaller ones, and that each resulting triangle after the mesh generation step lies in an area with small changes in colour. Therefore, the smaller the triangle, the lower resolution it needs to represent the textured area of the original image. We bin the triangles based on their pixel area in the original image into three different bins. These bins correspond to increasing resolution of mesh colours $r \in \{1, 2, 4\}$. This benefits performance as lower resolution textures require less samples to be generated, reducing CPU-GPU congestion and improving the speed of the fitting. Optionally, the resolution of the mesh colours can be increased for smaller patches to $r = 4$ by creating additional mesh colour vertices through linear interpolation of existing mesh colours, which allows unified rendering of triangles later on.

Offset Sampling. We found that bilinear sampling does not fetch correct colours when sampling close to hard edges, causing colour bleeding artefacts to appear on the triangles (Figure 6, top). Inspired by Liao et al. [LHFY12; OBW*08], we use a one-pixel padding around hard edges. When sampling at hard edges, the actual sampling is offset from the padded region. This ensures that sharp transitions on the raster image are preserved by sampling on the correct side of the edge (Figure 6, bottom). By offsetting the sampling we are able to create crisp edges without affecting the smooth interpolation on either side of the edge.

Colour Smoothing. After each individual triangle in the mesh has been fitted with its associated mesh colours, we can increase the fidelity of our vectorisation by smoothing mesh colours that are on

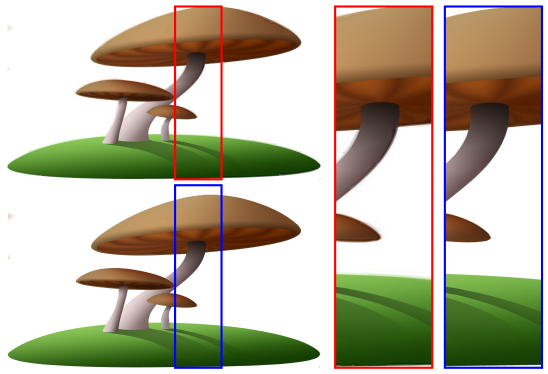

Figure 6: *Vectorised image without (top) and with (bottom) offset sampling around hard edges. Insets show the increased sharpness not only on silhouettes, but also hard edges coming from other sources like shading.*

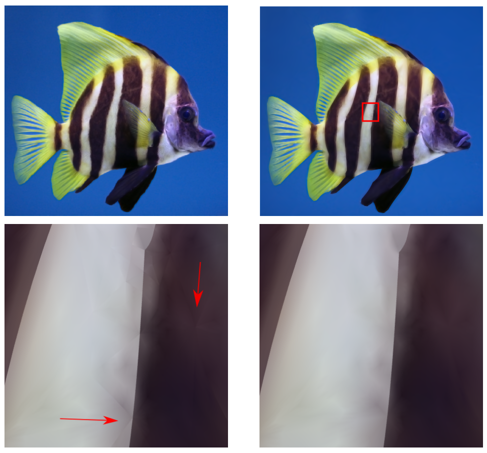

Figure 7: *Top: Input image (left) and our vectorised result (right). Bottom: Without smoothing (left) the seams of the mesh become apparent (some examples are highlighted by the red arrows), something especially undesirable at large magnification factors typical for vector graphics. The proposed smoothing of the mesh colour values of neighbouring mesh colour patches removes such seams for a more natural and higher quality result (right).*

the edges of the triangles. The smoothing procedure, where mesh colour values on edges are averaged with respect to their values on adjacent mesh colour patches, only needs to be done for edges that were previously deemed to be smooth, i.e., supporting (smooth) edges that were created in the triangulation step (Section 5), or the smooth edges from the feature extraction stage (Section 4). This step guarantees at least C^0 interpolation of colour in smooth regions. Figure 7 shows the difference between before and after colour smoothing. Before smoothing, the underlying triangulation is clearly visible in some regions. After smoothing, these artefacts vanish and the resulting vector image has C^0 colour interpolation everywhere except at hard features.

6.2. Rendering

The resulting vector graphics representation should lend itself to efficient rendering. To this end, we employ tessellation shaders in the modern graphics pipeline. All triangles are rendered as cubic Bézier triangles. We store the mesh colours in textures as proposed by [MSY19] and evaluate them using the process outlined in Section 6 in a fragment shader. This of course leads to duplication of mesh colour data for edges and vertex positions. This is actually necessary to model hard edges; this gives distinct colours on either side of the edge and no filtering should be applied there. For smooth edges, there does exist duplication.

We have not focused our efforts on improved filtering of textures, such as trilinear filtering and MIP-mapping. There have been recent improvements and variants on the mesh colour technique that extend them with increased filtering capabilities and hardware support [Yuk16; YLT19; MSY20].

Due to the unified processing of different resolution mesh colour patches, we are able to render each vector image using a single draw call. In addition to this, we employ adaptive tessellation to increase the performance even more. We use the projected edge length of the triangles to determine its tessellation factor. This guarantees that this optimisation does not create any gaps between adjacent (tessellated) triangles.

7. Results

Figure 1 shows an intricate input image with lots of details. Our extracted image features keep the sharpness hard features and capture relevant features like specular highlights and folds (Figure 1, f), while the mesh colour patches can capture fine texture as desired; see the accompanying video for a deeper look into this example. Figure 2 shows a simple logo that turns into a relatively simple vector image that could be edited further to remove the background, for example. Figure 3 shows an interesting combination of sharp and soft details, which our image features help capture accurately. Figure 6 shows another example of clean stylised graphics faithfully captured by our representation.

Figure 8 shows additional results showcasing a variety of image features, textures and details. Most of the small errors with respect to the original image accumulate along the edges. This can be attributed to the offset sampling we do along hard edges (Section 6) and the fact that the vectorised edges might not perfectly align with the actual edges in the image.

7.1. Performance

Table 1 shows the performance of our vectorisation method, for several of results featured in this paper. In addition to the total timings, we also show timings of several of the intermediate steps, and rendering times of the rasterisation of the obtained vector images. We ran the method on a low-end laptop, with an NVIDIA MX150 GPU, 8GB of RAM, and an Intel i5-8250 CPU; obtaining times that make our method practical.

The proposed method can vectorise any image with reasonable performance, taking just a few seconds for the whole process. The feature extraction step (Section 4) is dependent on the resolution of the input image, and the number of features in the image. The mesh generation step (Section 5) depends on the content of the image, and the number of extracted features and their orientations. For

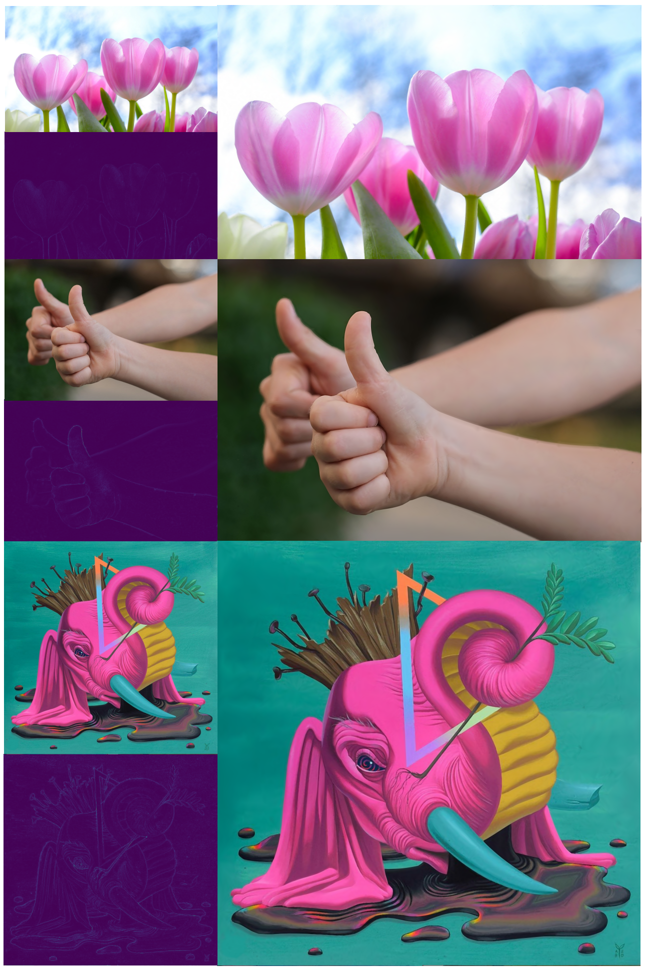

Figure 8: *Left: Input raster images and their corresponding error maps with respect to our rendered vector images (right). Overall, these examples were vectorised accurately all over the image with a mean squared error of* 2.73, 2.25, 4.69*, respectively, top to bottom. Bottom artwork by Allison Bamcat.*

instance, curved features that lie close to each other will generate more triangles than just a single curve. Then the performance of the colour fitting step (Section 6) depends on the number of generated triangles. However, due to our parallelisation and our adaptive patch resolution, it can be achieved quite efficiently. The resulting vector images can be rendered efficiently through our use of tessellation shaders. Even for images with a large number of triangles we achieve real-time performance rates (see accompanying video).

Compression. We also provide the compression ratios for the images featured in this paper in column CR in Table 1. We calculated the compression ratio with respect to the raw image data as is common in other vectorisation papers [CLL*20]. We base the size of our representation on the following. First of all we consider the difference between linear triangles with $3 \cdot 2 \cdot 4 = 24$ bytes and curved triangles, triangles with at least one curved edge, with $9 \cdot 2 \cdot 4 = 64$ bytes each. Furthermore, we consider the three different resolutions used with $3 \cdot 3 = 9$, $6 \cdot 3 = 18$, and $15 \cdot 3 = 45$ bytes each, respectively. Then the raw size of our representation is $\#LinearTriangles \cdot 24 + \#CurvedTriangles \cdot 64 + \#SmallPatches \cdot 9 + \#RegularPatches \cdot 18 + \#LargePatches \cdot 45$ bytes. Our approach is able to achieve comparable compression rates to other techniques and the raw data could be compressed even further using standard compression techniques such as zip. In addition, the size of the representation could be decreased even further, by not optimising for triangle size, but rather for the content that a triangle depicts, having lower mesh colour resolution for smoother or constant colour image regions.

Feature Parameters. To investigate the influence of the feature extraction step on the resulting vectorisation we have vectorised an image with varying levels of edge detection. In Figure 9, we show two different vectorisations of the same input image using different parameter settings for edge detection. The left image has an abundance of hard features detected in the furry areas of the image. By increasing the threshold values we are able to extract a sparser set of features. We can see that in the areas that had edges before, the banding features take over now. This, however, blurs the features as they are not deemed to be hard anymore and some detail is lost as evidenced by the insets. Nevertheless, the right image is still a good approximation of the original image as the most salient features are still preserved. The large number of features also leads to a large number of generated triangles, and although we are still able to render it efficiently, it negatively effects the compression ratio as seen in Table 1, rows 9 and 10.

7.2. Editing

We support low-level deformations, i.e., dragging vertex positions and tangent handles of the cubic edges of the Bézier triangles. However, in some areas a relatively large number of elements can be generated and editing can become tedious. Instead, the mesh can be effectively manipulated using proportional editing tools [LHFY12] paired with handling of curves in the spirit of [LJG14].

Table 1: *The performance of our vectorisation pipeline on several of the results featured in this paper. The time measurements are shown in seconds except for the rendering time, which is shown in milliseconds, and are split over the elements of our vectorisation pipeline: FE = Feature Extraction, MG = Mesh Generation, CF = Colour Fitting, RT = Rendering Time, CR = Compression Ratio.* $\triangle$ *indicates the number of triangles.*

Image	Resolution	$\triangle \cdot 10^3$	FE	MG	CF	Total	RT	CR
Fig 1	854×1024	~ 38	0.5	0.8	3.0	4.3	~ 23	0.72
Fig 2	785×618	~ 2	0.24	0.1	.5	0.9	< 1	0.07
Fig 3 bottom left	848×1280	~ 50	0.5	0.7	4	5.3	~ 25	0.78
Fig 6	1280×1181	~ 4	1.7	0.05	1.1	2.9	~ 2	0.06
Fig 7	441×441	~ 10	0.03	0.1	1.1	1.3	~ 3	0.89
Fig 8 top	1920×1284	~ 22	4.7	0.4	2.7	7.8	~ 25	0.18
Fig 8 middle	1280×853	~ 8	0.9	0.7	2.8	4.5	~ 10	0.16
Fig 8 bottom	1198×1198	~ 60	0.8	0.9	5.9	7.6	~ 17	0.77
Fig 9 left	864×864	~ 147	0.8	18.6	16	35.6	~ 33	2.68
Fig 9 right	864×864	~ 20	0.5	1.9	3.5	6	~ 16	0.52
Fig 13 left	639×479	~ 22	0.03	0.3	1.8	2.1	~ 7	1.24
Fig 14	1113×1291	~ 18	1.5	0.3	2.3	4.1	~ 24	0.26
Fig 15	441×631	~ 8	0.1	0.1	1.1	1.3	~ 4	0.56

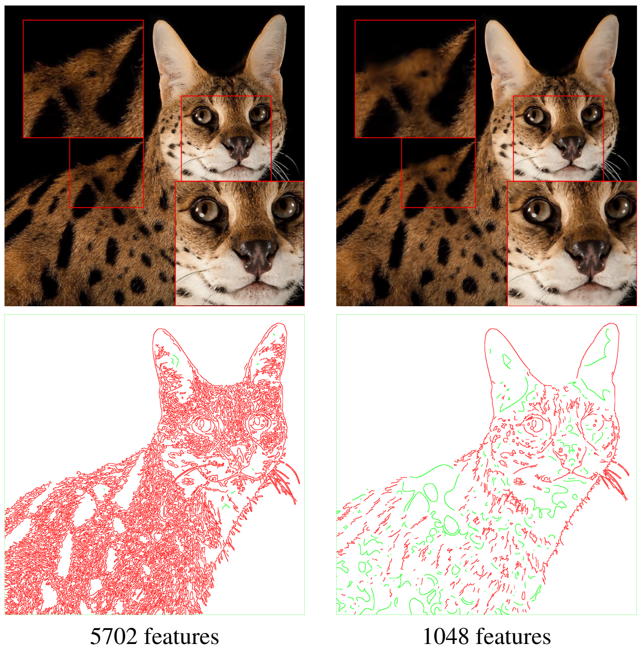

Figure 9: *Two vectorisations of the same image using different edge detection parameters. Left: Canny edge detection with low set to 15 and high to 100. Right: low set to 100 and high to 200.*

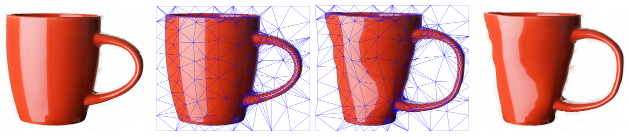

Figure 10: *Our vector images are easily editable by manipulating the curved mesh. From left to right: One of our vectorised results, the mesh of the vector, the edited mesh, and the rendered edited image. See the accompanying video for the editing session.*

We show an example of this in Figure 10 and the supplementary video. Because of our real-time rendering performance, users can efficiently zoom in and out for precise control. More elaborate manipulations such as grouping of features and others as proposed in [LHFY12] are also possible, but we did not implement them. Because of our overall performance, some hybrid raster-vector editing workflows, where local edits to a rasterised vector image could be quickly re-vectorised, could be explored as future work.

We have also created an application that allows users to create spline curves on top of a raster image. Similarly to our feature extraction stage of the pipeline, the user can mark them as hard or soft features. The created curved features are used as inputs to the rest of the pipeline. This user-guided process of vectorisation can be done fully interactively thanks to the efficiency of our pipeline. In addition, the user adds curves locally, which requires only local updates to the triangulation and mesh colours. Figure 11 shows an interactively generated vectorisation of a raster image. In addition, the same workflow could be used to clean up automatically vectorised images by adding or fixing features that were not captured correctly in the edge detection phase.

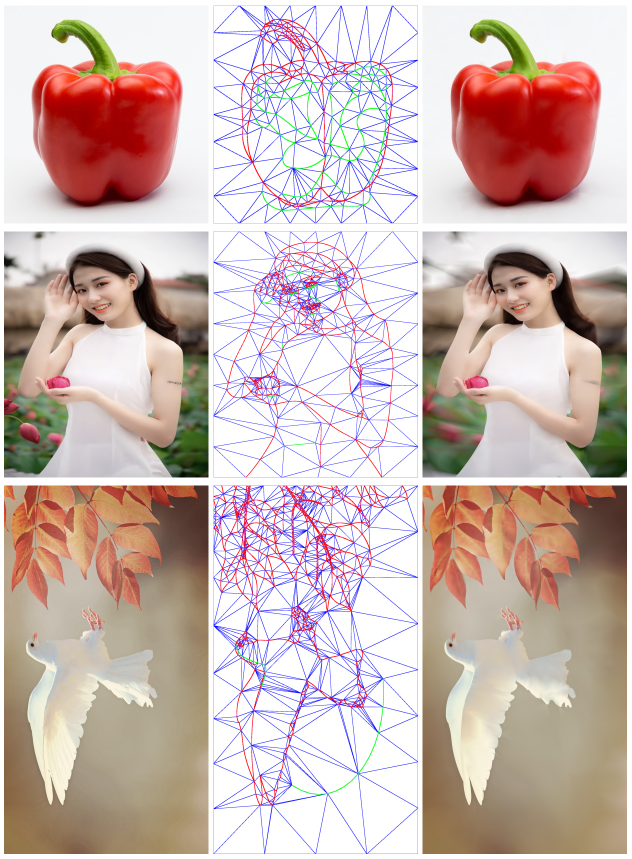

Figure 11: *User-guided vectorisations of several images. Left to right: Original image, user-placed curves in red and green, and generated mesh in blue, and resulting vectorisation. Top: 47 curves generated 460 triangles. Middle: 155 curves generated 993 triangles. Bottom: 186 curves generated 1404 triangles. Please, see the accompanying video for the interactive vectorisation session.*

7.3. Comparisons with Previous Work

We compare against recent methods and relevant primitives for image vectorisation: thin-plate splines (TPS), subdivision meshes, and diffusion curves. For all comparisons we must state that we can only approximately compare as many algorithm have only partly vectorised the image by first manually segmenting the image or for others we were not able to find the original input images and we were left with a lower quality input image. Still, we have to the best of our abilities tried to compare to the existing methods.

Figure 12 shows a comparison with a partially automatic TPS-based method [CLL*20]. As can be seen, their method is very good at capturing fine texture detail, but at the same time it scales similarly to a raster image, thus loosing some sharpness around hard edges. In addition, their vector patches often show seams under magnification. In contrast, our method keeps sharpness around hard edges and does not show texture seams thanks to our colour smoothing around soft edges. Our texture detail is affected both by the feature extraction step and the patch resolution, obtaining less realistic abstracted looks when not sufficient.

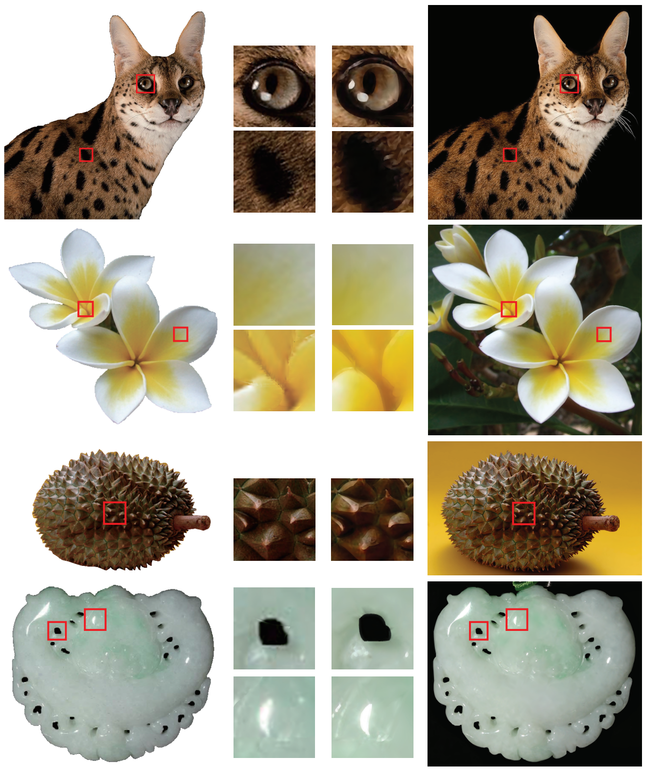

Figure 12: *Comparisons between [CLL*20] (left) and our proposed method (right). Our method seems better at preserving sharpness coming from geometric discontinuities (petals in the second row, holes in the last row), which are easily picked up by our extracted image features. However, ours is not that good at capturing extremely detailed textures like fur (first row). Insets also show the relevance of our colour smoothing across patches, absent in [CLL*20] (second and bottom rows). Vectorised backgrounds were not available from [CLL*20], but included for completeness.*

Because our image quality is often comparable to that of [CLL*20], we chose not to include gradient meshes [LHM09] in our comparisons, as their limitations when capturing highly detailed textures were already demonstrated by Chen et al. Figure 14 shows a comparison against another TPS-based method [XLY09], where our simpler and more performant pipeline achieves comparable results to their more intensive vectorisation method. Figure 15 shows a comparison with another mesh-based approach [LHFY12] that uses subdivision surfaces. Our decoupling into a spatial 2D mesh and 3D (RGB) mesh colours allows finer control over colour whilst not complicating the geometry, capturing higher level of detail while achieving comparable smoothness and mesh density.

Figure 13 shows comparisons with hierarchical diffusion curves [XSTN14]. We found that their image feature extraction translates into a global loss of clarity and detail for photos. For designed graphics, their method extracts cleaner features that produce quality closer to our method.

7.4. Discussion

Previous vector graphics primitives have focused mostly on high colour continuity surfaces. Our vector representation is 'only' C^0,

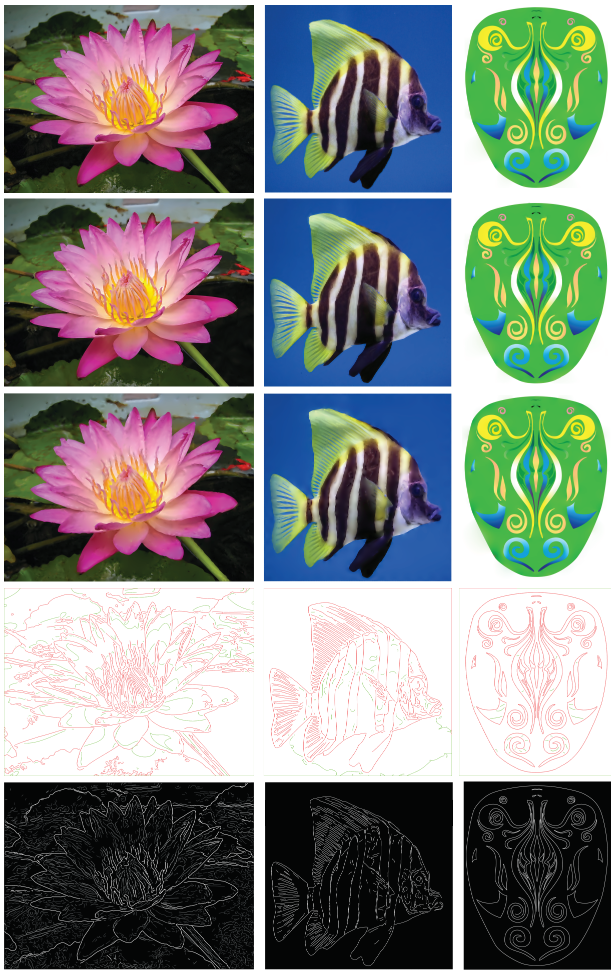

Figure 13: *From top to bottom: Input images, our results, results from [XSTN14], our extracted image features, and the ones from [XSTN14]. When applied to photos, our method produces sharper and cleaner results across the whole image (please, zoom in for details). For simpler inputs with clearer discontinuities, both methods perform similarly.*

but given our proposed feature extraction step, the mesh is generated in a way that all triangles lie on areas with little variation in colour. These areas can then easily be handled by the mesh colour patches, removing the need for smoother interpolation strategies like subdivision surfaces or thin-plate splines, which are less efficient to evaluate than our representation.

Although our pipeline uses standard and also state-of-the-art techniques for extracting features and constructing the geometry, our proposed steps are not dependent on each other. This means that the proposed pipeline can be potentially updated/upgraded in the future; specifically the curved triangulation might benefit the efficiency and quality of the pipeline. Although our feature extraction step manages to capture most of the salient features of the image, the resulting vector representation can only be as good as the features that have been extracted. Our focus in this paper has not been perfecting the feature extractor, but rather show the power of our choice of primitive for vectorisation. Naturally, there is room left

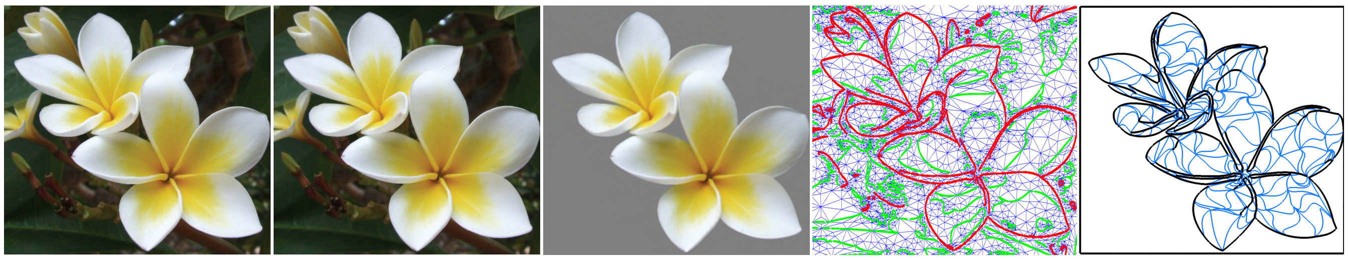

Figure 14: *From left to right: Input image, our result, result from [XLY09], our image features and triangle mesh, and the ones from from [XLY09]. Final quality is very similar between both methods, including sharpness around hard edges due to their feature alignment, and our offset sampling and mesh colours. The main differences lie in the 2D mesh, with theirs being sparser. However, the complexity of their mesh optimisation and the evaluation of their thin-plate splines is higher than our simpler but more efficient approach. Note that the vectorised background was not available from [XLY09], but we included it in our result for completeness.*

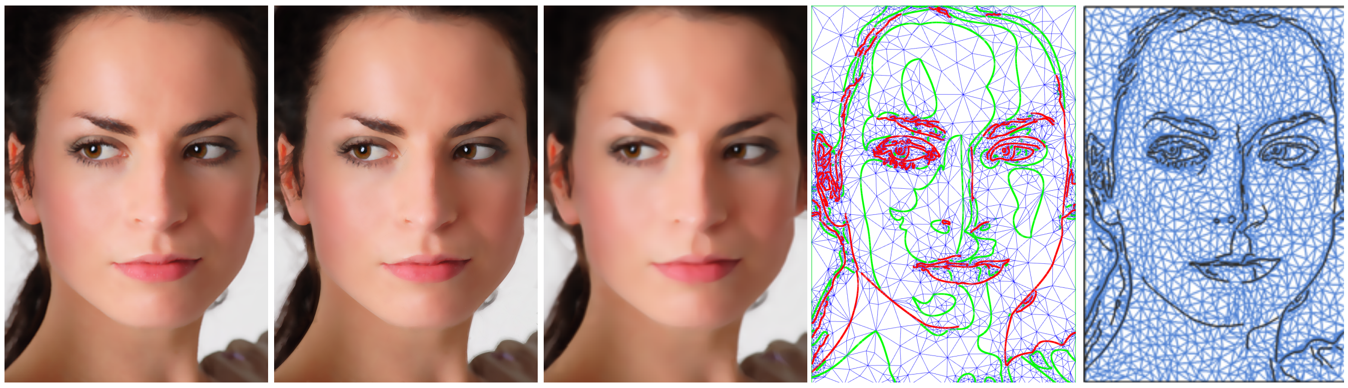

Figure 15: *From left to right: Input image, our result, result from [LHFY12], our image features and mesh, image features and mesh from [LHFY12]. While image features and triangle structure look similar, our mesh colour patches are able to capture more detail, even in smooth regions such as under the nose.*

for exploring combinations of traditional filtering and neural models. In any case, the rest of our pipeline can generally generate an adequate vectorisation given any set of curved input features.

Although tessellation shaders increase the rendering performance, it is still possible to render the images using only the CPU, or alternatively using compute shaders or texture based solutions for the Web. In addition, a downgraded approximation of the vector image can be created easily by extracting each mesh colour patch as a piece-wise linear triangulation, using the evaluated mesh colour positions as vertices. This may help the compatibility with current vector format standards such as PDF [Ado06] that do not directly support our proposed vector representation.

Theoretically, we could push our method to two extremes: on the one hand, the whole input image could be represented using only two triangles with a very high resolution of the mesh colours; on the other hand, each pixel of the input image could be captured by two triangles with a very low (in this case zero) texture resolution. Both are similar in spirit to [WZG*19], which is an efficient method, but it does not adapt its resolution to local regions with higher detail. In addition, not having explicit image features for geometric edits means the user would need to move each vertex manually, so having some sort of higher-level vectorised control features is more desirable. In summary, one needs to be find a good balance between the number (and geometry) of the mesh triangles and the mesh colour resolution. Our pipeline achieves this balance to a very good extent, but additional optimisations are still possible, such as more elaborate local adaptivity or iterative feedback between different steps of our pipeline.

8. Conclusion

We have presented a fully automatic image vectorisation method that can vectorise any image, ranging from natural images to logos and cartoon images, with good performance and reconstruction accuracy. Through our novel use of mesh colours, we are able to transfer detailed colour textures to a mesh of curved triangles generated from our feature extraction step.

Our efficient pipeline is capable of vectorising a wide range of image types in seconds on commodity hardware. To ensure that not only hard edges but also smooth image regions are captured correctly, we add soft edges obtained from colour banding to our image features. These features are vectorised, and then respected by the mesh of curved triangles equipped with mesh colours representing the vectorised image. This results in relatively sparse vector representations that are flexible and easy to edit, as demonstrated in the varied examples presented throughout the paper and the supplementary video.

References

[Ado06] ADOBE. *Adobe PDF*. https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdf_reference_archive/pdf_reference_1-7.pdf. 2006 2, 10.

[Ado19a] ADOBE. *Adobe Illustrator: Image Trace*. https://helpx.adobe.com/illustrator/using/image-trace.html. Online; accessed 13 December 2020. 2019 2.

[Ado19b] ADOBE. *Adobe Illustrator: Meshes*. https://helpx.adobe.com/illustrator/using/meshes.html. Online; accessed 13 December 2020. 2019 1, 2.

[BBG12] BOYÉ, SIMON, BARLA, PASCAL, and GUENNEBAUD, GAËL. "A Vectorial Solver for Free-Form Vector Gradients". *ACM Trans. Graph.* 31.6 (Nov. 2012). ISSN: 0730-0301 2.

[BDF14] BENJAMIN, MARK D., DIVERDI, STEPHEN, and FINKELSTEIN, ADAM. "Painting with Triangles". *NPAR 2014, Proceedings of the 12th International Symposium on Non-photorealistic Animation and Rendering*. Aug. 2014 4.

[BHEK21] BAKSTEEN, SARAH D., HETTINGA, GERBEN J., ECHEVARRIA, JOSE, and KOSINKA, JIŘÍ. "Mesh Colours for Gradient Meshes". *STAG: Smart Tools and Applications in Graphics*. Ed. by FROSINI, P., GIORGI, D., MELZI, S., and RODOLÀ, E. The Eurographics Association, 2021 2.

[BLHK18] BARENDRECHT, PIETER J, LUINSTRA, MARTIJN, HOGERVORST, JONATHAN, and KOSINKA, JIŘÍ. "Locally refinable gradient meshes supporting branching and sharp colour transitions". *The Visual Computer* 34.6-8 (2018), 949–960 2.

[BLW11] BOWERS, JOHN C, LEAHEY, JONATHAN, and WANG, RUI. "A ray tracing approach to diffusion curves". *Computer Graphics Forum*. Vol. 30. 4. Wiley Online Library. 2011, 1345–1352 2.

[Can86] CANNY, J. "A Computational Approach to Edge Detection". *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6 (1986), 679–698 3.

[CLL*20] CHEN, K., LUO, Y., LAI, Y., et al. "Image Vectorization With Real-Time Thin-Plate Spline". *IEEE Transactions on Multimedia* 22.1 (2020), 15–29 2, 3, 7–9.

[DDI06] DEMARET, LAURENT, DYN, NIRA, and ISKE, ARMIN. "Image compression by linear splines over adaptive triangulations". *Signal Processing* 86.7 (2006), 1604–1616 2.

[DLS13] DAI, WEN, LUO, TAO, and SHEN, JIANBING. "Automatic image vectorization using superpixels and random walkers". *Image and Signal Processing (CISP), 2013 6th International Congress on*. Vol. 2. IEEE. 2013, 922–926 2.

[DSG*20] DOMINICI, EDOARDO ALBERTO, SCHERTLER, NICO, GRIFFIN, JONATHAN, et al. "PolyFit: Perception-Aligned Vectorization of Raster Clip-Art via Intermediate Polygonal Fitting". *ACM Trans. Graph.* 39.4 (July 2020). ISSN: 0730-0301 3.

[EVA*20] EGIAZARIAN, VAGE, VOYNOV, OLEG, ARTEMOV, ALEXEY, et al. "Deep Vectorization of Technical Drawings". *Computer Vision – ECCV 2020*. Ed. by VEDALDI, ANDREA, BISCHOF, HORST, BROX, THOMAS, and FRAHM, JAN-MICHAEL. Cham: Springer International Publishing, 2020, 582–598. ISBN: 978-3-030-58601-0 3.

[EZ98] ELDER, J.H. and ZUCKER, S.W. "Local scale control for edge detection and blur estimation". *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20.7 (1998), 699–716 3.

[FLB16] FAVREAU, JEAN-DOMINIQUE, LAFARGE, FLORENT, and BOUSSEAU, ADRIEN. "Fidelity vs. Simplicity: A Global Approach to Line Drawing Vectorization". *ACM Trans. Graph.* 35.4 (July 2016). ISSN: 0730-0301 3.

[FLB17] FAVREAU, JEAN-DOMINIQUE, LAFARGE, FLORENT, and BOUSSEAU, ADRIEN. "Photo2clipart: Image Abstraction and Vectorization Using Layered Linear Gradients". *ACM Trans. Graph.* 36.6 (Nov. 2017). ISSN: 0730-0301 2.

[HDS*18] HOSHYARI, SHAYAN, DOMINICI, EDOARDO ALBERTO, SHEFFER, ALLA, et al. "Perception-Driven Semi-Structured Boundary Vectorization". *ACM Trans. Graph.* 37.4 (July 2018). ISSN: 0730-0301 3.

[HSG*19] HU, YIXIN, SCHNEIDER, TESEO, GAO, XIFENG, et al. "TriWild: Robust Triangulation with Curve Constraints". *ACM Trans. Graph.* 38.4 (July 2019), 52:1–52:15. ISSN: 0730-0301 4.

[HZY*20] HE, J., ZHANG, S., YANG, M., et al. "BDCN: Bi-Directional Cascade Network for Perceptual Edge Detection". *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2020), 1–14 3.

[JCW09] JESCHKE, STEFAN, CLINE, DAVID, and WONKA, PETER. "A GPU Laplacian Solver for Diffusion Curves and Poisson Image Editing". *Transaction on Graphics (Siggraph Asia 2009)* 28.5 (Dec. 2009), 1–8. ISSN: 0730-0301 2.

[JCW11] JESCHKE, STEFAN, CLINE, DAVID, and WONKA, PETER. "Estimating color and texture parameters for vector graphics". *Computer Graphics Forum*. Vol. 30. 2. Wiley Online Library. 2011, 523–532 2.

[KL11] KOPF, JOHANNES and LISCHINSKI, DANI. "Depixelizing Pixel Art". *ACM Trans. Graph.* 30.4 (July 2011). ISSN: 0730-0301 2.

[LCH*17] LIU, Y., CHENG, M., HU, X., et al. "Richer Convolutional Features for Edge Detection". *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, 5872–5881 3.

[LHFY12] LIAO, ZICHENG, HOPPE, HUGUES, FORSYTH, DAVID, and YU, YIZHOU. "A subdivision-based representation for vector image editing". *IEEE transactions on visualization and computer graphics* 18.11 (2012), 1858–1867 2, 5, 7–10.

[LHM09] LAI, YU-KUN, HU, SHI-MIN, and MARTIN, RALPH R. "Automatic and topology-preserving gradient mesh generation for image vectorization". *ACM Transactions on Graphics (TOG)*. Vol. 28. 3. ACM. 2009, 85 2, 9.

[Lin96] LINDEBERG, T. "Edge detection and ridge detection with automatic scale selection". *Proceedings CVPR IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 1996, 465–470 3.

[LJD*19] LU, SHUFANG, JIANG, WEI, DING, XUEFENG, et al. "Depth-aware image vectorization and editing". *The Visual Computer* (2019), 1–13 2.

[LJG14] LIU, SONGRUN, JACOBSON, ALEC, and GINGOLD, YOTAM. "Skinning cubic Bézier splines and Catmull-Clark subdivision surfaces". *ACM Transactions on Graphics (TOG)* 33.6 (2014), 1–9 7.

[LJH13] LI, XIAN-YING, JU, TAO, and HU, SHI-MIN. "Cubic mean value coordinates." *ACM Trans. Graph.* 32.4 (2013), 126–1:10 2.

[LKSD17] LIENG, HENRIK, KOSINKA, JIŘÍ, SHEN, JINGJING, and DODGSON, NEIL A. "A Colour Interpolation Scheme for Topologically Unrestricted Gradient Meshes". *Computer Graphics Forum*. Vol. 36. 6. Wiley Online Library. 2017, 112–121 2.

[LL06] LECOT, GREGORY and LEVY, BRUNO. "Ardeco: automatic region detection and conversion". *17th Eurographics Symposium on Rendering-EGSR'06*. 2006, 349–360 2.

[LLGR20] LI, TZU-MAO, LUKÁČ, MICHAL, GHARBI, MICHAËL, and RAGAN-KELLEY, JONATHAN. "Differentiable Vector Graphics Rasterization for Editing and Learning". *ACM Trans. Graph.* 39.6 (Nov. 2020). ISSN: 0730-0301 2.

[MA09] MAINI, RAMAN and AGGARWAL, HIMANSHU. "Study and comparison of various image edge detection techniques". *International journal of image processing (IJIP)* 3.1 (2009), 1–11 3.

[MC20] MANDAD, MANISH and CAMPEN, MARCEL. "Bézier Guarding: Precise Higher-Order Meshing of Curved 2D Domains". 39.4 (July 2020). ISSN: 0730-0301 4.

[MSY19] MALLETT, IAN, SEILER, LARRY, and YUKSEL, CEM. "Patch Textures: Hardware Implementation of Mesh Colors". *High-Performance Graphics (HPG 2019)*. Strasbourg, France: The Eurographics Association, 2019. ISBN: 978-3-03868-092-5 2, 4, 6.

[MSY20] MALLETT, IAN, SEILER, LARRY, and YUKSEL, CEM. "Patch Textures: Hardware Support for Mesh Colors". *IEEE Transactions on Visualization and Computer Graphics* (2020). ISSN: 1077-2626 6.

[NS19] NAJGEBAUER, PATRYK and SCHERER, RAFAŁ. "Inertia-based Fast Vectorization of Line Drawings". *Computer Graphics Forum* 38.7 (2019), 203–213 3.

[OBW*08] ORZAN, ALEXANDRINA, BOUSSEAU, ADRIEN, WINNEMÖLLER, HOLGER, et al. "Diffusion Curves: A Vector Representation for Smooth-Shaded Images". *ACM Transactions on Graphics* 27.3 (2008), 92–1 2, 3, 5.

[PB06] PRICE, BRIAN and BARRETT, WILLIAM. "Object-based vectorization for interactive image editing". *The Visual Computer* 22.9 (2006), 661–670 2.

[PJS15] PRÉVOST, ROMAIN, JAROSZ, WOJCIECH, and SORKINE-HORNUNG, O. "A Vectorial Framework for Ray Traced Diffusion Curves". *Computer Graphics Forum* 34 (2015) 2.

[RLB*14] RICHARDT, C., LOPEZ-MORENO, J., BOUSSEAU, A., et al. "Vectorising Bitmaps into Semi-Transparent Gradient Layers". *Computer Graphics Forum* 33.4 (2014), 11–19 2.

[SBBB20] STANKO, TIBOR, BESSMELTSEV, MIKHAIL, BOMMES, DAVID, and BOUSSEAU, ADRIEN. "Integer-Grid Sketch Simplification and Vectorization". *Computer Graphics Forum* 39.5 (2020), 149–161 3.

[Sch90] SCHNEIDER, PHILIP J. "An Algorithm for Automatically Fitting Digitized Curves". *Graphics Gems*. USA: Academic Press Professional, Inc., 1990, 612–626. ISBN: 0122861695 4.

[SKFS20] SHUGRINA, MARIA, KAR, AMLAN, FIDLER, SANJA, and SINGH, KARAN. "Nonlinear Color Triads for Approximation, Learning and Direct Manipulation of Color Distributions". *ACM Trans. Graph.* 39.4 (July 2020). ISSN: 0730-0301 2.

[SLWS07] SUN, JIAN, LIANG, LIN, WEN, FANG, and SHUM, HEUNG-YEUNG. "Image vectorization using optimized gradient meshes". *ACM Transactions on Graphics (TOG)*. Vol. 26. 3. ACM. 2007, 11 2.

[SMC*13] SILVA, M. A. G., MONTENEGRO, A., CLUA, E., et al. "Real Time Pixel Art Remasterization on GPUs". *2013 XXVI Conference on Graphics, Patterns and Images*. 2013, 274–281 2.

[VK18] VERSTRAATEN, TEUN W. and KOSINKA, JIŘÍ. "Local and Hierarchical Refinement for Subdivision Gradient Meshes". *Computer Graphics Forum* 37.7 (2018), 373–383 2.

[WZG*19] WEI, GUANGSHUN, ZHOU, YUANFENG, GAO, XIFENG, et al. "Field-aligned Quadrangulation for Image Vectorization". *Computer Graphics Forum*. Vol. 38. 7. Wiley Online Library. 2019, 171–180 2, 10.

[XLY09] XIA, TIAN, LIAO, BINBIN, and YU, YIZHOU. "Patch-based image vectorization with automatic curvilinear feature alignment". *ACM Transactions on Graphics (TOG)*. Vol. 28. 5. ACM. 2009, 115 2–4, 9, 10.

[XSTN14] XIE, GUOFU, SUN, XIN, TONG, XIN, and NOWROUZEZAHRAI, DEREK. "Hierarchical diffusion curves for accurate automatic image vectorization". *ACM Transactions on Graphics (TOG)* 33.6 (2014), 230 2, 9.

[XT15] XIE, SAINING and TU, ZHUOWEN. "Holistically-Nested Edge Detection". *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. Dec. 2015 3.

[XWLS17] XIE, JUN, WINNEMÖLLER, HOLGER, LI, WILMOT, and SCHILLER, STEPHEN. "Interactive Vectorization". *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI '17. Denver, Colorado, USA: Association for Computing Machinery, 2017, 6695–6705. ISBN: 9781450346559 2.

[YCZ*16] YANG, M., CHAO, H., ZHANG, C., et al. "Effective Clipart Image Vectorization through Direct Optimization of Bezigons". *IEEE Transactions on Visualization & Computer Graphics* 22.02 (Feb. 2016), 1063–1075. ISSN: 1941-0506 1.

[YKH10] YUKSEL, CEM, KEYSER, JOHN, and HOUSE, DONALD H. "Mesh Colors". *ACM Trans. Graph.* 29.2 (Apr. 2010). ISSN: 0730-0301 2, 4.

[YLT19] YUKSEL, CEM, LEFEBVRE, SYLVAIN, and TARINI, MARCO. "Rethinking Texture Mapping". *Computer Graphics Forum (Proceedings of Eurographics 2019)* 38.2 (2019), 535–551 4, 6.

[Yuk16] YUKSEL, CEM. "Mesh Colors with Hardware Texture Filtering". *ACM SIGGRAPH 2016 Talks*. SIGGRAPH '16. Anaheim, California: Association for Computing Machinery, 2016. ISBN: 9781450342827 5, 6.

[ZDZ17] ZHAO, SHUANG, DURAND, FREDO, and ZHENG, CHANGXI. "Inverse Diffusion Curves using Shape Optimization". *IEEE transactions on visualization and computer graphics* (2017) 2.

[ZZW14] ZHOU, HAILING, ZHENG, JIANMIN, and WEI, LEI. "Representing images using curvilinear feature driven subdivision surfaces". *IEEE Transactions on Image Processing* 23.8 (2014), 3268–3280 2.