Pacific Graphics 2020
E. Eisemann, A. Jacobson, and F.-L Zhang
(Guest Editors)

Volume 39 (2020), Number 7

Deep Diffusion Curves: High-Performance Attributes-Controllable Vector Graphics

paper1052

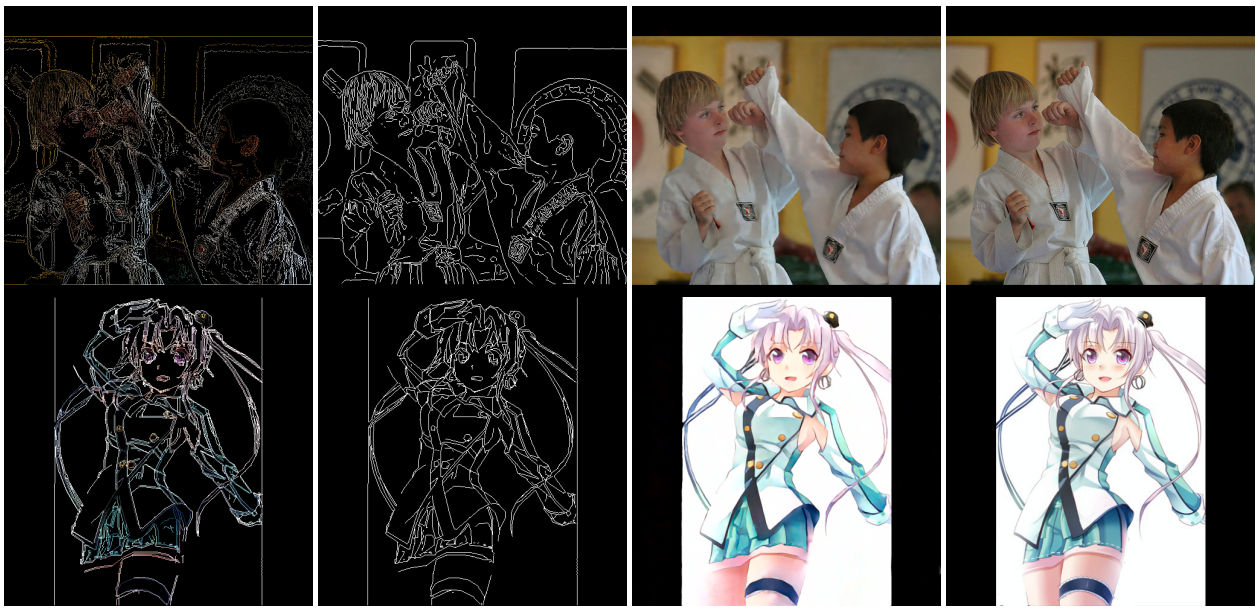


Figure 1: ***Examples of reconstruction of real-world photo and anime-style painting.*** *Columns from left to right correspond to color source map, binary edge map, reconstructed image, and the input image, respectively. The DDC framework can learn to reconstruct different styles of images from sparse color sources and edge locations and runs order-of-magnitude faster than Diffusion Curves.*

Abstract

Vector graphics represent visual objects using geometric primitives and offer several benefits over raster images such as ease of post-editing and compact representation. Diffusion Curves (DCs) and its variants (e.g. Biharmonic Diffusion Curves, Poisson Vector Graphics) gain popularity ever since its inception in vector graphics, thanks to their power in color gradient representation. However, their representation power is confined by harmonic or bi-harmonic color gradients. Moreover, the vectorization methods for DCs require certain complicated numerical optimization algorithms, resulting in the sluggish batch-processing of large image databases. To tackle the challenges, we advocate a data-driven vector graphics framework, called Deep Diffusion Curves (DDCs). In essence, DDCs comprise a vectorizer for image vectorization and a reconstructor for rendering. DDCs have three salient advantages over conventional methods. First, DDC combines the simplicity of the DC format with learned color shading styles. We used the deep neural networks as an approximate numerical regressor that learns the mapping between sparse color sources and color shadings, thus achieving much better reconstruction results with no need for any types of PDEs. Second, DDC is highly efficient once trained, whose reconstructor learns to render the raster image directly via neural networks. Third, the learned neural networks are attribute-controllable. By adjusting some key attributes, DDCs could serve as a powerful authoring tool to create new image samples different from training data. We demonstrate the power of DDCs by applying the framework to natural images and anime-style paintings. Extensive qualitative and quantitative evaluations have confirmed that DDCs could reconstruct (low-frequency) shadings of both natural images and stylized paintings and the DDC vectorizer, reconstructor run 3, 1 order-of-magnitude faster, respectively while retaining lower regression error than DCs.

CCS Concepts

• ***Computing methodologies*** → ***Non-photorealistic rendering;*** *Graphics systems and interfaces;*

1. Introduction

Two-dimensional vector graphics are essential components widely used in various modern-day applications such as clip-arts, fonts rendering, and user interface components. One common challenge for 2D vector graphics is to create or represent color gradients; current industrial standard (i.e. SVG) only supports linear or radial gradients. Plural methods were proposed to give vector graphics the power to represent non-linear color gradients including mesh-based methods such as Gradient Meshes [SLWS07] and curve-based methods such as Diffusion Curves (DCs) [OBW*08]. Due to ease of authoring, DCs has gained popularity during the last decades and inspired a series of improvements [XSTN14] [ZDZ17] [HSF*20]. DCs and its variants assume that color sources were defined on each edge, and diffuse into neighboring regions follow certain partial differential equations (PDEs) such as the Poisson's equation, and the representation power is determined by these PDEs. However, the process of formulating PDEs for a certain type of color gradients, and designing solvers for these non-trivial PDEs is often difficult. For example, hand-painted artworks such as anime-style paintings shown in row 2 of Figure 1, is challenging for DCs which were developed for nature images. Another issue with DCs is the performance of the vectorization methods. Currently-available vectorization methods often rely on certain optimization algorithms that require tens of seconds to finish, making vectorization of large image database impractical.

With the rapid growth of the availability of large image data-sets and Graphics Processing Units (GPUs), it'll be beneficial if we can design high-performance data-oriented methods that can learn to represent different kinds of color gradients directly from data, bypassing the need to develop PDEs and solvers.

In this paper, we develop a vector graphics framework based on DCs, called Deep Diffusion Curves (DDCs), to learn color shadings directly from data-sets, besides, we accelerate the vectorization process of DCs using a deep neural net. Since the core components of DDC were mostly composed of deep neural nets (DNNs), with its highly parallel nature, DDCs benefit from modern GPU hardware which has shown superb inference performance and was already shown their applicability in many fields. Instead of relying on partial differential equations (PDEs) to model color gradients, we trained a deep neural regressor that minimizes the reconstruction error given color sources. Also, the regressor is attribute-controllable; by adjusting input parameters, the regressor can be used to author new images that have never seen in the training data-set. Additionally, we overcome the performance issue of the Gaussian Scale Space analysis [Lin94] which was used in the vectorization process of DCs by training a DNN to approximate the function that maps given input image pixels to edge parameters including edge location, edge length, and edge blurriness.

The DDC framework includes two components: the vectorizer and the reconstructor. The vectorizer takes a bitmap image as inputs and converts the input image to feature maps. The feature maps then go through a post-processing procedure that extracts edges from the feature maps and convert the edges to diffusion curves. The reconstructor takes the feature map as well as the edge map as inputs and reconstructs the original input to the vectorizer as close as possible. Note that the vectorizer and the reconstructor can operate independently; thus the vectorizer can be used to batch vectorizing images while the reconstructor being used in a post-editing authoring tool.

To demonstrate the practicability of our framework, we developed a web-based authoring tool that supports post-editing of our vectorized format as well as vector graphics creation from scratch. We show the applicability of our framework by using our prototypical authoring tool to do color-editing, geometry editing, and object creation from scratch.

We evaluate the applicability of our framework by comparing the reconstruction results with DCs on both nature images and stylized paintings. For stylized paintings, we did a case study on anime-style paintings which is a symbolic style of modern Asian sub-culture.

Through qualitative and quantitive evaluations, we show that DDCs performed vectorization and reconstruction at least an order-of-magnitude faster than conventional vectorization methods while achieving lower reconstruction error for both anime-style paintings and nature images.

In short, our contributions include:

- A highly efficient neural network regressor that learns the mapping between given edge attributes and color shading from a given data-set, giving rise to much better vectorized images than conventional PDE based methods.
- Accelerated the Diffusion Curve's vectorization process at least 3 order-of-magnitudes and could be used for vectorization of large image databases.
- A web-based authoring tool for editing edge attributes (color, location, blurriness, and lengths) of DDCs.

2. Related Work

Vectorized representations, vectorization methods, and applications of computer graphics techniques on anime-style paintings are the most related works of our method. In this section, we briefly review the existing works of these categories.

Vectorized representations. Vector graphics have long been considered having limited representing power. Formats like SVG [Wor18] using geometric primitives to describe shapes and only support simple color gradients such as radial and linear gradients. Around a decade ago, Gradient Meshes [SLWS07] and Diffusion Curves [OBW*08], which both have much stronger shape and shading representation power, were proposed. Gradient Meshes are 2D Fegurson patches [Fer64] [Far93] with color parameters associated with each control point; color gradients are represented by blending colors on control points. Although Gradient Meshes could represent almost any kind of color gradients, to represent complex shapes, numerous control points have to be added and edited; to create or edit complex shapes easily become tedious, if not infeasible. On the other hand, Diffusion Curves model the shading of the image by partial differential equations (PDEs); colors are only defined on a set of sparse curves and are used as boundary conditions for the underlying PDEs. This made DCs well-suited for free-from cliparts authoring and editing because of sparser geometric primitives; besides, this "define boundary first, and then fill in colors" work-flow are well-suited to artists' habits. In the process

of creating anime-style paintings, artists often sketch the contours of an object first and then fill in the colors. To better accommodate the creation process of artworks, DDC used a similar curve-based representation as DCs with underlying shading model tailored for anime-style paintings.

Variations of DCs were proposed to further improve its representation power and performance. Bezerra et al. [BEDT10] proposed diffusion constraints to create various kinds of artistic effects. Thin plate splines [FSH11] were used to extend the color gradient representation power of DCs from first order to second order. Illbery et al. [IKCM13] proposed biharmonic Diffusion Curves and proposed a solver using boundary element methods. Generalized Diffusion Curves [Jes16] used two first-order Diffusion Curves to model higher-order diffusion curves. Shading Curves [LTKD15] utilize subdivision meshes and Phong shading to represent color transitions. Poisson Vector Graphics (PVG) [HSF*20] extends the Laplace equation used in DCs to Poisson's equations and provides Poisson Regions and Poisson Curves which directly constraint laplacian values and can be used to create lighting effects. Although all of the previously introduced methods provide higher representation power for color gradients, all of these methods model color gradients through explicitly define the governing PDEs, in contrast, our method learns the shading style from data.

It needs to solve Poisson's equations to render Diffusion Curve images. Fast DC solvers based on GPUs [War10] [JCW09] were proposed that made DC rendering real-time, thus animations based on DCs were possible. Bowers et al. [BLW11] formulate DC solving as a raytracing problem and provides texture mapping capability for DCs. Finite element methods [BBG12] and boundary element methods [IKCM13] were also been used in solving Diffusion Curves images. Compared with these solvers, our reconstruction model learned the mapping from given color sources to the original images and had 10 to 20 milliseconds inference time for a typical image of 512 by 512 pixels.

Vectorization. Image vectorization is a topic of particular importance for vector graphics. Conventional vectorization methods generally follow three steps: extracting salient edge features, convert these features to vector primitives, and optimizing parameters of these primitives to reduce reconstruction error. For example, ArDeco [LL06] first extracts relevant edges from images, and then discretize the image domain to triangular mesh. The method goes on to group neighboring triangles with similar colors into regions and fitting linear gradients for the grouped regions. The output is a set of polygonal regions with their color gradient parameters.

Several vectorization methods [SLWS07] [XLY09] [LHFY12] for Gradient Meshes have been proposed. These methods often use subdivision-meshes that can iteratively subdivide part of the mesh to a smoother geometry that can better fit underlying image features. However, to reconstruct complex color shadings, a dense mesh that is difficult to edit is required.

With the growing popularity of Diffusion Curves (DCs), various vectorization methods were proposed to vectorize raster images into DCs. Orzan et al. [OBW*08] extract relevant edge features in a Gaussian scale-space, the extracted edges are fitted and converted to Bézier curves or polylines. Color parameters are assigned by sampling color from a small distance along edge normals. Jeschke et al. [JCW11] proposed a method that can better estimate color parameters for DCs. Their method first extracts relevant edges, transforms them into Bézier-curves, and used an optimization method to assign colors to these curves; also, Gabor noise was used to represent certain types of high-frequency patterns. Xie et al. [XSTN14] proposed hierarchical diffusion curves that extract curves in the laplacian domain. They also showed that extracting curves in the laplacian domain resulted in fewer primitives. Their method also supports level-of-detail reconstruction by only retaining curves extracted from a certain level of the laplacian scale space. Certain optimizations based on empirical properties of extracted curves were used to reduce the size of the linear system required to be solved. However, finding the inverse matrix of the reduced-size matrix is still taxing for modern-day CPU and GPUs. Unlike previous methods that extract curves and fixed their geometry, Zhao et al. [ZDZ17] adapted the geometry of extracted curves using a geometric gradient descent method that iteratively adapts curve geometry to better match image features and achieved lower reconstruction error compared with previous works. However, on average it still takes more than 10 seconds to vectorize a typical nature image. Song et al. [SWWW15] presented a Multi-Layered-Perceptron (MLP)-based texture compression algorithm for Diffusion Curve textures, and it's efficient to render on GPU. As far as our knowledge, their method is the only method that utilizes deep learning techniques for Diffusion Curves. Their neural network accepts a 2D pixel position as input and returns rendered color for that pixel position. In contrast, our reconstructor network accepts a whole image that encodes color source constraints as input and generates a whole image as output. In brief, conventional vectorization methods mostly rely on optimization and parameter fitting techniques which are less efficient. Our framework utilizes deep neural networks that once trained, having high inference performances.

Applications for anime-style paintings. Anime-style is often used to referring to artworks produced for Japanese animations, mangas (comics), and video games. This style is especially popular among young adults and has strong cultural influences world-wide. Nowadays, two lines of research form the majority of anime-style related researches. One popular trend is generating anime-style paintings. With the rise of Generative Adversarial Networks (GANs) and high quality public accessible anime-style image databases [AcB20], generating anime-style faces [KKKL19] [JZL*17] [Lei17] [tdr16] [Pre17a] or full-body figures [HTL*18] is made possible. Another popular topic is coloring manga, or comics, which were often drawn in monotone via line sketches. Unlike nature images, much less shading information exhibit in mangas, which often relies on artistic techniques such as hatching and screen tones to express shading. To mitigate this, Qu et al. [QWH06] proposed a semi-automatic manga colorization method that propagates colors of user strokes to near-by image areas based on regional pixel pattern and intensity continuity. Sato et al. [SMYA14] developed an example-based method for manga colorization. They used graph matching to match segments of the reference image to the target image. However, their method only works on simple images. Hensmen and Aizawa [HA17] designed a manga colorization framework that was being able to colorize manga using a single reference image. Their method uses cGANs [MO14] along with classic image post-processing tech-

niques, e.g. segmentation, color quantization, to achieve visual-pleasing results. Zhang et al. [ZLW*18] proposed to use two-stage cGANs, one for generating color proposals, another for color post-processing, to achieve semi-automatic manga colorization.

Slightly different from mangas, which use lots of screen tones to express surface textures, line-arts is an art form that consists of clean line strokes, and are often lack of surface textures and shading. PaintChainer [Pre17b] is an online line-arts colorization tool that provides fully automatic colorization and semi-automatic colorization hinted by sparse user strokes. Ci et al. [CMW*18] applied cGAN with local feature networks for line-arts colorization and achieved state-of-the-art results. Compared with these approaches, our method is based on vector graphics instead of raster images, thus, we can provide benefits that raster images can not offer, such as better editability.

Although previous approaches that utilize DNNs have shown great results on manga and line-arts colorization, there have been very few works for vectorizing manga, line-arts, and cartoons. Saito et al. [SMYA14] embedded anime-style paintings into semantic vector spaces that facilitate semantic painting retrieval. Zhang et al. [ZCZ*09] vectorized cartoon animations based on trap-ball segmentation and temporal consistency. Yao et al. [YHL*17] proposed a manga vectorization method that used regional screen tone properties to do segmentation, segment refinement, and regional lighting estimation; these regions are later fitted to Bézier patches which can be manipulated and rendered in real-time. Compared with their method, our method is designed for colored anime-style paintings and we don't rely on screen-tone features.

3. Deep Diffusion Curves

Deep Diffusion Curves (DDCs) follow the same spirits of Diffusion Curves (DCs), which assume an image consists of a set of curves, and color sources locate on the curves and diffuse to nearby areas. In this section, we first briefly give an overview of the related background, followed by detailed discussions of DDCs.

3.1. Overview

Overview of the DDC framework is shown in Figure 2 and Figure 3. The DDC framework consists of two components, the vectorizer, and the reconstructor. The vectorizer takes an input image, extract relevant edge parameters, and convert them to polylines or Bézier curves. The reconstructor rasterize given polylines to obtain a color source map and an edge map. The color source map and edge map are fed to the reconstruction neural network to obtain a rendered image. We describe each part in detail in the following section.

Figure 3 shows the reconstructor of DDC. The core of the DDC reconstructor is a regressor that was trained by (color source map, edge map, original image) tuples. The color source maps and edge maps were generated from the vectorized format that was vectorized using the vectorization method described in Diffusion Curves [OBW*08]. We'll discuss in detail the color source map and edge map generating procedure in Section 3.3.

Figure 2 shows how we accelerate the vectorization method described in [OBW*08]. Since the performance bottleneck lies in the process of the Gaussian Scale Space analysis, we use a deep neural net to extract the edge map by supervising the neural nets with the (original image, edge map) pairs where the edge map was generated by the Gaussian Scale Space analysis process. The edge map will then go through the pixel-chain tracing, color sampling, and polyline fitting procedure described in detail in Section 3.2.

3.2. Vectorizer

Given an image, the vectorizer processing the image in three steps: (1) salient edge detection; (2) pixel chain tracing; (3) polyline fitting. We describe these steps in the following subsections.

Salient edge detection. Given an image, the first step of our vectorizer is detecting salient edges from given images. Our vectorizer closely follows the edge detection procedure used in DCs [OBW*08]. In their approach, a vision-based algorithm was used to construct a Gaussian-scale-space. A Gaussian-scale-space is a stack of images constructed by repeatedly applying the same Gaussian blur kernel. So, in the Gaussian-scale-space higher level corresponds to blurrier images. The Gaussian-scale-space is utilized for estimating how blurry an edge is, i.e., to estimate the blur factors for each edge pixel. Canny edge detector was applied to every level of the Gaussian-scale-space, so it extracts a set of edges from every level. From this stack of extracted edge, the algorithm computes the best level that can best represent the edge, and the best level was used as the blur factor for this edge. The final output is an image containing blur factors for edge pixels. We call the output image "edge map". Note that in the edge map, blur factors are used to identify the degree of blur of an edge, and are also used to localize edges.

Acceleration of edge map extraction. The edge extracting method described above is time-consuming. A typical extraction time for a 512 by 512 pixels images consumes about $5 \sim 10$ seconds. We accelerate this process by training a neural network to learn the mapping between the input image and the output blur factors. This mapping is pixel-to-pixel mapping similar to image segmentation. In image segmentation, every input pixel was mapped to a value representing the class label. In our network, every pixel is mapped either to zero (not an edge pixel) or a blur factor (an edge pixel with how blurry the edge is). We used a UNET [RFB15] to learn this mapping since UNET is well-known to perform well on image segmentation. The input to our UNET is the original image, the output is the edge map generated by UNET. The UNET is trained by the image pairs of the input image and its corresponding edge map generated by the Gaussian Scale Space analysis process described above. Once trained, the edge map generating process is highly efficient.

Pixel chain tracing. After obtaining the edge map, we track the pixel chains from this edge map that represent object contours. The pixel chains will then be converted to polylines or Bézier curves. Algorithm 1 shows the pixel-tracing procedure. The algorithm scans each pixel in the input image. If it found a pixel with non-zero values (potential edge pixel), it searches the *2-ring* neighbor of the founded pixel to see if there are any non-zero valued pixels in this region. If no such pixels are found, then the algorithm continues to the next iteration. Otherwise, it calls the *FollowChain* procedure which iteratively finds non-zero value pixels in

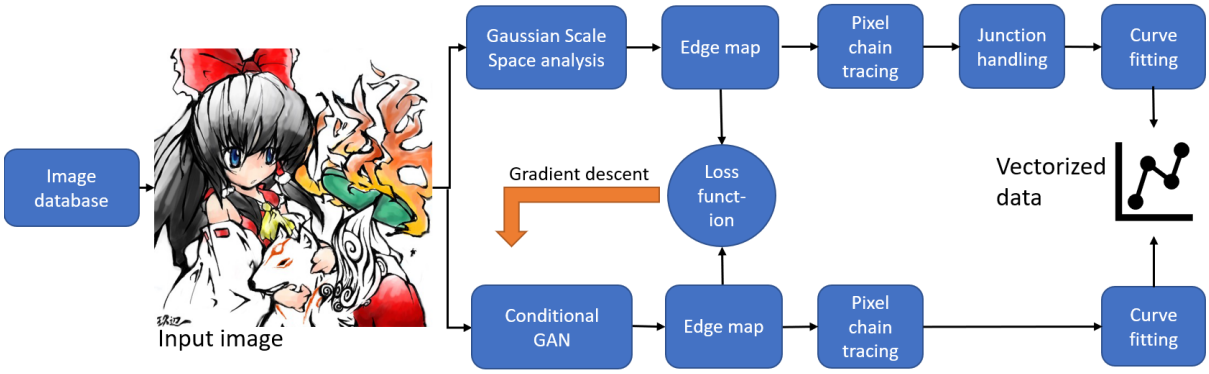


Figure 2: ***Overview of the DDC vectorizer.*** *Our framework consists of two parts, the vectorizer (top) and the re-constructor (bottom). In each part of the chart, the upper pipeline corresponds to our method while the lower pipeline corresponds to a conventional method [OBW*08]. The vectorizer takes an input image, extract relevant edges using a conditional-GAN, tracing pixels chains from the output of the cGAN, convert them into Bézier curves (vectorized data), and optionally save them into a JSON file. Since the performance bottleneck lies in the Gaussian Scale Space analysis process, we train a conditional-GAN by image pairs of (input image, edge map) to accelerate the whole process. Note that compared with the conventional method, our framework does not require handling junctions (i.e. the intersection of curves).*

Algorithm 1 Tracing pixel chains.

Input: an image **I** containing relevant edge pixels.
Output: an list of pixel chains.
pixelChains ← *empty*
chain ← *empty*
for row **y** in **I do**
  for column **x** in **I do**
    if $I(x,y) \neq 0$ **then**
      nextPixel ←2-ring neighbor pixels with lowest non-zero color values
      if *nextPixel* ≠ *empty* **then**
        pixelChains ← *FollowChain*(*nextPixel*, *pixelChains*, *chain*)
      end if
    end if
  end for
end for

the neighbors and removes the founded pixels on the image while adding the founded pixel to the pixel chains. We chose to follow the pixel with the lowest pixel value since it represents the sharpest edge. The pixel-chain tracing algorithm stops when all pixels are scanned. The result is a list of founded pixel chains.

For each pixel along the pixel chains, we sample left-side color and right-side color from the input image. The blur factors were sampled from the edge map directly. For each pixel in the pixel chains, a left-side color and a right-side color should be sampled from the distance $3 \times blur\ factor$, which covers 99% of the edge's contrast, assuming a Gaussian-shaped blur kernel [Eld99], from the normal direction and the inverted normal direction, respectively. After the pixel chains are found, we fit polylines for each pixel chain using the Potrace [Sel01] algorithm.

3.3. Reconstructor

Color gradients regression. The presumption of DC, i.e. Equation 1, can only model first-order color gradients. In contrast, we learn the mapping from color source map and binary edge map to target images. The objective of our model can be described using the following equation:

$$\underset{\theta}{\operatorname{argmin}} \|\Sigma_x f_\theta(s,e) - C_x\|_1 \quad (1)$$

where the function f represents the transform done by the generator of the conditional GAN, and the parameter θ are the weights to be learned by the generator; s, e are the color source map and the binary edge map respectively; C are target r-g-b images which subscripted by the sample index x. Our neural net basically minimizes the L_1 loss plus GAN loss between target images and generated images. We used a network architecture similar to pix2pix [IZZE17] for this task.

Training data. We used the color source map and binary edge map as the input, original image as the target to do aligned training. We used the Gaussian Scale Space analysis method described in [OBW*08] to extract an edge map where each pixel contains a value representing the blurriness of the edge. After getting the edge map, the binary edge map was constructed by binary thresholding the edge map, where each blur factor greater than and equal to zero was converted to one and zero respectively. The binary edge map was used to constrain the boundary of the color sources and help the reconstructor differentiate between pixels with black color and empty spaces.

The color source map was generated using Algorithm 2. This procedure iterates through each polyline; for each line segment along the polyline, 2 line segments are drawn at a distance computed from the blur factors interpolated along the line segment. Examples of color source maps can be seen in the third column of Figure 4. A binary edge map is generated by binary thresholding the edge map. We use a threshold value of 5 (255 is the maxi-

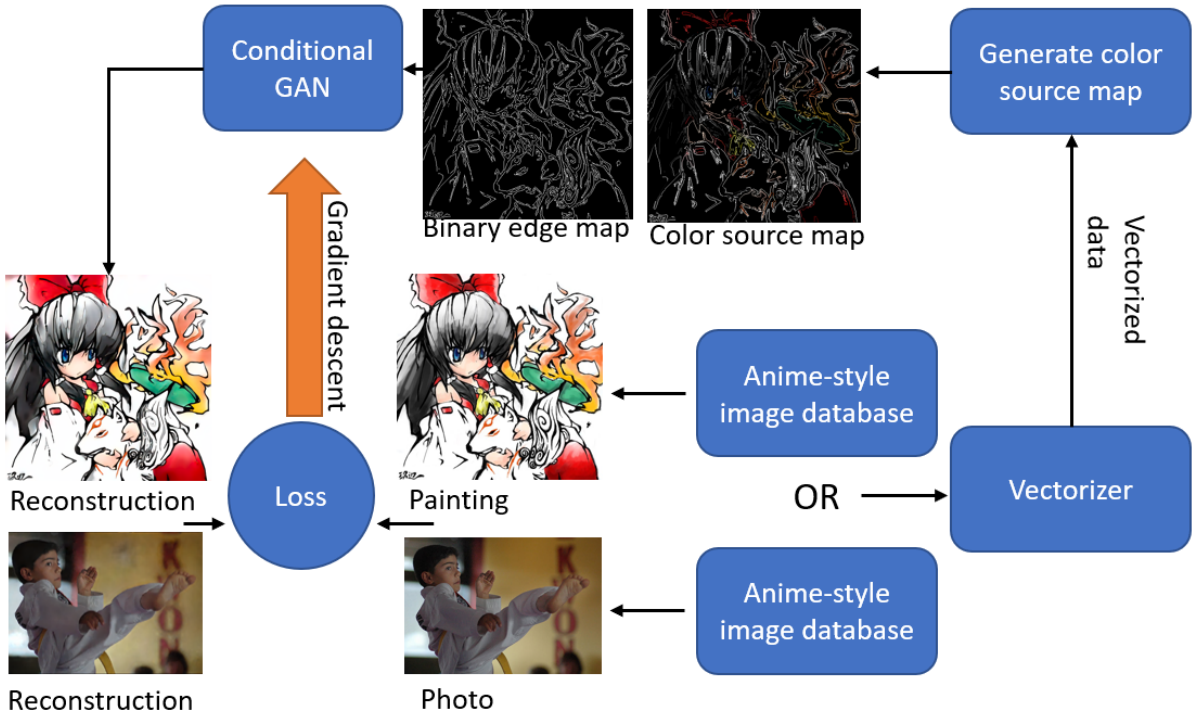


Figure 3: ***Overview of the DDC reconstructor.*** *The re-constructor takes the vectorized data, generates the color source map and the edge map, and use a conditional GAN to reconstruct the original image. The training data can come from any type of images, in our case, we trained the conditional GAN on anime-style painting database and photo database. Our model shows high reconstruction quality on both datasets.*

Algorithm 2 Generate color source map and binary edge map.

Input: a list of DDCs.
Output: color source map and binary edge map
$colorSourceMap \leftarrow$ *empty image*
$binaryEdgeMap \leftarrow$ *empty image*
for DDC **ddc** in **DDCs do**
  for line segments **s** in **ddc.polyline do**
    $normal \leftarrow computeNormal(s)$
    $blur \leftarrow$ the blur factor of this line segment
    $sigma \leftarrow 0.8 + 0.4 \times (blur - 1)$
    $(lp1, lp2) = (s.start - normal \times sigma, s.end - normal \times sigma)$
    $(rp1, rp2) = (s.start + normal \times sigma, s.end + normal \times sigma)$
    Draw a line segment (lp1,lp2) which colors are linear interpolated from lp1 to lp2 on **colorSourceMap**
    Draw a line segment (rp1,rp2) which colors are linear interpolated from rp1 to rp2 on **colorSourceMap**
    Draw a white line segment **s** on **binaryEdgeMap**
  end for
end for

mum color value) for the binary edge map. If the input is Bézier curves, the same generation procedure as described in [OBW*08] was used.

After we got the color source map and binary edge map, we use them as well as the target images C_x to train the DNN. Given the color source map s_x and the binary edge map e_x, the cGAN learns how to generate images as close to C_x as it can.

Inferencing. During the inferencing phase, the DDC reconstructor generates the binary edge map and color source map from vectorized data using Algorithm 1. Since the network architecture is fully convolutional, any image's width or height which is the power of 2 can be accepted. In general, the pre-processing and post-processing only involve conversion from r-g-b color space to L-a-b color-space and reverse; padding and cropping can be added in the pre-processing or post-processing if the input image dimension doesn't match requirements.

4. Implementation Details

The DDC framework was implemented and trained by C++ and PyTorch respectively on a machine with Intel i7-7700k CPU and Nvidia 1080Ti GPU. The edge map generator used a network similar to UNET, while the reconstructor used an architecture similar to pix2pix. The generator of the vectorizer composed of 5 convolution layers followed 5 strided-convolution layers, batch-norms was

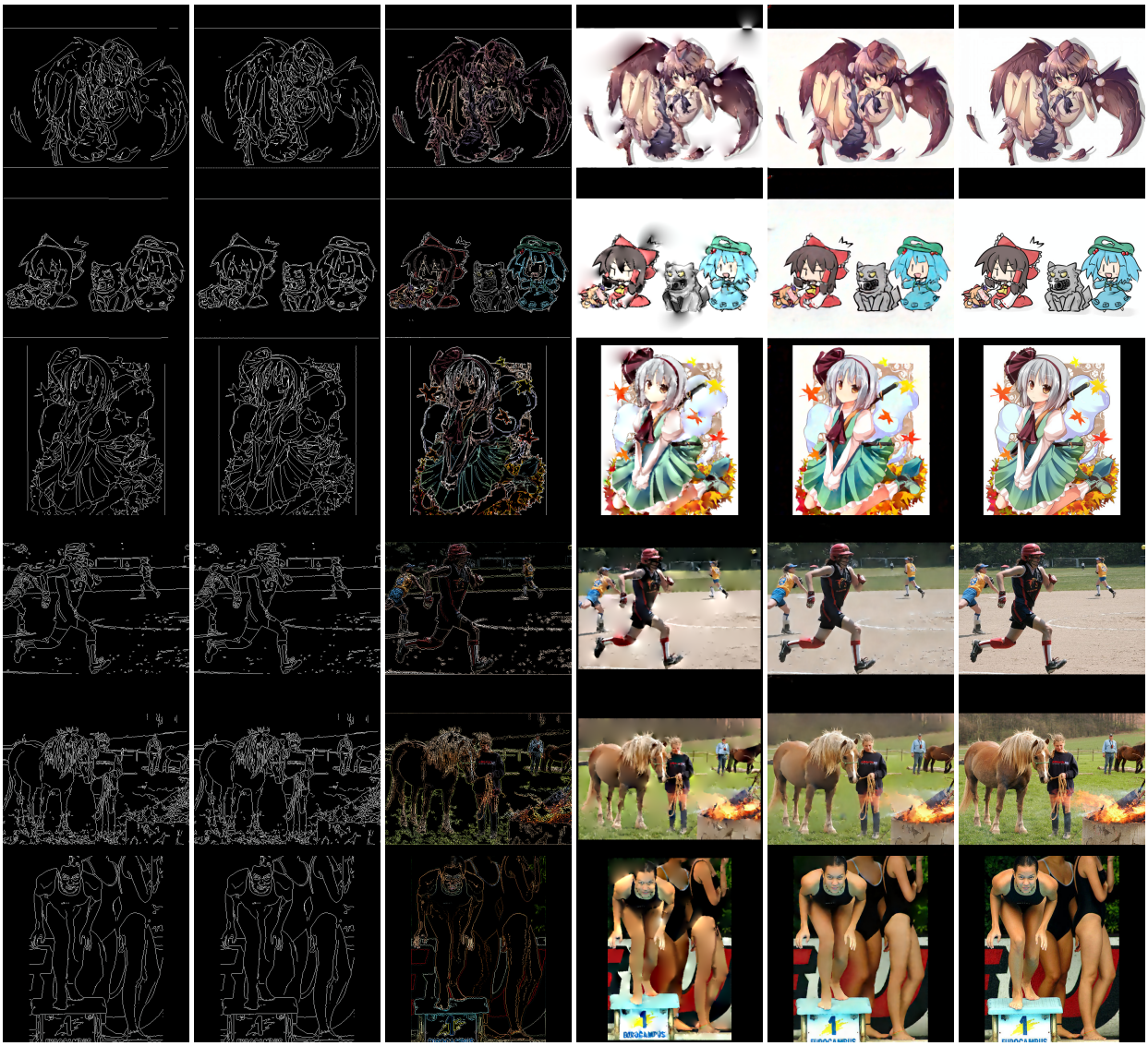


Figure 4: ***Results.*** *(From left to right): edge maps (DC), edge maps (DDC), color source maps (DDC), reconstructed images (DC), reconstructed images (DDC), input images.*

used for all layers except the output layer. The discriminator was a 3-layer convolution network 70 by 70 patch discriminator.

The reconstructor had a similar structure to pix2pix, it was composed of 3 convolution layers followed by 9 Resnet blocks followed by 3 up-sampling and convolution layers. Instance norms [UVL16] are used in all layers except the output layer. Both of the vectorizer and reconstructor were trained using LSGAN [MLX*17].

For pre-processing images for edge map generator training, a 3 by 3 kernel was used for the bilateral filter. For each image, a 10-level Gaussian Scale Space was built and edges are extracted from each level. In the edge filtering process, we filtered out all edges less than 3 pixels long. A 5 by 5 pixels window was used for pixel chain tracing. For reconstructor, the input image was padded 40 pixels along the image border. Input images were converted from RGB color space to LAB color space, on the other hand, output images were converted back to RGB color spaces and cropped to its original size. Besides padding, cropping and color space conversion, no other pre-processing or post-processing were used in the reconstructor.

We trained our networks for two styles, real-world photo, and anime-style paintings. For the photo style, we used the Flickr image dataset [Hsa18] for training and testing. We randomly selected 2783 photos for training and 92 photos for testing. For the anime-style painting style, we used the Danbooru [AcB20] dataset, which is the biggest crowd-curated anime-style painting data set openly accessible online. For the Danbooru dataset, we used a specific

	No. of curves	Edge map extraction time (ms)	Post-processing time (ms)	Re-construction (ms)	Re-construction error (RMSE)
Fig.1 row1	680 / 653	6 / 27106	192 / 422	14 / 5848	0.11 / 0.16
Fig.1 row2	- / 1024	- / 27902	- / 755	14 / 5134	0.06 / 0.08
Fig.2	1116 / 486	12 / 27552	201 / 805	14 / 8221	0.23 / 0.37
Fig.3 photo	- / 798	- / 28305	- / 482	14 / 4391	0.09 / 0.14
Fig.4 row1	728 / 710	6 / 27800	163 / 421	17 / 5340	0.14 / 0.22
Fig.4 row2	624 / 497	6 / 28197	162 / 294	14 / 5116	0.19 / 0.25
Fig.4 row3	1011 / 800	6 / 27720	205 / 534	13 / 4515	0.18 / 0.22
Fig.4 row4	- / 616	- / 27352	- / 327	14 / 4224	0.09 / 0.14
Fig.4 row5	- / 1145	- / 28077	- / 739	13 / 5256	0.11 / 0.15
Fig.4 row6	- / 495	- / 28642	- / 397	13 / 4197	0.09 / 0.16
Fig.7 row1	- / 466	- / 26921	- / 434	13 / 3861	0.06 / 0.09
Fig.7 row2	- / 497	- / 25348	- / 393	13 / 4004	0.09 / 0.08
Fig.8 row1	- / 615	- / 27504	- / 483	13 / 4109	0.05 / 0.07
Fig.8 row2	- / 1577	- / 30179	- / 1276	13 / 12033	0.05 / 0.06
Fig.8 row3	- / 864	- / 27385	- / 535	13 / 4694	0.06 / 0.07
Fig.9	698 / 434	6 / 25356	178 / 303	14 / 4867	0.14 / 0.19
Fig.10	674 / 531	6 / 26312	151 / 353	13 / 5967	0.14 / 0.29
Fig.13 row1	1012 / 713	13 / 28198	259 / 493	16 / 7528	0.20 / 0.24
Fig.13 row2	375 / 344	6 / 25519	139 / 211	14 / 4456	0.13 / 0.18
Fig.13 row3	745 / 659	6 / 31393	164 / 471	15 / 4973	0.13 / 0.18
Fig.13 row4	750 / 593	6 / 26972	173 / 362	16 / 4545	0.14 / 0.23

Table 1: ***Performance and statistics of DDC versus DC.*** *In each table entry the first number shows the results of DDC; the second number shows the results of DC. Since we use the same color source map and edge map as DC for photo test images, those entries are denoted with "-".*

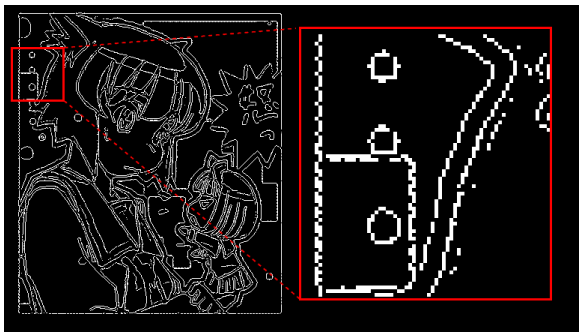


Figure 5: ***Example of noisy edge map.*** *The image shows the raw output of the vectorizer network. Jaggy edges and random-dot pixel patterns exist in the magnifying area, making pixel-chain tracing difficult.*

branch, in which each image was already cropped or padded to 512 by 512 pixels. In this branch, around 3 million images were randomly separated into 200 folders, and each folder contains about 2100 images. We randomly choose two sub-folders, 0150 and 0147, for training and evaluation, respectively from the dataset. The training data was augmented by flipping and random cropping input images to 256 by 256 pixels. In total, around 20 thousand image pairs were used to train both the edge map generator and reconstructor. For both of them, we used an ADAM optimizer [KB14] with a learning rate of 0.0002 for 1st to 100th epochs and 0.0001 for 101st to 200th epochs. The loss function for the generator networks was set to use the GAN loss plus L_1-regulations, while for the discriminators, the mean square error function was used. The training took about 4 days for both the edge map generator and the reconstructor. We plan to open-source our codes after the work is completed.

5. Results

In this section, we compare performance and extracted primitive statistics between our DDC framework and the pure DC vectorization framework. The implementation of the pure DC vectorization framework was generously provided by Orzan et al. [OBW*08], and was written in C++, all results are generated on a machine with Intel Core i7-8750H CPU and Nvidia GeForce GTX 1060M GPU with 6GB VRAM.

5.1. Reconstruction

Figure 4 and Figure 13 show the reconstruction results of DDC v.s. DC with their root-mean-square-error (RMSE). The edge maps and color source maps for anime-style paintings are all generated using our accelerated vectorizer, in contrast, the inputs to the reconstructor for nature images were generated by the Gaussian Scale Space analysis process. This choice is because edge feature estimating for nature image is a far more difficult problem than estimating edge features for anime-style paintings, and edge extraction is not our major concern, we leave this part for future work.

The results showed that among 20 test images, 19 images showed lower reconstruction error using DDCs regardless of image styles. Besides, the outputs had perceivable higher visual quality than the baseline. The reconstructed images of the baseline method having lots of perceivable artifacts around object contours. Additionally, colors leaking happened frequently in the reconstruction results of the baseline. This may due to the limitation that the baseline method can't handle edges with less than 3-pixel width. Moreover, the quality of high-frequency contents are significantly better using our framework, for example, hair and furs in Figure 4 row 5, and human faces in Figure 4 row 6. The only case where DDC failed is an under-water photo that contains large blurry regions; we'll discuss in detail in the limitation section.

The reconstruction time for DDC is around $10 \sim 20$ milliseconds which is at least 3 order-of-magnitudes faster than DC. Although DC takes a much longer time for reconstruction, ranging from around 4 to 6 seconds, this may due to the provided implementation used an unoptimized Poisson solver. Several fast solvers for DC has been reported having real-time performance.

5.2. Vectorization

Table 1 shows the performance statistics of our DDC framework for all the images shown in this paper. Note that we only apply our vectorizer to anime-style paintings. The test images are all padded or cropped to 512 by 512 pixels.

In terms of the number of curves extracted, the results show that the edge map generator extracted slightly more curves than the baseline methods. This is due to the noisy output in the extracted edge map. In edge maps extracted by DDC, while the localization of edge is similar to edges extracted by the baseline, there exist many jaggy or random-dots patterns in the extracted edge map. See Figure 5 for an example for a noisy edge map. This made the pixel chain tracing more difficult in our framework than in the baseline.

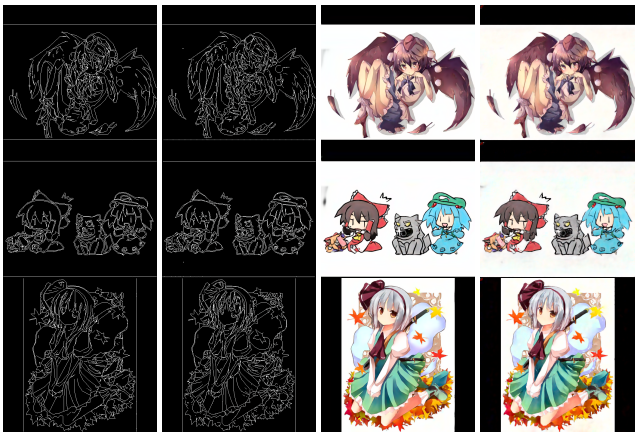


Figure 6: ***Results.*** *(From left to right): edge maps generated using Gaussian Scale Space Analysis (DC), edge maps generated by the DDC vectorizer (DDC), reconstructed results using edge maps (DC), reconstructed results using edge maps (DDC).*

The edge map generation of the DDC framework is at least 3 orders-of-magnitudes faster than the DC. Compared with the post-processing (pixel chain tracing and polyline fitting) time, the edge map extraction time is almost negligible. Furthermore, the post-processing time required for DDC is also shorter than DC, this is because that DDC doesn't require junction handling in the post-processing. The DDC reconstructor is tolerant to small amounts of noises, and since disabling the junction handling only generate a small amount of noise, we decided to ignore the junction handling part.

Figure 6 shows the reconstruction results using our edge map generator versus using the Gaussian Scale Space analysis approach. Although our learned edge map generator gives noisy inputs, the visual quality of the reconstruction remains good while the edge map-generating times are significantly lower than the Gaussian Scale Space analysis process.

In conclusion, the results show that the DDC reconstructor can learn (low-frequency) color gradients style from photo dataset and anime-style painting dataset and runs 3 order-of-magnitude faster; the DDC vectorizer having order-of-magnitude performance improvements over the baseline method without sacrificing visual qualities for anime-style paintings.

5.3. Comparisons

In this section, we compare our method with the latest vectorization methods, Hierarchical Diffusion Curves (HDC) [XSTN14] and Inverse Diffusion Curves (IDC) [ZDZ17] for curve-based vector graphics. We have two disclaimers for the results. First, because we do not have the source codes of the methods in comparison, we compare our results with the results extracted from the published paper. Second, we decided to only compare the perceived visual quality since the accurate and fair quantitative comparison is not possible. Also, since HDC and IDC are not designed for anime-style paintings, we only compare the visual quality of reconstructed nature images.

Figure 7: ***Comparison with DC, HDC and IDC.*** *(Columns from left to right): reference input, Diffusion Curves (DC) reconstruction, Hierarchical Diffusion Curves (HDC) reconstruction, Inverse Diffusion Curves (IDC) reconstruction, Deep Diffusion Curves (DDC) reconstruction. Note that results for HDC and IDC were extracted from their papers.*

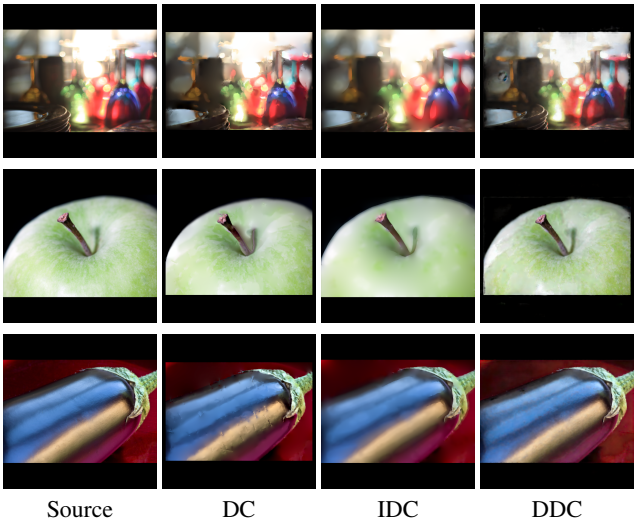


Figure 8: ***Comparison with DC and IDC*** *(Columns from left to right): reference input, Diffusion Curves (DC) reconstruction, Inverse Diffusion Curves (IDC) reconstruction, Deep Diffusion Curves (DDC) reconstruction. Note that results for HDC and IDC were extracted from their papers.*

Figure 7 and Figure 8 shows reconstructed images for DC, HDC, IDC and DDC. In general, all methods have shown high visual quality for reconstructions. The HDC gives the best results in Figure 7, however, based on the information disclosed in their paper, HDC extract the highest number of primitives compared to IDC and DC, on the other hand, IDC extracts the fewest primitives in most cases and also maintain high reconstruction quality. The reconstructed images of IDC tend to be more abstract, besides, boundaries are relatively sharp in IDC's results even when blur pass applied. The results of DDC is of similar quality to other methods except for the dolphin image. DDC performed best on (relatively)

high-frequency details, see the texture detail of the green apple in Figure 8 row 2 for example. Another advantage of DDC is runtime performance. The reconstruction time of DDC is within 20 ms while all other DC-variants reported several hundred milli-seconds reconstruction time.

In summary, each method in comparison has its own merits, and we can't (and don't want to) draw a clear conclusion on which method is the best.

5.4. Applications

To demonstrate the applicability of the DDC framework, we built a prototypical web-based authoring tool for the user to do basic editing. We show the results of shape editing, color editing, and object creation from scratch in this section.

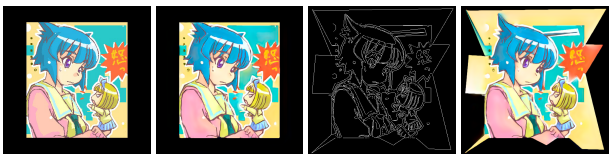


Figure 9: ***Shape editing of vectorized images.*** *The upper-left is the input image uploaded by the user; the upper-right image is the initial reconstruction result; the bottom-left image is the geometry after shape editing; the bottom-right image is the final results. The whole process took 1 minute and 32 seconds.*

Shape editing. Figure 9 shows the process of shape editing. The user first used the client application to upload an image to our server, the server then generated and returned the vectorized results to the client. The user then used the "move point" tool to adjust control point locations. The client detected content changes, and ask the server to reconstruct a new image to show to the user. The shape editing session takes 1 minute 32 seconds to finish.

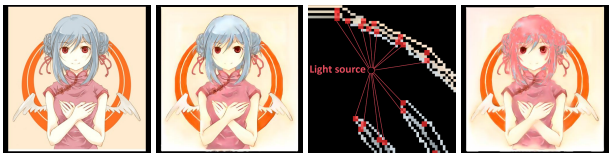


Figure 10: ***Color editing of vectorized images.*** *The upper-left is the input image uploaded by the user; the upper-right image is the initial reconstruction result; the bottom-left image shows the "light source brush"; the light source brush works by casting rays from the cursor position to nearby control points to determine their visibility, only control points with positive visibility will be affected by the brush color. In this example, only the light blue colors are modified to peach colors. The bottom-right image is the final result. The whole process took 1 minute and 10 seconds.*

Color editing. Figure 10 shows the process of color editing. The process is similar to shape-editing. Since color editing point by point is tedious, we provide the "light source brush" to facilitate color-editing. The light source brush works by casting rays to nearest control points within a range from the cursor position to determine the visibility of the control points from the cursor's point of view. This tool could reduce the amount of time required for regional color editing. The color editing session takes 1 minute and 10 seconds to finish.

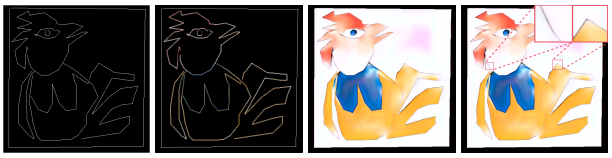


Figure 11: ***Object creation from scratch.*** *The upper-left image shows object contours created by the user from scratch using the polyline tool; the user then fills in the color sources as shown in the upper-right image; the final result was shown in the lower-left image. The whole process took about 15 minutes. The lower-right image shows magnifying areas where stroke effects are automatically generated. Note that in this example, each polyline created the "stroke" effects for object contours, no explicit black color was assigned on the object contour polylines, the black stroke effect was learned from training data.*

Object creation from scratch. Figure 11 shows the process of object creation from scratch. The workflow is similar to most vector art authoring tools. The artist first design object contours, and fill in colors using control point level editing or the light source brush. The creation session took around 15 minutes to complete.

The results of object creation from scratch showed a feature of our system: stroke effect can be created using a single curve or polyline without any post-processing. Using Diffusion Curves and its variants, this stroke effect is not possible without post-processing. In contrast, our reconstructor learned from the data that a single curve often represents contour stroke, thus users could benefit from this feature and use it to create anime-style artworks.

6. Limitations and future works

Although the results showed that DDC could efficiently learn color shadings for both nature image and anime-style paintings, still, some limitations exist. In this section, we discuss some observed limitations and shed light on how to improve the results.

Figure 12 shows some typical less satisfying cases. Our method failed mainly two types of contents: large regional gradients and de-focused areas such as backgrounds. Figure 12 row 1 shows an image that exhibits large regional gradients in the wall region. This may due to the sparseness of training data. Figure 12 row 2 and Figure 7 show pictures of blurry backgrounds. The reason for the poor reconstruction may also be the imbalanced training data; since de-focused regions typically contain many blurry edges, and blurry edges in our training data are far less than high-frequency edges, as a result, blurry edges are not extracted properly.

Additionally, our model can not reconstruct high-frequency details with a reasonable number of primitives. Since our reconstruction results are highly dependent on edge maps and color source maps, if we extract too much edge, the resulting vectorization won't

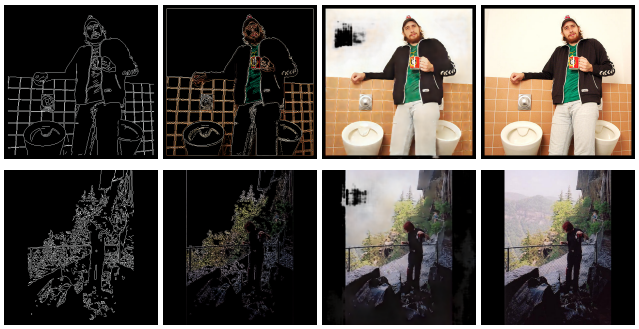


Figure 12: ***Less satisfying cases.*** *(Columns from left to right): Color source maps, edge maps, reconstructions, reference inputs.*

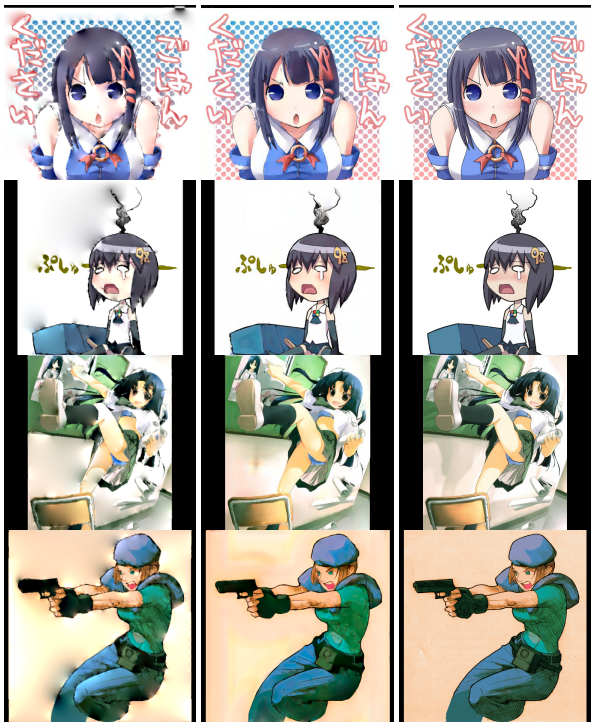
be editable. On the other hand, if we extract too few edges, our model can't reconstruct texture details. This is a dilemma faced in most vector graphics works. We suggest to decompose the image into low-frequency shading and texture details, and learn texture regression function [SWWW15] for textures.

Another limitation is that the neural network we have learned has low variation and lost the generative power of GANs. Since the attributes are not explicitly encoded, the conditional-GAN essentially becomes a one-to-one mapping function regressor. To improve the generative power, attributes and styles must be explicitly encoded in latent layers. We leave this improvement as a future work.

Despite the limitations mentioned above, our work still has much room for improvement. Although we have shown that our framework could efficiently learn the low-frequency color gradients in nature images and anime-style paintings, other kinds of vector graphics, e.g. Biharmonic Diffusion Curves, provide bi-harmonic color gradients tools such as Poisson Region for vector graphics authoring. We would like to apply similar data-oriented machine-learning-based approaches to vectorizing images for other kinds of vector graphics in the future. Furthermore, we would like to explore the effect of intrinsic image decomposition on vector graphics, for example, by decomposing the image into low-frequency shading and texture details, the low-frequency component could be easily vectorized while the texture details could be encoded and synthesized, providing user high-fidelity and high editability at the same time.

7. Conclusions

To meet the growing demand for non-linear color gradient authoring using vector graphics, this paper presents a powerful, efficient vector graphics framework that is capable of batch-vectorizing different style of images efficiently and faithfully reconstruct the original images. At the core lies the Deep Diffusion Curve, our DNN-based models for image reconstruction. By training the neural networks on a large data set, our framework has quite a few merits over the existing solutions. First, our method can learn low-frequency color shadings directly from data, bypassing the need to design PDEs and their solvers. Our framework does not rely on optimization or parameter fitting techniques, for anime-style paintings our



DC DDC Source

Figure 13: ***Additional results.*** *(From left to right): reconstructed images (DC), reconstructed images (DDC), input images.*

vectorizer benefits from the efficient inference performance, and holds order-of-magnitude performance improvement over the baseline. Meanwhile, our reconstructor is attribute-controllable, providing a powerful authoring tool for vector graphics. Finally, our work provides a machine-learning-based perspective for vector graphics, which has rarely seen on previous works and opens a broad avenue for machine-learning-based vector graphics applications.

References

[AcB20] ANONYMOUS, COMMUNITY D., BRANWEN G.: Danbooru2019: A large-scale crowdsourced and tagged anime illustration dataset. https://www.gwern.net/Danbooru2019, January 2020. Accessed: DATE. 3, 7

[BBG12] BOYÉ S., BARLA P., GUENNEBAUD G.: A vectorial solver for free-form vector gradients. *ACM Transactions on Graphics (TOG) 31*, 6 (2012), 173. 3

[BEDT10] BEZERRA H., EISEMANN E., DECARLO D., THOLLOT J.: Diffusion constraints for vector graphics. In *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering* (New York, NY, USA, 2010), NPAR '10, Association for Computing Machinery, p. 35–42. 3

[BLW11] BOWERS J. C., LEAHEY J., WANG R.: A ray tracing approach to diffusion curves. *Computer Graphics Forum 30*, 4 (2011), 1345–1352. 3

[CMW*18] CI Y., MA X., WANG Z., LI H., LUO Z.: User-guided deep anime line art colorization with conditional adversarial networks. *CoRR abs/1808.03240* (2018). 4

[Eld99] ELDER J. H.: Are edges incomplete? *International Journal of Computer Vision 34*, 2-3 (1999), 97–122. 5

[Far93] FARIN G.: *Curves and surfaces for computer aided geometric design (3rd ed.): a practical guide*. Academic Press, 1993. 2

[Fer64] FERGUSON J.: Multivariable curve interpolation. *J. ACM 11*, 2 (Apr. 1964), 221–228. 2

[FSH11] FINCH M., SNYDER J., HOPPE H.: Freeform vector graphics with controlled thin-plate splines. *ACM Trans. Graph. 30*, 6 (Dec. 2011), 1–10. 3

[HA17] HENSMAN P., AIZAWA K.: cgan-based manga colorization using a single training image. *CoRR abs/1706.06918* (2017). 3

[Hsa18] HSANKESARA: Flickr image dataset. https://www.kaggle.com/hsankesara/flickr-image-dataset, 2018. 7

[HSF*20] HOU F., SUN Q., FANG Z., LIU Y., HU S., QIN H., HAO A., HE Y.: Poisson vector graphics (pvg). *IEEE Transactions on Visualization and Computer Graphics 26*, 2 (2020), 1361–1371. 2, 3

[HTL*18] HAMADA K., TACHIBANA K., LI T., HONDA H., UCHIDA Y.: Full-body high-resolution anime generation with progressive structure-conditional generative adversarial networks. *CoRR abs/1809.01890* (2018). 3

[IKCM13] ILBERY P., KENDALL L., CONCOLATO C., MCCOSKER M.: Biharmonic diffusion curve images from boundary elements. *ACM Trans. Graph. 32*, 6 (Nov. 2013), 219:1–219:12. 3

[IZZE17] ISOLA P., ZHU J.-Y., ZHOU T., EFROS A. A.: Image-to-image translation with conditional adversarial networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), pp. 5967–5976. 5

[JCW09] JESCHKE S., CLINE D., WONKA P.: A gpu laplacian solver for diffusion curves and poisson image editing. *ACM Trans. Graph. 28*, 5 (Dec. 2009), 116:1–116:8. 3

[JCW11] JESCHKE S., CLINE D., WONKA P.: Estimating color and texture parameters for vector graphics. *Comput. Graph. Forum 30* (2011), 523–532. 3

[Jes16] JESCHKE S.: Generalized diffusion curves: An improved vector representation for smooth-shaded images. *Computer Graphics Forum 35* (2016), 71–79. 3

[JZL*17] JIN Y., ZHANG J., LI M., TIAN Y., ZHU H., FANG Z.: Towards the automatic anime characters creation with generative adversarial networks. *CoRR abs/1708.05509* (2017). 3

[KB14] KINGMA D. P., BA J.: Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014). 8

[KKKL19] KIM J., KIM M., KANG H., LEE K.: U-GAT-IT: unsupervised generative attentional networks with adaptive layer-instance normalization for image-to-image translation. *CoRR abs/1907.10830* (2019). 3

[Lei17] LEI J.: Animegan. https://github.com/jayleicn/animeGAN, 2017. 3

[LHFY12] LIAO Z., HOPPE H., FORSYTH D., YU Y.: A subdivision-based representation for vector image editing. *IEEE Transactions on Visualization and Computer Graphics 18*, 11 (2012), 1858–1867. 3

[Lin94] LINDEBERG T.: Scale-space theory: A basic tool for analyzing structures at different scales. *Journal of applied statistics 21*, 1-2 (1994), 225–270. 2

[LL06] LECOT G., LEVY B.: Ardeco: Automatic region detection and conversion. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques* (Goslar, DEU, 2006), EGSR '06, Eurographics Association, p. 349–360. 3

[LTKD15] LIENG H., TASSE F., KOSINKA J., DODGSON N. A.: Shading curves: Vector-based drawing with explicit gradient control. *Computer Graphics Forum 34*, 6 (2015), 228–239. 3

[MLX*17] MAO X., LI Q., XIE H., LAU R. Y. K., WANG Z., SMOLLEY S. P.: Least squares generative adversarial networks. In *2017 IEEE International Conference on Computer Vision (ICCV)* (2017), pp. 2813–2821. 7

[MO14] MIRZA M., OSINDERO S.: Conditional generative adversarial nets. *CoRR abs/1411.1784* (2014). 3

[OBW*08] ORZAN A., BOUSSEAU A., WINNEMÖLLER H., BARLA P., THOLLOT J., SALESIN D., ORZAN A., BOUSSEAU A., WINNEMÖLLER H., BARLA P., THOLLOT J., SALESIN D.: Diffusion curves: A vector representation for smooth-shaded images. *ACM Trans. Graph. 27*, 3 (Aug. 2008), 1–8. 2, 3, 4, 5, 6, 8

[Pre17a] PREFERED NETWORKS: Crypko.ai. https://crypko.ai, 2017. 3

[Pre17b] PREFERED NETWORKS: Paintchainer. https://paintschainer.preferred.tech/index.html, 2017. 4

[QWH06] QU Y., WONG T.-T., HENG P.-A.: Manga colorization. *ACM Trans. Graph. 25*, 3 (July 2006), 1214–1220. 3

[RFB15] RONNEBERGER O., FISCHER P., BROX T.: U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015* (Cham, 2015), Springer International Publishing, pp. 234–241. 4

[Sel01] SELINGER P.: Potrace. http://potrace.sourceforge.net/, 2001. 5

[SLWS07] SUN J., LIANG L., WEN F., SHUM H.-Y.: Image vectorization using optimized gradient meshes. *ACM Trans. Graph. 26* (07 2007), 11. 2, 3

[SMYA14] SATO K., MATSUI Y., YAMASAKI T., AIZAWA K.: Reference-based manga colorization by graph correspondence using quadratic programming. In *SIGGRAPH Asia 2014 Technical Briefs* (New York, NY, USA, 2014), SA '14, ACM, pp. 15:1–15:4. 3, 4

[SWWW15] SONG Y., WANG J., WEI L.-Y., WANG W.: Vector regression functions for texture compression. *ACM Trans. Graph. 35*, 1 (2015), 5. 3, 11

[tdr16] TDRUSSELL: Illustrationgan. https://github.com/tdrussell/IllustrationGAN, 2016. 3

[UVL16] ULYANOV D., VEDALDI A., LEMPITSKY V.: Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022* (2016). 7

[War10] WARREN J.: *Solving Diffusion Curves on GPU*. PhD thesis, University of Dublin, Trinity College, 2010. 3

[Wor18] WORLD WIDE WEB CONSORTIUM: Scalable vector graphics (svg) 2. https://www.w3.org/TR/SVG/, 2018. 2

[XLY09] XIA T., LIAO B., YU Y.: Patch-based image vectorization with automatic curvilinear feature alignment. *ACM Trans. Graph. 28*, 5 (2009), 115. 3

[XSTN14] XIE G., SUN X., TONG X., NOWROUZEZAHRAI D.: Hierarchical diffusion curves for accurate automatic image vectorization. *ACM Trans. Graph. 33*, 6 (2014), 230. 2, 3, 9

[YHL*17] YAO C., HUNG S., LI G., CHEN I., ADHITYA R., LAI Y.: Manga vectorization and manipulation with procedural simple screentone. *IEEE Transactions on Visualization and Computer Graphics 23*, 2 (Feb 2017), 1070–1084. 4

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

[ZDZ17] ZHAO S., DURAND F., ZHENG C.: Inverse diffusion curves using shape optimization. *IEEE Transactions on Visualization and Computer Graphics 24*, 7 (2017), 2153–2166. 2, 3, 9

[ZLW*18] ZHANG L., LI C., WONG T.-T., JI Y., LIU C.: Two-stage sketch colorization. *ACM Trans. Graph. 37*, 6 (Dec. 2018), 261:1–261:14. 4