Generalized Diffusion Curves: An Improved Vector Representation for Smooth-Shaded Images

Stefan Jeschke†

IST Austria

Figure 1: An input raster image (a) with a set of Bezier curves (b) is converted to vector graphics. (c) shows a Diffusion Curves Image (DCI) and (d) shows a generalized version of this model (GDCI) presented in this paper. The better three-dimensional appearance of the GDCI model can be noticed in the lampshade and in the wooden pieces. The GDCI in (d) can be edited just like a DCI, as is shown here in (e).

Abstract

This paper generalizes the well-known Diffusion Curves Images (DCI), which are composed of a set of Bezier curves with colors specified on either side. These colors are diffused as Laplace functions over the image domain, which results in smooth color gradients interrupted by the Bezier curves. Our new formulation allows for more color control away from the boundary, providing a similar expressive power as recent Bilaplace image models without introducing associated issues and computational costs. The new model is based on a special Laplace function blending and a new edge blur formulation. We demonstrate that given some user-defined boundary curves over an input raster image, fitting colors and edge blur from the image to the new model and subsequent editing and animation is equally convenient as with DCIs. Numerous examples and comparisons to DCIs are presented.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation

1. Introduction

Vector graphics research has seen significant progress in recent years, with numerous new powerful representations. Among them are mesh-based representations like the Ardeco system [LL06], gradient meshes [SLWS07], curvilinear patches [XLY09] and subdivision surfaces [LHYF12]. Such meshes have been automatically fitted to natural, detailed images with high image quality, but the dense mesh topology restricts later editing. Another line of research started from the well-known Diffusion Curves Images (DCI) [OBW*08]. Here, images are modeled as partial differential equations (PDE) with some sparse boundary constraints. In the case of DCIs these are Bezier curves with colors attached on either side, that get diffused over the domain by a Laplace equation (Section 1.1 will provide more details). Figure 1(c) shows an example. DCIs generated high interest because of their very compact and intuitive nature with the absense of dense topological constraints: designers can directly operate on image boundaries. Sig-
Bilaplace diffusion curves

Generalized diffusion curves

Figure 2: The Bilaplace function shown on the left interpolates from colors and cross-boundary gradient values, similar to a Hermite interpolation. Consequently, colors scale with the boundary geometry, resulting here in undesirably oversaturated red color in the center. The proposed generalized diffusion curves are shown on the right. Here one can observe that colors do not scale with curve geometry, thus avoiding the undesirable behaviour. At the same time, it provides a similar control as the Bilaplace model by blending two Laplace functions.

significant work was presented to improve DCI rendering stability and speed [JCW09a], and to provide more control over the diffusion process [BEDT10]. DCIs also have other applications, among them representing 3D surface details [JCW09b] and modeling inner structures of 3D objects [TSN10].

Recent research aimed at providing more control for how color changes away from the boundary, which is important because humans can interpret shape by lighting variation over a surface very well: even slightly bent surfaces reflect light differently than planar ones. Figure 1 illustrates this point with a comparison between a DCI and our new representation. To make this possible, several existing approaches extended DCIs by employing a Bilaplace formulation as governing PDE [FSH11, BBG12, IKCM13]. Unfortunately, while Bilaplace formulations can provide a lot of control for designers [FSH11, BBG12], there are significant problems associated with them. The first is demonstrated in the left part of Figure 2 which was created using the tool of Finch et al. [FSH11]. A circular shape with a blue color defined outside and red color inside is enlarged. Comparing the left and the right image reveals that the color range changes with the boundary geometry. This behaviour is conceptually similar to Hermite curve interpolation as sketched at the top of Figure 2, and it is clearly not desirable in 2D image editing and animation applications. We note that Jacobson et al. [JWS12] proposed a method to bound Bilaplace functions such that the oversaturation in Figure 2 would be prevented, but the color range would still drastically change with boundary geometry. Another issue is that Bilaplace functions are known to be much more computationally expensive and subject to numerical instability because the system is less well-conditioned, as opposed to Laplace functions. Apart from this, modeling a blurred image edge requires two parallel curves as Bilaplace constraints (called “compound curves” in Finch et al. [FSH11]), which adds to representation complexity and editing effort compared to Ozran et al. [OBW08], who work with single curves and image blur. Existing Bilaplace models [FSH11, BBG12] generally provide extensive sets of constraints giving lots of freedom for designers. But this makes fitting a given color image to a set of Bezier curves challenging because selecting the right constraints is not trivial. Recently, Xie et al. [XSTN14] showed how to automatically convert a raster image to a set of Bezier curves together with a Bilaplace image model. However, their representation is based on the boundary element method so that colors and Bezier curve geometry remain mostly static, i.e., both cannot be edited later.

The image model proposed in this paper naturally generalizes DCIs while overcoming the limitations mentioned above; we call it Generalized Diffusion Curve Images (GDCIs). One goal is to provide a similar control over image color as the Bilaplace model. In the latter, Dirichlet boundary conditions define image color directly in the curve vicinity and Neumann boundary conditions (color gradients) define how color changes away from the curve, as is depicted in Figure 2 (left). Conceptually, we achieve this “two-stop” color interpolation in the direction perpendicular to the curve by a special blending of two separate Laplace functions (Figure 2, right) whereas DCIs only offer control for a single color directly at the curve. We will show that fitting a given raster image to a GDCI and then editing and animating it afterwards is equally convenient as for DCIs, but with higher representation quality. Figure 1 shows an example of a GDCI and a comparison to the DCI model. Figure 9 sketches the editing workflow and Figures 1 and 10 provide editing examples. To the best of our knowledge, no existing work provides a similar flexibility. In addition, the interpolation of Laplace functions avoids the color range scaling problem demonstrated in Figure 2, and Laplace functions used in a DCI can directly be modeled within the new framework. A further extension is a new edge blur formulation without the need for compound curves. Finally, Laplace functions yield fast and numerically stable computations, as will be demonstrated in Section 4.

1.1. Background and Notation

For a self-contained presentation, we shortly introduce important concepts of a DCI (referring the interested reader to [OBW08]), and introduce the notation used in this paper. As noted above, a DCI is defined as a set of Bezier curves with colors on either side. These colors are specified by linear interpolation between a number of so-called control points, located at arbitrary positions along each Bezier curve side. The diffused image is the solution of a Laplace equation with colors as Dirichlet boundary conditions. More formally, we define all functions over a 2D domain \( \Omega \) with a point in this domain being denoted as \( p = (x,y) \in \Omega \). Functions over this domain are defined as \( F(x,y) \). Bold function symbols \( \mathbf{F} \) indicate that \( F \) is defined and computed as three separate scalar functions, one for each RGB color channel. Using this notation, a DCI \( \mathbf{L} \) is defined as the following Laplace function:

\[
\Delta \mathbf{L}(p) = 0, \quad p \in \Omega
\]

\[
\mathbf{L}(p) = \mathbf{L}_\Gamma(p), \quad p \in \Gamma.
\]

Here, \( \Gamma \) defines the domain boundary modeled as Bezier curves. \( \mathbf{L}_\Gamma \) refers to the color function on this boundary, i.e., to the interpolated control point colors.
2. Generalized Diffusion Curves Images

Using the notation introduced in Section 1.1, the basis for our image model are two individual Laplace functions \( L_1 \) and \( L_2 \). Both are defined by Equation 1 over the same Bezier curve boundary and both have individual boundary color functions \( L_G \). In practice this means that for each curve side, two separate color functions are specified as opposed to a single one in a DCI. In theory, \( L_1 \) and \( L_2 \) can also have individual sets of control points, but in our implementation we use the same set (with different color values). Using an interpolation function \( W_l \) that spatially blends between \( L_1 \) and \( L_2 \), we first define a "sharp" image \( I_l \) as follows:

\[
I_l(p) = W_l(p)I_1(p) + (1 - W_l(p))I_2(p).
\]

(2)

We will later show that the color of the sharp image \( I_l \) near the boundary is determined by \( L_2 \), and away from the boundary \( L_1 \) takes over, depending on a relative distance measure to the boundary. Figure 3 shows the functions \( L_1 \), \( L_2 \), \( W_l \) and \( I_l \) for the frog example in Figure 7 (a). A detailed definition and justification for \( W_l \) will be given in Section 2.1.

The second component of our model allows blurring boundary edges of the sharp image \( I_l \) via a user-defined blur function along the Bezier curves, analog to the colors. To this end we introduce another interpolation function \( W_b \) that determines image blur across boundaries: it balances minimizing the image gradient in the final GDCI \( I \) against matching the unblurred image \( I_l \) from above. More formally, we define the final GDCI as the following energy minimizing function \( I_l \):

\[
\arg\min_I \int_{\Omega} W_b(p) \|\nabla I_l(p)\|^2 + (1 - W_b(p)) \|I(p) - I_l(p)\|^2 dp
\]

(3)

The following Sections 2.1 and 2.2 describe \( W_l \) and \( W_b \) in more detail, and in Section 2.3 we will present our practical implementation to solve Equation 3.

2.1. Laplace Blending Function \( W_l \)

As mentioned above, \( W_l \) blends between \( L_1 \) and \( L_2 \) in Eq. (3). Intuitively, \( W_l \) must not depend on some absolute boundary distance measure but it must be a smooth, intrinsic, relative distance to all surrounding boundaries. In other words, it should provide a measure of how ‘deep in the domain’ a point \( p \) is. Furthermore, its gradient at boundary curves should be non-zero such that boundary color gradients can be modeled by blending \( L_1 \) and \( L_2 \). To define \( W_l \) we employ a Poisson formulation \( B \), by setting Dirichlet boundary conditions to zero and fix the Laplacian \( l \) everywhere in the domain:

\[
\Delta B(p) = l, \quad p \in \Omega
\]

\[
B(p) = 0, \quad p \in \Gamma
\]

(4)

The parameter \( l \) determines how quickly \( L_1 \) blends into \( L_2 \). For now we set \( l \) to some constant to make the image fitting process feasible, as will become clear in Section 3. Figure 3 (d) was produced using \( l = 8 \) which is used for all examples in this paper because in practice it provided visually balanced results. Figure 4 shows two alternative choices for \( l \) to give an idea how it influences the final result. Using \( B, W_l \) is now defined as:

\[
W_l = \frac{1}{\sqrt{1 + \|\nabla B\|^2}}
\]

(5)

The square root \( \sqrt{B} \) transforms the parabolic cross section of \( B \) into a circular one. For details the interested reader is referred to Section 4.4 and Figure 10 in [SKv*14], who use a similar model for illumination computations. A geometric interpretation of Equation 5 is as follows: imagine \( \sqrt{B} \) being a heightfield defined along an or-
thogonal dimension \( z \) over the image plane \((x, y) = \Omega\), then \( W_j \) is defined as the \( z \)-component of the unit-length normal vector of this heightfield. Figure 3 shows an example of how \( W_j \) blends \( L_1 \) and \( L_2 \). One can see that \( W_j \) is at 0 at the boundary and increases with boundary distance smoothly towards 1.

\( W_j \) controls how the sharp image \( I_s \) is formed: away from boundaries via the function \( L_1 \) and close to the boundary by the function \( L_2 \). Because the gradient of \( W_j \) is non-zero at the boundary, \( L_1 \) directly influences gradients at the boundary. Consequently, this formulation provides similar control like Bilaplace frameworks without introducing the color scaling problem shown in Figure 2. In other words, since \( I_s \) interpolates two Laplace functions, the result always stays within bounds defined by \( L_1 \) and \( L_2 \) no matter how the boundary geometry is modified. Furthermore, by setting \( L_1 = L_2 \), this formulation reduces to the DCI model of Orzan et al. [OBW+08], making it a natural generalization. In summary, by blending two Laplace functions we avoid the associated problems found in Bilaplace frameworks that rely on color function derivatives at boundaries.

Finally, we note that the above definition of \( W_j \) is not unique. For example, it might be defined via diffusing normals as proposed in Johnston [Joh02] and in Boyé et al. [BBG12]. The former article also demonstrates different methods to scale normals afterwards which can be used to further control \( W_j \). In addition, Johnston introduced various ways for combining different color functions (including environment mapping) to obtain complex, realistic coloring and shading effects for supporting designers. By contrast, in this paper we limit the number of interacting parameters such that fitting an input raster image becomes feasible.

### 2.2. Boundary Blur Function \( W_b \)

Blurred edges are often found in natural images, making boundary blur an important component for diffusion curves. The original DCI formulation [OBW+08] defined absolute blur radii. In a Bilaplace framework, blurred edges are modeled with two parallel boundary curves, as was demonstrated with the compound curves feature in Finch et al. [FSH11]. Unfortunately, when a blur kernel (or compound curve in the Bilaplace case) crosses other image boundaries, undesirable artifacts are introduced. This happens because unrelated image content is included into the blur kernel because the latter is modeled independently from other curves. Consequently, special care must be taken when editing boundary geometry and when preparing curve animations.

To remedy this behaviour, in this paper we define the blur relative to the distance to other curves, thus implicitly preventing blurring over other boundary curves. Similar to Orzan et al. [OBW+08], in a DCI the user defines a blur function \( C_T \) along boundary curves using interpolation between control points, similar to the colors. \( C_T \) ranges between 0 and 1, with the latter referring to a maximally blurred boundary. We will now describe how \( C_T \) defines \( W_b \) in Equation 3, which balances between the unblurred image model \( I_s \) presented so far (\( W_b = 0 \)) and a fully blurred one (\( W_b = 1 \)). Any blurred curve with \( C_T > 0 \) should have \( W_b = 1 \) directly on it, and \( C_T \) defines how quickly \( W_b \) vanishes away from the curve: it should instantly vanish at sharp edges (with low \( C_T \)), while it should vanish more gradually for blurry edges. Figure 5 illustrates this for a 1D example and Figure 6 shows a practical 2D example.

We now describe how to model this behavior. Note that blur is a property that is defined independently for each curve and not a function happening between curves. Consequently, if blur values would be diffused just like the colors, even distant boundaries would induce some blur next to a sharp curve, which is clearly undesirable. To avoid this, our idea is to concentrate the diffusion process around the boundaries, thus making boundary influence more local. Specifically, we diffuse boundary values \( C_T \) using an inhomogeneous diffusion, where \( 1 - W_j \) serves as spatial diffusivity as follows:

\[
\nabla \cdot ((1 - W_j)(p) \nabla C(p)) = 0, \quad p \in \Omega
\]

\[
C(p) |_{\Gamma} = C_T, \quad p \in \Gamma
\]

Equation 6

One can see that close to the boundary, the diffusivity is high so that \( C_T \) gets diffused similar to the homogeneous case. However, as \( W_j \) increases further from the boundary, the diffusivity is reduced such that the blur value of the closest boundary dominates at any point \( p \). The resulting function \( C \) is now used to define the blur function \( W_b \), using \( W_j \) again as follows:

\[
W_b(p) = \frac{1}{2} + \frac{1}{2} \cos \pi \min \left(1, \frac{W_j(p)}{C(p)}\right)
\]

Equation 7

In the above formulation we use the fact that \( C \) and \( W_j \) are both defined between 0 and 1, and \( W_j \) continuously increases away from the boundary. If it eventually surpasses \( C, W_b \) vanishes. The point where this happens depends on \( C \) which is determined by the boundary blur function \( C_T \). Consequently, \( W_b \) always starts at...
1 at the boundary and \( C_\Gamma \) determines how far it spreads out into the domain. The cosine term lets the function smoothly approach zero and one (i.e., with zero gradient each). Similarly, a cubic Hermite interpolation with zero gradient constraints could be employed here. Finally, we set \( W_b \) to zero if \( C(p) \) approaches zero in order to ensure numerical stability.

### 2.3. Implementation

There are several options for implementing the presented image model. In related literature, pixel grid-based solvers \([OBW^{*}08, JCW09a, BEDT10, FSH11]\), mesh-based FEM-like solvers \([BBG12]\), and Boundary Element Methods \([SXD^{*}12, IKCM13]\) have been proposed. For interactive editing feedback, we compute the GDCI model by using and extending the fast GPU Laplace solver of Jeschke et al. \([JCW09a]\) working on a pixel grid. It is based on Jacobi iterations with a variably-sized four point stencil. To let information quickly travel over the domain (thus reducing low-frequency error), samples are drawn from distant locations at the beginning. The stencil radius is then gradually reduced to remove local error. A more detailed description of the solver is provided in \([JCW09a]\). The required modifications are described in the following paragraphs.

For computing \( L_1 \) and \( L_2 \), the solver is employed without any modification. More precisely, given a pixel position \( p = (x, y) \) and a spatially varying stencil radius \( r(p) \), in each iteration the pixel value \( L(p) \) is set to:

\[
L(p) = \frac{1}{4} \sum_{w \in U} L(p + r(p)u)
\]

with \( U = \{(1, 0), (-1, 0), (0, 1), (0, -1)\} \) indicating the positions of the four stencil samples around the current pixel position \( p \).

For computing \( B \) in Equation (4), Equation (8) is modified to handle \( I \) as non-zero right hand side. This modification is straightforward: after applying the averaging scheme in (8), the right hand side \( f \) is simply added:

\[
B(p) = \frac{-4r(p)^2}{4} + \frac{1}{4} \sum_{w \in U} B(p + r(p)u)
\]

In the above equation, the multiplication with the squared kernel radius \( r(p)^2 \) automatically accounts for image scaling when zooming, because \( r(p) \) depends linearly on image scale.

For the inhomogenous Laplace Equation 6, Equation 8 is modified by weighting each of the four samples with the spatial diffusivity \( W_i \) (see Equation 6) and the result is renormalized. More formally, Equation 8 is modified as follows:

\[
C(p) = \sum_{w \in U} \left( 1 - W_i \left( p + \frac{r(p)u}{2} \right) \right) \frac{C(p + r(p)u)}{\sum_{w \in U} \left( 1 - W_i \left( p + \frac{r(p)u}{2} \right) \right)}
\]

In order to compute the final blurred image \( I \) in Equation 3, we first need to compute a new kernel radius function \( r_b(p) \) to take advantage of the variable stencil solver performance. For this, pixels with a non-zero blur weight \( W_b \) define a mask where image blur needs to be computed. Consequently, we define \( r_b(p) \) as the distance map to the closest pixel with \( W_b = 0 \), and refer to Jeschke et al. \([JCW09a]\) for methods how to compute it. Using this \( r_b(p) \), Equation (8) is modified to solve eq. (3) as follows:

\[
I(p) = (1 - W_b) I_s(p) + W_b \frac{1}{4} \sum_{w \in U} I(p + r_b(p)u).
\]

### 3. Fitting an Image

When vectorizing raster images, it is desirable to robustly fit a given image to a GDCI representation that can be used for editing and animation afterwards. Starting from some user-defined Bezier curve network with control points attached, we present an algorithm that automatically fits blur and color values to the given control points such that the difference between GDCI and input image is as small as possible. These fitting approaches are described in Sections 3.1 and 3.2 respectively.

#### 3.1. Blur Estimation

Orzan et al. \([OBW^{*}08]\) estimate blur values as a side product of a 2D scale space analysis used for edge detection. By contrast, in this work we assume the boundary geometry being given as input Bezier curves. Using this information, we perform a 1D scale space analysis across boundaries. Our goal is to compute a boundary blur function \( C_\Gamma \) as used in Equation 6 such that edge blur for the input image is well approximated. The fitting proceeds in two general steps: first, blur values are estimated locally along the curves for each piecewise linear curve segment. Afterwards, these dense blur estimation values are fitted to the given blur control points along each curve.

For the first step, we note that blur fitting is nonlinear and different blur values can be chosen for the left and for the right side of each curve (see Section 2.2). Our basic approach is to analyze the 1D image profile for a line orthogonal to each linear curve segment. Along such a line, for each curve side, the boundary blur function \( C_\Gamma \) defines some region extent where the image gets blurred, as is illustrated in Figure 5. This extent is identified for a dense set of potential boundary blur values \( b_i \in \{ \frac{1}{2}, \frac{3}{2}, ..., \frac{19}{2} \} \) \((n = 20 \text{ in our implementation})\) in the following way. We globally set \( C_\Gamma = b_i \) and

---

Figure 6: The blur function \( W_b \) used to produce Figure 7 (a) from \( I_s \) in Figure 3 (d). Darker pixels indicate a higher value \( W_b \). The left image shows the result of the per-segment blur estimation described in Section 3.1. Note the ‘noisy’ local variations. In the right image these estimated values have been fitted to blur control points, providing \( W_b \) as used in Figure 7(a).
compute the according blur function \( W_b \) (see Section 2.2). Now starting at each linear curve segment, we trace along the orthogonal line until \( W_b \) falls below some threshold (0.01 in our implementation) or \( W_b \) increases (which can happen when tracing from concave curves). This process is done independently on both curve segment sides, resulting in a set of \( n \) blur distances \( d_i^L \) and \( d_i^R \) with \( i, j \in \{1, \ldots, n\} \) for each side of a curve segment. Now the 1D input image profile is analyzed along the orthogonal line, using all combinations of \( d_i^L \) and \( d_i^R \) as start- and endpoints of the line segment. The goal is to find the segment (i.e., the scale left and right) which best describes the curve blur (see again Figure 5). For this we apply the scale space analysis of Lindeberg [Lin96]. Specifically, for each combination of \( d_i^L \) and \( d_i^R \), we convolve the according 1D image profile with the Derivative of Gaussian \( \text{DoG}(x) \) scaled over the line segment:

\[
\text{DoG}(x) = \sqrt{\sigma} \frac{-x}{\sqrt{2}\pi\sigma^2} e^{\left(-\frac{x^2}{2\sigma^2}\right)}
\]

with \( x \) in the interval \([-d_i^L - d_i^R, d_i^L - d_i^R]\) and \( \sigma = \frac{1}{2} \frac{d_i^L + d_i^R}{2} \) when convolving with the segment \((-d_i^L, d_i^R)\). According to scale space theory [Lin96], \( \text{DoG}(x) \) is a normalized value, making different segment scales comparable. This means, we can simply pick \( b_l \) and \( b_r \) as left and right blur values belonging to the segment \((-d_i^L, d_i^R)\) that provides the highest absolute convolution value.

Having determined the left and right blur values \( b_l \) and \( b_r \) for every linear segment along the curves, in a final step we simply fit these to the set of blur control points \( x \) using an overdetermined linear system \( Ax = b \). The center point of each linear curve segment (which we used to anchor the orthogonal line segments described above) is a weighted combination of two blur control points. These weights are stored in \( A \) with the estimated values \( b_l \) and \( b_r \) forming the right hand side \( b \). Solving the matrix for the control points \( x \) in a least squares sense is then straightforward. Figure 6 shows an example of \( W_b \) computed from the estimated per-segment blur values (a) and after fitting these blur values to the control points (b).

### 3.2. Color Estimation

For color fitting, we follow the idea in Jeschke et al. [JCW11]. We exploit the fact that the influence of color control points \( z \) on image pixels \( y \) can be stated as a linear system \( Bz = y \) with \( B \) defining the influence of control points to image pixels. Conversely, given pixel colors \( y \) from an input image, we can fit colors to control points \( z \) such that the difference between GDCI output and the input image is minimized in a least squares sense. For this we have to solve \( B^T Bz = B^T y \), which is straightforward. However, because \( B \) is usually too large to fit in a computer memory, the idea is to directly compute \( B^T B \) and \( B^T y \) from a single image where each pixel contains the \( n \) most influential control points. In contrast to Jeschke et al. [JCW11] who use a specially designed diffusion method to obtain these influences in a single diffusion, we compute these influences separately. More precisely, for each pixel we keep a list of the \( n \) most influential control points as follows. First, for each control point, we set its value to 1 and all other control point values to 0 and compute the GDCl (Equation 3). Then, for each GDCl pixel the contribution for this control point is inserted into the list, keeping the \( n \) highest ones and discarding all others. In practice we use \( n = 50 \) which proved sufficiently accurate for all our examples. This process is simpler to implement and more accurate than the one proposed in Jeschke et al., because we store a lot more control point influences per pixel and we also include boundary blur into the influence computation. Assembling \( B^T B \) and \( B^T y \) and solving for \( z \) remains identical to [JCW11] and is omitted here. Figure 3 (a) and (b) show \( L_1 \) and \( L_2 \) fitted from the input image shown in Figure 7 (a).

Figures 1, 7 and 8 provide examples for fitted GDCIs and comparisons to fitted DCIs. We note that our individual control point influence diffusion is is clearly slower than [JCW11], taking several minutes to compute. This is partly due to our unoptimized implementation were we always compute \( W_b \) and \( W_l \) for each control point. DCI and GDCI fitting times are similar in our implementation because the per-control point diffusion process takes up most computation time and we compute \( L_1 \) and \( L_2 \) influence in parallel. We opted in this work for highest possible image quality and leave optimizations and accelerations as future work.

### 4. Results

The proposed GDCI model was implemented in C++ and the DirectX10® graphics API. The solver described in Section 2.3 runs at float precision with 32 iterations and the ’shrink half’ strategy (see [JCW09a]) to compute all functions in this paper (\( L_1, L_2, B, C, \) and \( f \)). Concerning numerical stability, the given boundary colors constrain \( L_1 \) and \( L_2 \) by the maximum principle of harmonic functions. \( B \) as defined in Equation 4 is potentially unbound, but in practice we never found this being a problem. Similarly, intersecting curves do not cause any stability issues, which is important if curves meet exactly at their endpoints. Regarding computational performance, all shown examples render interactively between 7 and 15 frames per second on a 1000 × 1000 pixel grid on our NVidia GeForce GTX 680 graphics card. The curves of all examples were manually created with an editing tool provided by Orzan et al. [OBBW08], which is available online. For the ’frog’, ’face’ and ’duck’ examples we manually defined the control points along curves using the aforementioned tool. For the other examples the set of control points was automatically computed simply by defining a control point every 10 pixels along each curve. Figures 1, 7 and 8 show our results and Table 4 provides related statistics, i.e., the number of Bezier curves, number of color control points, the time required to fit image blur and image colors as described in Section 3, and the image resolution used for fitting. Note that as a means of stylization, we did not apply blur to Figures 1, 7(b), 7(d) and 8.

<table>
<thead>
<tr>
<th>Fig.</th>
<th>#Bezier curves</th>
<th>#ctrl pts</th>
<th>blur fit (seconds)</th>
<th>color fit (seconds)</th>
<th>fitting resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 (a)</td>
<td>40</td>
<td>714</td>
<td>-</td>
<td>69</td>
<td>578x666</td>
</tr>
<tr>
<td>7 (b)</td>
<td>32</td>
<td>245</td>
<td>6.5</td>
<td>101</td>
<td>1000x1000</td>
</tr>
<tr>
<td>7 (c)</td>
<td>55</td>
<td>446</td>
<td>-</td>
<td>44</td>
<td>800x800</td>
</tr>
<tr>
<td>7 (d)</td>
<td>47</td>
<td>412</td>
<td>12.6</td>
<td>165</td>
<td>1000x1000</td>
</tr>
<tr>
<td>7 (e)</td>
<td>30</td>
<td>1242</td>
<td>-</td>
<td>92</td>
<td>900x600</td>
</tr>
<tr>
<td>7 (f)</td>
<td>28</td>
<td>221</td>
<td>13.3</td>
<td>40</td>
<td>1000x1000</td>
</tr>
<tr>
<td>8</td>
<td>159</td>
<td>2118</td>
<td>-</td>
<td>323</td>
<td>800x800</td>
</tr>
</tbody>
</table>

The automatic fitting process generally works very well: note
that no image has been manually edited after the fitting, except for the editing examples provided in Figure 10. In all examples one can see that GDCIs capture a more three-dimensional look compared to the DCI model. The ‘lamp’, ‘chair’, ‘face’, ‘smile’ and ‘car’ examples especially reveal the lack of color variation in DCIs, as is required to convincingly represent shaded curved surfaces. As a re-
result, the DCI color fitting process generates undesirably high color variations at some control points, in an attempt to fit brighter or darker regions further away from the boundary. For the ‘duck’ example we manually traced curves over an image taken from Finch et al. [FSH11]. Note how well the GDCI resembles the original image, requiring a similar number of curves as their Bilaplace model (compare to Figure 1 in Finch et al. [FSH11]). The additional materials document for this paper provides more examples and comparisons to existing approaches.

A convenient property of the proposed GDCI model is that editing colors, curve blur and curve geometry can intuitively be done by a click and drag metaphor, similar to the one proposed in Jeschke et al. [JCW11]. We quickly outline our variant of this method here and Figure 9 gives an example. When the user clicks somewhere in a GDCI image, we determine the Bezier curve point closest to the clicked position. This implicitly selects the two control points adjacent to this boundary point with the respective weights. In addition, the weight value $W_l$ of the clicked position selects either $L_1$ or $L_2$ for the editing operation, such that clicking close to a curve ($W_l \leq 0.3$) will edit $L_2$ while clicking further away from a boundary ($W_l > 0.3$) will edit $L_1$ instead. As a result, a user simply has to click where she wants the GDCI to change most. Figure 1 shows how we edited the lamp using our tool. Figure 10 shows two additional editing examples where we changed the color and geometry of the frog, turning it into another subspecies, as well as the ‘smile’ example, turning her into a vampire. The video accompanying this paper demonstrates this simple and intuitive editing process. The video also shows some animations we performed with the acquired GDCI models, which would not be possible using existing approaches.

5. Conclusions and Future Work

This paper proposes a new GDCI image model as a natural generalization and improvement of DCIs. We show that a spatial blending of multiple Laplace functions provides a similar expressive power as Bilaplace formulations without introducing the related problems discussed in Section 1. Natural images can be fitted directly, and later editing and animation remains equally convenient as with DCIs, which was not possible with existing techniques. At the same time this work narrows the gap in output image quality between curve-based representations and mesh-based or patch-based representations.

Concerning future work, an interesting question is how to obtain Bezier curves directly from an input image, similar to what Orzan et al. [OBW*08] and Xie et al. [XSTN14] did for DCIs and Biharmonic models, respectively. Combining it with texture details as in [JCW11] or environment mapping as proposed in Johnston [Joh02] and Boyé et al. [BBG12] might even further improve output quality. Concerning the model itself, it might be interesting to blend a higher number of Laplace functions $L_i$. This would...
provide even finer control over colors away from the curves with each $L_i$, introducing a comparably small additional cost. An important question in this connection is to which point the added flexibility will be beneficial for designers without overwhelming them with model complexity. Similarly, the blending function $W_j$ could be changed locally, for example, by diffusing the parameter $l$ in (4), which we defined globally in this work. However, this would introduce a non-linear parameter, thereby significantly complicating the image fitting process. Finally, there are numerous applications that make use of smoothly interpolated functions from boundary data, which could directly profit from the proposed idea if more control inside the domain is desirable. Among them are texture draping applications [WOB08] and [FSH11], and gradient domain techniques such as seamless cloning [PGB03], or solid modeling from boundary meshes [TSN10].

Acknowledgements

We thank the anonymous reviewers for their helpful comments and David Hahn for proofreading the paper. We also thank the authors of [OBW*08] and [FSH11] for making their software publicly available, which helped us to generate the results and comparisons. This work was financed by the Austrian Science Fund (FWF) project DEEP PICTURES (no. P24352-N23).

References


[PGB03] Pérez P., Gangnet M., Blake A.: Poisson image editing. ACM Trans. Graph. 22, 3 (July 2003), 313–318. 9


