

## Diffusion Curves: A Vector Representation for Smooth-Shaded Images

Alexandrina Orzan, Adrien Bousseau, Holger Winnemöller, Pascal Barla, Joëlle Thollot, David Salesin

► **To cite this version:**

Alexandrina Orzan, Adrien Bousseau, Holger Winnemöller, Pascal Barla, Joëlle Thollot, et al.. Diffusion Curves: A Vector Representation for Smooth-Shaded Images. ACM Transactions on Graphics, Association for Computing Machinery, 2008, Special Issue: Proceedings of ACM SIGGRAPH 2008, 27 (3), pp.92:1-8. 10.1145/1399504.1360691 . inria-00274768

**HAL Id: inria-00274768**

**<https://hal.inria.fr/inria-00274768>**

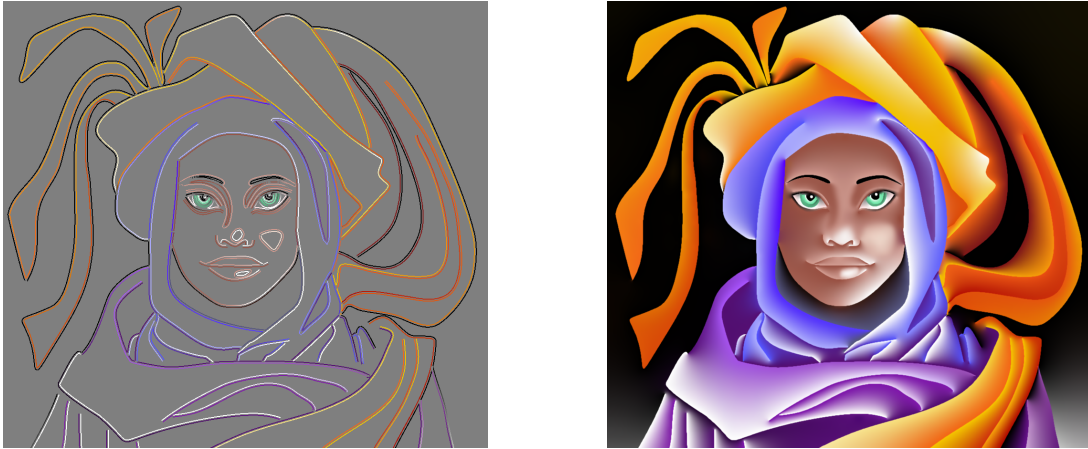
Submitted on 14 Jun 2011

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Diffusion Curves: A Vector Representation for Smooth-Shaded Images

Alexandrina Orzan<sup>1,2</sup> Adrien Bousseau<sup>1,2,3</sup> Holger Winnemöller<sup>3</sup> Pascal Barla<sup>1</sup> Joëlle Thollot<sup>1,2</sup> David Salesin<sup>3,4</sup>  
<sup>1</sup>INRIA <sup>2</sup>Grenoble University <sup>3</sup>Adobe Systems <sup>4</sup>University of Washington



**Figure 1:** Diffusion curves (left), and the corresponding color image (right). Note the complex shading on the folds and blur on the face.

## Abstract

We describe a new vector-based primitive for creating smooth-shaded images, called the *diffusion curve*. A diffusion curve partitions the space through which it is drawn, defining different colors on either side. These colors may vary smoothly along the curve. In addition, the sharpness of the color transition from one side of the curve to the other can be controlled. Given a set of diffusion curves, the final image is constructed by solving a Poisson equation whose constraints are specified by the set of gradients across all diffusion curves. Like all vector-based primitives, diffusion curves conveniently support a variety of operations, including geometry-based editing, keyframe animation, and ready stylization. Moreover, their representation is compact and inherently resolution-independent. We describe a GPU-based implementation for rendering images defined by a set of diffusion curves in realtime. We then demonstrate an interactive drawing system for allowing artists to create artworks using diffusion curves, either by drawing the curves in a freehand style, or by tracing existing imagery. The system is simple and intuitive: we show results created by artists after just a few minutes of instruction. Furthermore, we describe a completely automatic conversion process for taking an image and turning it into a set of diffusion curves that closely approximate the original image content.

**CR Categories:** I.3.3 [COMPUTER GRAPHICS]: Picture/Image Generation, Graphics Utilities;

**Keywords:** Vector graphics, vectorization, gradient mesh, color diffusion, image creation, image reconstruction

## 1 Introduction

*Vector graphics*, in which primitives are defined geometrically, dates back to the earliest days of our field. Early cathode-ray tubes, starting with the Whirlwind-I in 1950, were all vector displays, while the seminal Sketchpad system [Sutherland 1980] allowed users to manipulate geometric primitives like points, lines, and curves. *Raster graphics* provides an alternative representation for describing images via a grid of pixels rather than geometry. Raster graphics arose with the advent of the framebuffer in the 1970s and is now commonly used for storing, displaying, and editing images. However, while raster graphics offers some advantages over vector graphics — primarily, a more direct mapping between the hardware devices used for acquisition and display of images, and their internal representation — vector graphics continues to provide certain benefits as well. Most notably, vector graphics offers a more compact representation, resolution-independence (allowing scaling of images while retaining sharp edges), and geometric editability. Vector-based images are more easily animated (through keyframe animation of their underlying geometric primitives), and more readily stylized (e.g. through controlled perturbation of geometry). For all of these reasons, vector-based drawing tools, such as Adobe Illustrator<sup>®</sup>, CorelDraw<sup>®</sup>, and Inkscape<sup>®</sup>, continue to enjoy great popularity, as do standardized vector representations, such as Flash and SVG.

However, for all of their benefits, vector-based drawing tools offer only limited support for representing complex color gradients, which are integral to many artistic styles. For example, the art movements of realism and hyperrealism rely on smooth gradients to achieve soft shadows, defocus blur, diffuse shading, glossy reflections, and various material effects. The *airbrush* technique, widely used in design and urban art, is fundamentally based on (physical) color diffusion. The *Art Deco* painting movement, various comic styles, as well as numerous painting styles also heavily feature color gradients. While graphic artists, such as Yukio Miyamoto<sup>1</sup>, have been able to achieve stunning results using existing vector-graphics tools, creating these artworks requires a high degree of artistic skill, not to mention an enormous degree of patience (the reported creation times for some art pieces run into the hundreds of hours).

<sup>1</sup>[www.khulsey.com/masters\\_yukio\\_miyamoto.html](http://www.khulsey.com/masters_yukio_miyamoto.html)

What makes creating this artwork so difficult using vector primitives is that most existing vector formats support only linear or radial gradients<sup>2</sup>. Currently, the most sophisticated vector-based tool for handling complex gradients is the *gradient mesh*. A gradient mesh is a lattice with colors at each vertex that are linearly interpolated across the mesh. While more powerful than simple gradients (or “ramps”), gradient meshes still suffer from some limitations. A significant one is that the topological constraints imposed by the mesh lattice give rise to an overcomplete representation that becomes increasingly inefficient and difficult to create and manipulate. Recently, Sun et al. [2007] presented an important advance: a semi-automatic method for optimizing a manually initialized mesh. While their approach lessens the manual demands on the user, it does not address the process of subsequent gradient-mesh manipulation, nor does it facilitate free-hand creation of gradient meshes from scratch.

In this paper we propose a novel vector-graphics primitive, called the *diffusion curve*. A diffusion curve is a curve that diffuses colors on both sides of the space that it divides. The motivations behind such a representation are twofold:

First, this representation supports traditional freehand drawing techniques. Artists often begin by sketching shapes, then adding color later. In a typical drawing session with our tool, the artist first lays down drawing curves corresponding to color boundaries. In contrast with traditional vector graphics, the color boundaries do not need to be closed and may even intersect. A diffusion process then creates smooth gradients on either side of the curves. By specifying blur values along a curve, the artist can also create smooth color transitions across the curve boundaries.

Second, most color variations in an image can be assumed to be caused by edges (material and depth discontinuities, texture edges, shadow borders, etc.) [Koenderink and Doorn 1979; Marr and Hildreth 1980]. Even subtle shading effects can be modeled as though caused by one or more edges, and it has been demonstrated that edges constitute a near-complete and natural primitive for encoding and editing images [Carlsson 1988; Elder 1999; Elder and Goldberg 2001]. In this work, we rely on vision algorithms, such as edge detection, to convert an image into our diffusion curves representation fully automatically.

The main contribution of this paper is therefore the definition of diffusion curves as a fundamental vector primitive, along with two types of tools: (1) A prototype allowing manual creation and editing of diffusion curves. Thanks to an efficient GPU implementation, the artist benefits from instant visual feedback despite the heavy computational demands of a global color diffusion. (2) A fully automatic conversion from a bitmap image based on scale-space analysis. The resulting diffusion curves faithfully represent the original image and can then be edited manually.

## 2 Previous work

We review here existing techniques to create complex color gradients and blur with modern vector graphic tools.

For a long time, vector graphics have been limited to primitives (paths, polygons) filled with uniform color or linear and radial gradients. Although skillful artists can create rich vector art with these simple tools, the resulting images often present flat or limited shading due to the limitations in terms of complex gradients and blur. Commercial tools such as Adobe Live Trace<sup>®</sup> assist the user in creating complex vector graphics from input bitmap images. They operate by segmenting an input image into regions of constant or

slowly varying color, and fitting polygons onto these primitives. Although this class of methods produces convincing results in uniform areas, the segmentation typically generates a prohibitive number of primitives in smooth regions.

The ArDeco system of Lecot et al. [2006] allows vectorization of more complex gradients using existing linear or radial gradient primitives. It is based on a segmentation of the input image into regions of slowly varying color, and an approximation of color variations within each region with linear or quadratic gradients. The resulting primitives are fully compatible with the SVG standard, but the approximation tends to produce sharp color transitions between segmented regions. A simpler solution to bypass these limitations, adopted by the SVG format and modern tools (Adobe Illustrator<sup>®</sup>, Corel CorelDraw<sup>®</sup>, Inkscape<sup>®</sup>), is to reblur the image once vector primitives have been rasterized. However, they only allow for a uniform blur for each given primitive, which, similar to the limitations of flat colors or simple gradients, necessitates an unpractically large number of primitives to approximate complex image content.

Gradient meshes have been recently introduced (Adobe Illustrator<sup>®</sup>, Corel CorelDraw<sup>®</sup>) to address all of these issues by allowing a user to specify color values on the vertices of a quad mesh and smoothly interpolating these values over the mesh faces. However, creating a mesh from scratch requires much skill and patience, because the artist needs to accurately anticipate the mesh resolution and topology necessary to embed the desired smooth features. This is why most users rely on an example bitmap to drive the design of realistic gradient meshes. The users first decompose an input photograph into several sub-objects and then draw meshes over each sub-object following their topology; finally, they sample colors in the original photograph, assigning them to the mesh vertices. Many tutorials describing this approach are available on the Web. Still, drawing effective meshes and performing accurate manual color sampling is very time consuming in practice (several hours or even days for detailed images) and requires a good appreciation of the image complexity to adopt an adequate mesh resolution. Recently, the paper of Sun et al [2007] proposed to assist the user by automatically fitting an input gradient mesh to an input image. The fitting is achieved by minimizing the reconstruction error between the resulting image and an input photograph. Their semi-automatic method greatly reduces the time required to draw a precise mesh and sampling colors, although the user still has to manually specify the sub-objects of the picture and draw the initial meshes with an adequate resolution. Price and Barret [2006] proposed a similar approach for object vectorization, using recursive subdivisions until the reconstruction error falls below a fixed threshold. Their method produces faithful results but also generates many small patches in smooth regions.

Yet, with both approaches, it remains unclear how to efficiently manipulate the resulting meshes for further editing. We believe this is due to the unnecessary constraints imposed by the use of a mesh: using a predefined topology, employing specific patch subdivision schemes, and choosing a global orientation. In practice, this translates into a dense net of patches that are not readily connected to the depicted content. Hence, the manipulation of such a set of primitives quickly becomes prohibitive for the non-expert.

The new representation described in this paper offers the same level of visual complexity as that reached by gradient meshes, but has two main advantages: it is *sparse*, and corresponds to *meaningful* image features. Indeed, the newly introduced diffusion curves are intuitive to create, as each primitive corresponds to an image feature; they are easy to manipulate and animate, as no constraint is imposed on connectivity, and no superfluous subdivision is required; and they are well adapted for stylization, which would be non-trivial with a gradient mesh approach. Moreover, compared to

<sup>2</sup>[www.carto.net/papers/svg/comparison\\_flash\\_svg/](http://www.carto.net/papers/svg/comparison_flash_svg/)

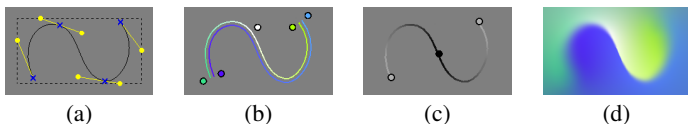
other methods reviewed above, our representation naturally lends itself to automatic extraction from a bitmap image: primitive locations are found completely automatically, and primitive attributes (color and blur) are extracted via vision algorithms.

In other words, compared to regions used in classic vector representations, or patches used in gradient meshes, our approach is motivated by the fact that most of the color variation in an image is caused by, or can be modeled with edges; and that (possibly open) regions or patches are implicitly defined in between. Such a sparse image representation is strongly motivated by the work of Elder [1999], who demonstrated that edges are a near-complete representation for images. Elder [2001] also suggested the possibility of using edges to efficiently manipulate images with basic operations (edge delete, copy and paste). However, we believe the full potential of this approach has as yet not been attained. For this reason, our conversion algorithm starts from the same premises as Elder’s system. But by vectorizing edges and their attributes, we extend its manipulation capabilities to include shape, color, contrast, and blur operations. This way, we provide the user with more intuitive editing tools, and also support resolution-independence, stylization and key-frame animation.

### 3 Diffusion Curves

In this section we introduce the basic primitive of our representation, called a *diffusion curve*, and describe how to efficiently render an image from such primitives. Specification and manipulation of diffusion curves are discussed in subsequent sections.

#### 3.1 Data structure



**Figure 2:** A Diffusion curve is composed of (a) a geometric curve described by a Bézier spline, (b) arbitrary colors on either side, linearly interpolated along the curve, (c) a blur amount linearly interpolated along the curve. The final image (d) is obtained by diffusion and reblurring. Note the complex color distribution and blur variation defined with a handful of controls.

The basic element of a diffusion curve is a geometric curve defined as a cubic Bézier spline (Figure 2(a)) specified by a set of control points  $P$ . The geometry is augmented with additional attributes: two sets of color control points  $C_l$  and  $C_r$  (Figure 2(b)), corresponding to color constraints on the *right* and *left* half space of the curve; and a set of *blur* control points ( $\Sigma$ ) that defines the smoothness of the transition between the two halves (Figure 2(c)). Intuitively, the curves diffuse color on each side with a soft transition across the curve given by its blur (Figure 2(d)).

Color and blur attributes can vary along a curve to create rich color transitions. This variation is guided by an interpolation between the attribute control points in attribute space. In practice, we use linear interpolation and consider colors in RGB space throughout the rendering process, because they map more easily onto an efficient GPU implementation and proved to be sufficient for the artists using our system. Controls points for geometry and attributes are stored independently, since they are generally uncorrelated. This leads to four independent arrays in which the control points (geometry and attribute values) are stored together with their respective parametric position  $t$  along the curve:

**DiffusionCurve:**  $\left\{ \begin{array}{l} P[n_{pos}]: \text{array of } (x, y, \text{tangent}); \\ C_l[n_l]: \text{array of } (r, g, b, t); \\ C_r[n_r]: \text{array of } (r, g, b, t); \\ \Sigma[n_\sigma]: \text{array of } (\sigma, t); \end{array} \right.$

The diffusion curves structure encodes data similar to Elder’s edge-based representation [1999]. However, the vectorial nature of a diffusion curve expands the capabilities of Elder’s discrete edges by allowing precise control over both shapes — via manipulation of control points and tangents — and appearance attributes — via color and blur control points (small circles on the Figures). This fine-level control, along with our realtime rendering procedure, facilitates the drawing and editing of smooth-shaded images.

#### 3.2 Rendering smooth gradients from diffusion curves

Three main steps are involved in our rendering model (see Figure 3): (1) rasterize a *color sources* image, where color constraints are represented as colored curves on both sides of each Bézier spline, and the rest of the pixels are uncolored; (2) *diffuse* the color sources similarly to heat diffusion — an iterative process that spreads the colors over the image; we implement the diffusion on the GPU to maintain realtime performance; and (3) *reblur* the resulting image with a spatially varying blur guided by the blur attributes. Technical details about these three steps are explained in the following sections.

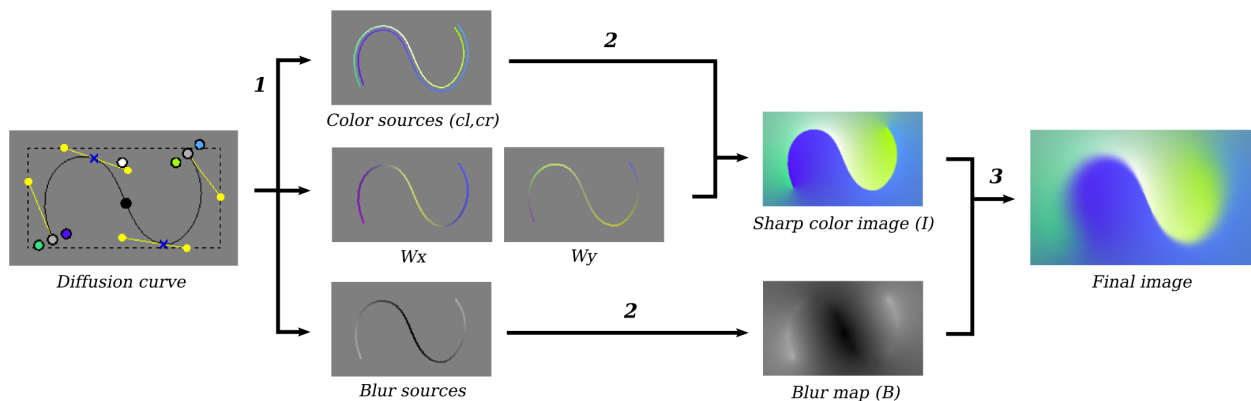
##### 3.2.1 Color sources

Using the interpolated color values, the first step renders the left and right color sources  $cl(t)$ ,  $cr(t)$  for every pixel along the curves. An alpha mask is computed along with the rendering to indicate the exact location of color sources versus undefined areas.

For perfectly sharp curves, these color sources are theoretically infinitely close. However, rasterizing pixels separated by too small a distance on a discrete pixel grid leads to overlapping pixels. In our case, this means that several color sources are drawn at the same location, creating visual artifacts after the diffusion. Our solution is to distance the color sources from the curve slightly, and to add a color gradient constraint directly on the curve. The gradient maintains the sharp color transition, while the colors, placed at a small distance  $d$  in the direction normal to the curve, remain separate.

More precisely, the gradient constraint is expressed as a gradient field  $\mathbf{w}$  which is zero everywhere except on the curve, where it is equal to the color derivative across the curve. We decompose the gradient field in a gradient along the  $x$  direction  $\mathbf{w}_x$  and a gradient along the  $y$  direction  $\mathbf{w}_y$ . For each pixel on the curve, we compute the color derivative across the curve from the curve normal  $N$  and the left ( $cl$ ) and right ( $cr$ ) colors as follow (we omit the  $t$  parameter for clarity):  $w_{x,y} = (cl - cr)N_{x,y}$

We rasterize the color and gradient constraints in 3 RGB images: an image  $C$  containing colored pixels on each side of the curves, and two images  $W_x, W_y$  containing the gradient field components. In practice, the gradient field is rasterized along the curves using lines of one pixel width. Color sources are rasterized using triangle strips of width  $2d$  with a special pixel shader that only draws pixels that are at the correct distance  $d$  (Figure 3(1)). In our implementation  $d$  is set at 3 pixels. Pixel overlap can still occur along a curve in regions of high curvature (where the triangle strip overlaps itself) or when two curves are too close to each other (as with thin structures or intersections). A simple stencil test allows us to discard overlapping color sources before they are drawn, which implies that solely the gradient field  $\mathbf{w}$  dictates the color transitions in these areas. An example of such case can be seen in Figure 1, where the eyebrows are accurately rendered despite their thin geometry.



**Figure 3:** Rendering diffusion curves requires (1) the rasterization of the color and blur sources, along with the gradient field  $\mathbf{w} = (\mathbf{w}_x, \mathbf{w}_y)$ , (2) the diffusion of colors and blur, and (3) the reblurring of the color image.

### 3.2.2 Diffusion

Given the color sources and gradient fields computed in the previous step, we next compute the color image  $I$  resulting from the steady state diffusion of the color sources subject to the gradient constraints (Figure 3(2)). Similarly to previous methods [Carlsson 1988; Elder 1999; Pérez et al. 2003], we express this diffusion as the solution to a Poisson equation, where the color sources impose local constraints:

$$\begin{aligned} \Delta I &= \text{div } \mathbf{w} \\ I(x, y) &= C(x, y) \text{ if pixel } (x, y) \text{ stores a color value} \end{aligned}$$

where  $\Delta$  and  $\text{div}$  are the Laplace and divergence operators.

Computing the Poisson solution requires solving a large, sparse, linear system, which can be very time consuming if implemented naively. To offer interactive feedback to the artist, we solve the equation with a GPU implementation of the multigrid algorithm [Briggs et al. 2000; Goodnight et al. 2003]. The paper of McCann and Pollard [2008], published in the same proceedings, gives a detailed description of a simple implementation very similar to ours. The idea behind multigrid methods is to use a coarse version of the domain to efficiently solve for the low frequency components of the solution, and a fine version of the domain to refine the high frequency components. We use Jacobi relaxations to solve for each level of the multigrid, and limit the number of relaxation iterations to achieve realtime performances. Typically, for a  $512 \times 512$  image we use  $5i$  Jacobi iterations per multigrid level, with  $i$  the level number from fine to coarse. This number of iterations can then be increased when high quality is required. All the images in this paper and in the accompanying video have been rendered using an Nvidia GeForce 8800, providing realtime performance on a  $512 \times 512$  grid with a reasonable number of curves (several thousands).

### 3.2.3 Reblurring

The last step of our rendering pipeline takes as input the color image containing sharp edges, produced by the color diffusion, and reblurs it according to blur values stored along each curve. However, because the blur values are only defined along curves, we lack blur values for off-curve pixels. A simple solution, proposed by Elder[1999], diffuses the blur values over the image similarly to the color diffusion described previously. We adopt the same strategy and use our multigrid implementation to create a blur map  $B$  from the blur values. The only difference to the color diffusion process is that blur values are located exactly on the curve so we do not require any gradient constraints. This leads to the following equation:

$$\begin{aligned} \Delta B &= 0 \\ B(x, y) &= \sigma(x, y) \text{ if pixel } (x, y) \text{ is on a curve} \end{aligned}$$

Giving the resulting blur map  $B$ , we apply a spatially varying blur on the color image (Figure 3(3)), where the size of the blur kernel at each pixel is defined by the required amount of blur for this pixel. Despite a spatially varying blur routine implemented on the GPU [Bertalmio et al. 2004], this step is still computationally expensive for large blur kernels (around one second per frame in our implementation), so we bypass it during curve drawing and manipulations and reactivate it once the drawing interaction is complete.

### 3.2.4 Panning and zooming

Solving a Poisson equation leads to a global solution, which means that any color value can influence any pixel of the final image. Even though the local constraints introduced by the color sources reduce such global impact, this raises an issue when zooming into a sub-part of an image, because curves outside the current viewport should still influence the viewport's content. To address this problem without requiring a full Poisson solution at a higher resolution, we first compute a low-resolution diffusion on the unzoomed image domain, and use the obtained solution to define Dirichlet boundary conditions around the zooming window. This gives us a sufficiently good approximation to compute a full-resolution diffusion only within the viewport.

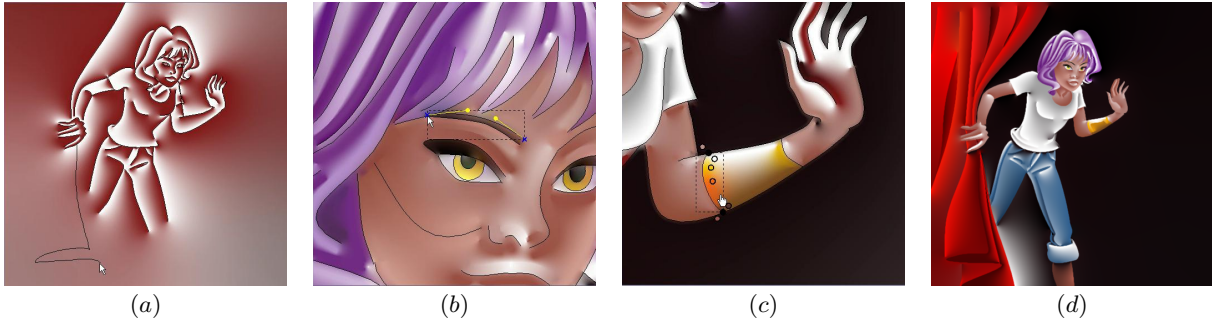
## 4 Creating diffusion curves

The process of creating an image varies among artists. One may start from scratch and give free rein to his imagination while another may prefer to use an existing image as a reference. We provide the user with both options to create diffusion curves. For *manual* creation, the artist can create an image with our tool by sketching the lines of the drawing and then filling in the color. When using an image as a template, we propose two methods. *Assisted*: The artist can trace manually over parts of an image and we recover the colors of the underlying content. *Automatic*: the artist can automatically convert an image into our representation and possibly post-edit it.

### 4.1 Manual creation

To facilitate content creation for the artist, we offer several standard tools: editing of curve geometry, curve splitting, copy/paste, zooming, color picking, etc. We also developed specific tools: copy/paste of color and blur attributes from one curve to another, editing of attributes control points (add, delete, and modify), etc. The tutorial provided on our project page<sup>3</sup> and conference DVD gives a more complete description of our current prototype interface.

<sup>3</sup><http://artis.imag.fr/Publications/2008/OBWBS08/>

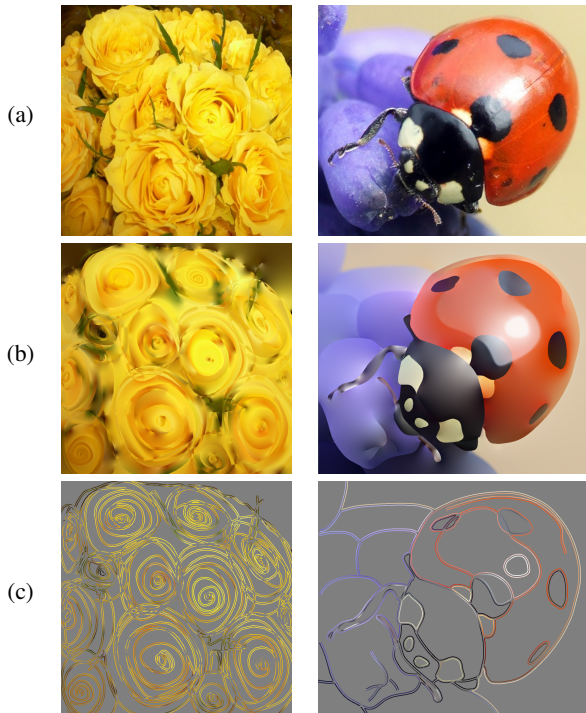


**Figure 4:** Example steps for manual creation: (a) sketching the curves, (b) adjusting the curve’s position, (c) setting colors and blur along the diffusion curve and (d) the final result. The image was created by an artist at first contact with the tool and it took 4 hours to create.

To illustrate how an artist can use our diffusion curves, we show in Figure 4 (and accompanying video) the different stages of an image being drawn with our tool. The artist employs the same intuitive process as in traditional drawing: a sketch followed by color filling.

## 4.2 Tracing an image

In many situations an artist will not create an artwork entirely from scratch, but instead use existing images for guidance. For this, we offer the possibility of extracting the colors of an underlying bitmap along a drawn curve. This process is illustrated in Figure 5.



**Figure 5:** Tracing with diffusion curves: (a) Original bitmaps; (b) left: Result of a stylistic tracing using color sampling (artist drawing time: less than a minute); right: Result of a tracing using active contours and color sampling (artist drawing time: 90 minutes). (c) The corresponding diffusion curves (color sources have been thickened for illustration purpose).

The challenge here is to correctly extract and vectorize colors on each side of a curve. We also need to consider that color outliers might occur due to noise in the underlying bitmap or because the curve positioning was suboptimal. We first uniformly sample the

colors along the curve at a distance  $d$  (same as the one used for rendering) in the direction of the curve’s normal. We then identify color outliers by measuring a standard deviation in a neighborhood of the current sample along the curve. To this end, we work in CIE  $L^*a^*b^*$  color space (considered perceptually uniform for just-noticeable-differences), and tag a color as an outlier if it deviates too much from the mean in either the  $L^*$ ,  $a^*$  or  $b^*$  channel. We then convert back colors to RGB at the end of the vectorization process for compatibility with our rendering system.

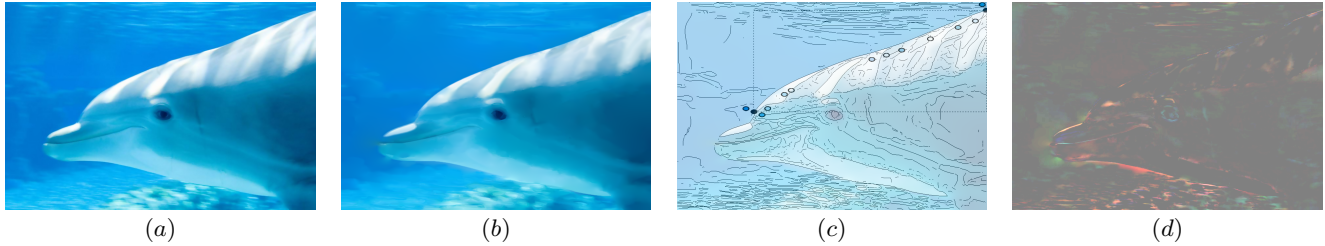
To obtain a linear color interpolation similar to that used for rendering, we fit a polyline to the color points using the Douglas-Peucker algorithm [Douglas and Peucker 1973]. The iterative procedure starts with a line connecting the first and last point and repeatedly subdivides the line into smaller and smaller segments until the maximum distance (still in CIE  $L^*a^*b^*$ ) between the actual values and the current polyline is smaller than the error tolerance  $\epsilon$ . The end points of the final polyline yield the color control points that we attach to the curve. A creative example that uses color sampling is illustrated in Figure 5(b)-left image, where an artist has drawn very stylistic shapes, while using the color sampling feature to reproduce the global tone of the original image, similarly to an in-painting process [Bertalmio et al. 2000].

When tracing over a template, one would normally want to position the curves over color discontinuities in the underlying image. Since it is not always easy to draw curves precisely at edge locations in a given image, we provide some help by offering a tool based on *Active Contours* [Kass et al. 1987]. An active contour is attracted to the highest gradient values of the input bitmap and allows the artist to iteratively snap the curve to the closest edge. The contour can also be easily corrected when it falls into local minima, or when a less optimal but more stylistic curve is desired. Figure 5(b)-right shows the image of a lady bug created using geometric snapping and color extraction. While the artist opted for a much more stylized and smoothed look compared to the original, the image still conveys diffuse and glossy effects, defocus blur, and translucency.

## 4.3 Automatic extraction from a bitmap

Finally, an artist might want to add stylization and expression to an already existing image. We propose a method for automatically extracting and vectorizing diffusion curves data from a bitmap.

**Data Extraction:** Many approaches exist to find edges and determine their blur and color attributes. We based our data extraction on our previous work [Orzan et al. 2007], as that work was “designed” for edge-based image stylization. Diffusion curves extend our previous approach from a raster-based method to a fully vectorized version. For brevity, we will only recall the basic steps of the previous method, and refer the interested reader to Orzan et al. [2007].



**Figure 6:** Example of our reconstruction: (a) original image; (b) result after conversion into our representation; (c) automatically extracted diffusion curves; (d) RGB difference between original and reconstructed image (amplified by 4); note that the most visible error occurs along edges, most probably because, through vectorization, we change their localization.

The basic approach relies on the Gaussian scale space, which can be pictured as a stack of increasingly blurred versions of an image, where higher scale images exhibit less and less detail. To obtain edge positions and blur estimates from this scale space, we first extract edges at all available scales using a classical Canny detector [Canny 1986]. Contrary to our previous approach, which works only on the luminance channel, we now apply the Canny detector to the multi-channel color gradient described in Di Zenzo [1986]. This allows us to detect sharp color variations in iso-luminant regions of the image, where a luminance gradient would fail. Then, taking inspiration from previous work in scale-space analysis [Lindeberg 1996; Elder 1999], we find the scale at which an edge is best represented (the more blurred the edge, the higher the scale). We use this ideal scale to identify the degree of blur of an edge, as previously, but also use it to localize edges. It should be noted that very blurry edges are difficult to detect and parameterize accurately. In our system we find that very large gradients are sometimes approximated with a number of smaller ones.

After the scale-space analysis, we are left with an have chosen to edge map, which contains the edge locations and the blur values for the edge pixels. As explained in our previous paper [Orzan et al. 2007], we also provide a method to extract an edge’s *lifetime* (a measure of its spread throughout consecutive scales), which allows for separation of detail and contour edges. While in the past we directly used the extracted data, our new approach requires an additional processing step: colors on either side of the edges must be extracted explicitly. To this end, we connect pixel-chains from the edge map and proceed to sample colors in the original image on each side of the edge in the direction of the edge normal. In practice, the gradient normal to the edge is difficult to estimate for blurry edges, so we use the direction given by the normal of a polyline fitted to each edge. For an estimated blur  $\sigma$ , we pick the colors at a distance  $3 \cdot \sigma$  from the edge location, which covers 99% of the edge’s contrast, assuming a Gaussian-shaped blur kernel [Elder 1999]. While the  $3 \cdot \sigma$  distance ensures a good color extraction for the general case, it poses numerical problems for structures thinner than 3 pixels ( $\sigma < 1$ ); in this particular case, color cannot be measured accurately.

**Conversion to diffusion curves:** For vectorization of positions, we take inspiration from the approach used in the open source Potrace<sup>®</sup> software [Selinger 2003]. The method first approximates a pixel chain with a polyline that has a minimal number of segments and the least approximation error, and then transforms the polyline into a smooth poly curve made from end-to-end connected Bézier curves. The conversion from polylines to curves is performed with classical least square Bézier fitting based on a maximum user-specified fitting error and degree of smoothness. For attribute vectorization, we use the same method as in Section 4.2.

Several parameters determine the complexity and quality of our vectorized image representation. For the edge geometry, the Canny

threshold determines how many of the image edges are to be considered for vectorization; a despeckling parameter sets the minimum length of a pixel chain to be considered for vectorization; and finally, two more parameters set the smoothness of the curve fitting and the fitting error. For the blur and color values, two parameters are considered: the size of the neighborhood for eliminating outliers, and the maximum error accepted when fitting the polyline. For most images in this paper, we use a Canny high threshold of 0.82 and low threshold of 0.328, we discard pixel chains with less than 5 pixels, we use a smoothness parameter of 1 (Potrace default) and we set the fitting error to 0, so the curve closely approximates the original edges. For attributes, we consider a neighborhood of 9 samples, and the maximum error accepted is 2 blur scales for the blur and 30 CIE L\*a\*b\* units for colors.

## 5 Results

Diffusion curves, as vector-based primitives, benefit from the advantages of traditional vector graphics: zooming-in preserves sharp transitions (Figure 8 (e)) and keyframing is easily performed via linear interpolation of geometry and attributes (Figure 7). Our representation is equally well suited for global and local image stylization. Curve shapes and attributes can be easily modified to obtain effects such as that presented in Figure8(d). For diffusion curves extracted from an image, the lifetime measure provided by Orzan et al. [2007] can be used to adjust preservation of detail (Figure8(c)).



**Figure 7:** Keyframing with diffusion curves: Three keyframes of an animation.

To validate our approach and to collect valuable practical feedback, we had various artists use our prototype. Most figures in this paper were generated in these sessions. All artists were well versed in digital content creation tools, but had no technical background. They were given a brief tutorial (see our project page and the conference DVD), amounting to approximately 10 minutes of instructions. The artists were able to create many varied and intricate examples from the very first session and found the manipulation of diffusion curves intuitive after a short accommodation phase. Manual image creation took anywhere from several minutes (Figure 5(b)) to a few hours (Figure 4). However, the artists agreed that a more powerful user interface would greatly speed up the creation process.



**Figure 8:** Stylization effects: (a) original bitmap; (b) Automatic reconstruction; (c) Reconstruction simplified by removing edges with low lifetime; (d) Global shape stylization applied to (c); (e) Enlargement of (b).

## 6 Discussion & Future work

In the previous sections, we presented our new vector-based primitive, and explained the various options at an artist’s disposal to create smooth-shaded images thanks to this intuitive representation. We now compare our approach with the most commonly used vector tool for creating images with similarly complex gradients: Gradient Meshes. Next, we identify the remaining challenges that we would like to address in future work.

### 6.1 Comparison with Gradient Meshes

**Representational efficiency:** In terms of sparsity of encoding, both gradient meshes and diffusion curves are very efficient image representations. A direct comparison between both representations is difficult, as much depends on the chosen image content (for example, gradient meshes require heavy subdivision to depict sharp edges and it can be difficult to conform the mesh topology to complex geometric shapes). Furthermore, Price and Barret [2006] presented a more compact sub-division gradient mesh, yet all available tools employ a regular mesh. While the diffusion curves representation appears more compact at first glance (see Figure 11), it should be noted that each geometric curve can hold an arbitrary amount of color and blur control points (see Table 9). So, while the sparsity of encoding of both representations can be considered comparable, we would argue the flexibility of diffusion curves to be a significant benefit, as it allows us any degree of control on a curve, without a topologically-imposed upper or lower bound on the number of control points.

	Curves	$P$	$Cl$	$Cr$	$\Sigma$
Roses (fig. 5 left)	20	851	581	579	40
Lady bug (fig. 5 right)	71	521	293	291	144
Curtain (fig. 4)	131	884	318	304	264
Dolphin (fig. 6)	1521	6858	3254	3271	3433

**Figure 9:** Number of curves, geometric control points ( $P$ ), left and right color control points ( $Cl$ , respectively  $Cr$ ) and blur control points ( $\Sigma$ ) for the images of this paper.

**Usability:** We believe that diffusion curves are a more natural drawing tool than gradient meshes. As mentioned previously, artists commonly use strokes to delineate boundaries in an image. Diffusion curves also allow an artist to evolve an artwork gradually and naturally. Gradient meshes, on the other hand, require careful planning and a good understanding of the final composition of the intended art piece. Most gradient mesh images are a complex combination of several individual — rectangular or radial — gradient meshes, often overlapping. All these decisions have to be made before the relevant image content can be created and visualized.

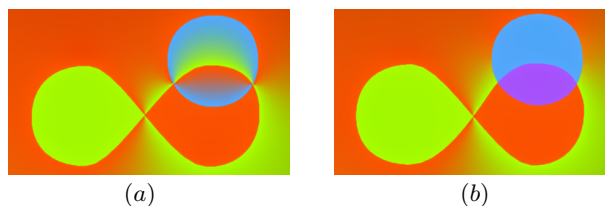
**Topology:** In some situations, the topology constraints of gradient meshes can be rather useful, for example when moving a gradient

mesh to a different part of an image, or when warping the entire mesh. Such manipulations are also possible in our representation, but not as straightforward. For moving part of an image, the relevant edges have to be selected and moved as a unit. More importantly, without support for layering and transparency (see Section 6.2) it is difficult to ascertain how the colors of outer edges should interact with their new surroundings. A mesh warp could be implemented as a space warp around a group of edges.

### 6.2 Future challenges

Currently, our representation is single layered, but we are aware that multiple, independent layers offer more flexibility to the artist. To fully take advantage of a layered system, we need to address the interaction of multiple layers (considering a global Poisson solution), and the additional computational demands. Blending of layers would also require a notion of (gradual) transparency. Our current representation is more related to planar-maps [Asente et al. 2007] that model vector graphics in a single layer.

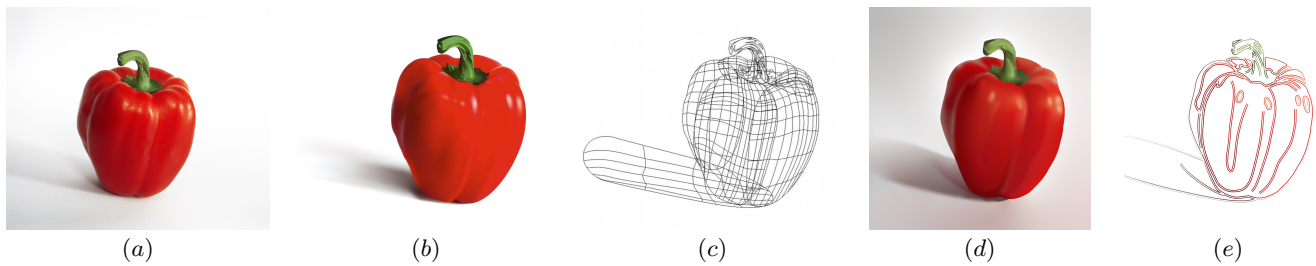
A different point worth improving is the way diffusion curves deal with intersections. Currently, diffusion curves present a specific (although predictable and meaningful) behavior: the colors attached to the two intersecting curves essentially compete with each other, which creates a smooth color gradient after diffusion (Figure 10(a)). If this default behavior is undesirable, the user can correct it by either adding color controls on each side of the intersection, or by splitting the curves in several parts with different colors (Figure 10(b)). Automating such behaviors would represent a powerful tool for easing user interactions.



**Figure 10:** The default behavior of diffusion curves at intersections (a) can be corrected by curve splitting and color editing (b).

Another limitation, common to all vector graphics, occurs in images or image regions that contain many small color or luminance variations, such as textures. In practice, most of the visual information of highly textured regions is captured by the automatic conversion, but imprecision occur when the texture is composed of many small structures (small compared to the distance  $d$  defined in Section 3.2.1). Moreover, the large amount of curves required to represent textures makes a vector representation inefficient and difficult to manipulate. Incorporating a diffusion curves version of texture synthesis tools is an interesting area of future research.





**Figure 11:** Gradient Mesh comparison: (a) Original photograph; (b,c) Manually created gradient mesh (© Brooke Nuñez Fetissoff <http://lifeinvector.com/>), with 340 vertices (and as many color control points); (d,e) Our drawing created by manually tracing over the image; there are 38 diffusion curves, with 365 geometric, 176 left-color, and 156 right-color control points.

Finally, as diffusion curves rely partially on a Poisson solution, nothing prevents their use in a variety of common Poisson-editing applications (e.g. for the transfer of image features such as shadows, or for merging multiple images in a same representation). We plan to extend our prototype in this direction in the near future.

## 7 Conclusion

We have introduced diffusion curves as a new image representation, offering most of the benefits usually found in vector approaches, such as resolution independence, exact editability, and compactness; while at the same time allowing to depict highly complex image content, generally only realizable with raster graphics. Diffusion curve images are comparable both in quality and coding efficiency with gradient meshes, but are simpler to create (according to several artists who have used both tools), and can be captured from bitmaps fully automatically.

## Acknowledgments

Several people have been instrumental in giving this paper its final form. We would like to especially thank Laurence Boissieux and Philippe Chabbaroux for the time they spent testing our prototype and for the artworks they created with it. Particular thanks also to Amy Durocher for the design of the icons in our interface, and to Yann Boulanger for his "sauté de dinde". Finally we would like to thank the anonymous reviewers, and all the people from Adobe and INRIA for their constructive comments and feedbacks. Alexandrina Orzan is supported by a grant from the European Community under the Marie-Curie project MEST-CT-2004-008270.

## References

ASENTE, P., SCHUSTER, M., AND PETTIT, T. 2007. Dynamic planar map illustration. *ACM TOG (Proc. of SIGGRAPH)* 26, 3, 30.

BERTALMIO, M., SAPIRO, G., CASELLES, V., AND BALLESTER, C. 2000. Image inpainting. In *Proc. of ACM SIGGRAPH 2000*, 417–424.

BERTALMIO, M., FORT, P., AND SANCHEZ-CRESPO, D. 2004. Real-time, accurate depth of field using anisotropic diffusion and programmable graphics cards. In *Proc. of 3DPVT*, 767–773.

BRIGGS, W. L., HENSON, V. E., AND MCCORMICK, S. F. 2000. *A multigrid tutorial (2nd ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

CANNY, J. 1986. A computational approach to edge detection. *IEEE PAMI* 8, 6, 679–698.

CARLSSON, S. 1988. Sketch based coding of grey level images. *Signal Processing* 15, 1, 57–83.

DOUGLAS, D., AND PEUCKER, T. 1973. Algorithms for the reduction of the number of points required for represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization* 10, 2, 112–122.

ELDER, J. H., AND GOLDBERG, R. M. 2001. Image editing in the contour domain. *IEEE PAMI* 23, 3, 291–296.

ELDER, J. H. 1999. Are edges incomplete? *International Journal of Computer Vision* 34, 2-3, 97–122.

GOODNIGHT, N., WOOLLEY, C., LEWIN, G., LUEBKE, D., AND HUMPHREYS, G. 2003. A multigrid solver for boundary value problems using programmable graphics hardware. In *Graphics Hardware*, 102–111.

KASS, M., WITKIN, A., AND TERZOPOULOS, D. 1987. Snakes: Active contour models. *International Journal of Computer Vision* 1, 4, 321–331.

KOENDERINK, J. J., AND DOORN, A. J. 1979. The internal representation of solid shape with respect to vision. *Biological Cybernetics* 32, 4, 211–216.

LECOT, G., AND LEVY, B. 2006. Ardeco: Automatic Region Detection and COntversion. In *Proc. of EGSR*, 349–360.

LINDBERG, T. 1996. Edge detection and ridge detection with automatic scale selection. In *Proc. of CVPR*, 465–470.

MARR, D., AND HILDRETH, E. C. 1980. Theory of edge detection. *Proc. of the Royal Society of London. Biological Sciences* 207, 187–217.

MCCANN, J., AND POLLARD, N. S. 2008. Real-time gradient-domain painting. *ACM TOG (Proc. of SIGGRAPH)* 27, 3.

ORZAN, A., BOUSSEAU, A., BARLA, P., AND THOLLOT, J. 2007. Structure-preserving manipulation of photographs. In *NPAP*, 103–110.

PÉREZ, P., GANGNET, M., AND BLAKE, A. 2003. Poisson image editing. *ACM TOG (Proc. of SIGGRAPH)* 22, 3, 313–318.

PRICE, B., AND BARRETT, W. 2006. Object-based vectorization for interactive image editing. *Visual Computer (Proc. of Pacific Graphics)* 22, 9, 661–670.

SELINGER, P. 2003. *Potrace: a polygon-based tracing algorithm*.

SUN, J., LIANG, L., WEN, F., AND SHUM, H.-Y. 2007. Image vectorization using optimized gradient meshes. *ACM TOG (Proc. of SIGGRAPH)* 26, 3, 11.

SUTHERLAND, I. E. 1980. *Sketchpad: A man-machine graphical communication system (Outstanding dissertations in the computer sciences)*. Garland Publishing, Inc., New York, NY, USA.

ZENZO, S. D. 1986. A note on the gradient of a multi-image. *Computer Vision, Graphics, and Image Processing* 33, 1, 116–125.