*Volume 0 (1981), Number 0 pp. 1–11* COMPUTER GRAPHICS *forum*

# A Vectorial Framework for Ray Traced Diffusion Curves

Romain Prévost[1,2] Wojciech Jarosz[2] Olga Sorkine-Hornung[1]

[1]ETH Zurich [2]Disney Research Zurich

**Abstract**

*Diffusion curves allow creating complex, smoothly shaded images by diffusing colors defined at curves. These methods typically require the solution of a global optimization problem (over either the pixel grid or an intermediate tessellated representation) to produce the final image, making fully parallel implementation challenging. An alternative approach, inspired by global illumination, uses 2D ray tracing to independently compute each pixel value. This formulation allows trivial parallelism, but it densely computes values even in smooth regions and sacrifices support for instancing and layering. We describe a sparse, ray traced, multi-layer framework that incorporates many complementary benefits of these existing approaches. Our solution avoids the need for a global solve and trivially allows parallel GPU implementation. We leverage an intermediate triangular representation with cubic patches to synthesize smooth images faithful to the per-pixel solution. The triangle mesh provides a resolution-independent, vectorial representation and naturally maps diffusion curve images to a form natively supported by standard vector graphics and triangle rasterization pipelines. Our approach supports many features which were previously difficult to incorporate into a single system, including instancing, layering, alpha blending, texturing, local blurring, continuity control, and parallel computation. We also show how global diffusion curves can be combined with local painted strokes in one coherent system.*

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

## 1. Introduction

Vector graphics provides several practical benefits over traditional raster graphics, including sparse representation, compact storage, and resolution-independence. Unfortunately, with traditional linear and radial gradient fills, incorporating complex yet controllable effects remains a challenge. Gradient meshes (GMs) provide a powerful way to create smooth-shaded vector images with more complex color variation; however, despite automatic methods to vectorize raster images into GMs [SLWS07, LHM09], authoring and editing remain time-consuming due to the dense mesh representation and complex topological constraints.

Orzan et al.'s [OBW*08] diffusion curves (DCs) provide an alternative that is more flexible and easier to manipulate since it completely removes explicit mesh topology. DCs express the color variation over an image through its boundaries and discontinuities, by defining curves with colors on each side. The images are then created either by solving a differential equation or by interpolating the curves' color constraints. The former requires a global solve over a pixel grid or an intermediate tessellated representation, while the latter can be performed by independent, per-pixel computation. Since the introduction of DCs, several methods have extended this basic idea by enhancing artistic control, providing higher-order continuity, or improving performance, among other features. Unfortunately, combining these improvements into a single system has remained challenging.

We propose a sparse, multi-layer diffusion curves framework that incorporates many complementary benefits of these existing approaches. Our solution avoids the need for a global solve and trivially allows parallel GPU implementation. The general integral formulation allow us to provide support for features such as texturing, instancing, layering, curve continuity control, and curves with local support.

## 2. Background and Related Work

**PDE Approaches.** Orzan et al. [OBW*08] initially formulated the image color as the solution to the Laplace equation, while constraining the color value on the curves. Concurrently, McCann and Pollard [MP08] formulated an equiv-

alent, but rasterized, process as a gradient-domain painting operation. These techniques were inspired by methods that solve Laplace or Poisson equations in other contexts [FLW02, PGB03, SJTS04] to create smooth functions with constrained values or gradients.

Over the past few years, researchers have significantly extended this idea by improving quality or speed [JCW09, BBG12]. Nevertheless, these improvements rely on a global solve which has several practical drawbacks. Firstly, though multi-grid and parallel solvers exist, such general optimizations do not exploit specific knowledge of the diffusion curve problem, therefore limiting the potential speedup. Secondly, since it is not feasible to solve the PDE on an unbounded domain, some boundary conditions need to be prescribed, which can lead to artifacts when panning or zooming. Finally, it can be tedious to extend upon this approach to create additional effects, since either the system needs to be tweaked [BEDT10] or higher-order equations need to be used to incorporate additional types of constraints [FSH11, BBG12].

**Explicit Approaches.** Several recent approaches [BLW11, SXD*12, PQC*12] have abandoned the implicit PDE formulation. Based on observations of Farbman et al. [FHL*09], the global solve can be circumvented by interpolating the boundary constraints using mean-value coordinates. The value of a pixel in a DC image is then expressed explicitly as a weighted average of colors specified on the curves. Though this provides a different solution than the PDE approach, it produces qualitatively similar smooth color functions that interpolate the constraints. More importantly, this formulation allows independent computation of pixels, making it trivial to parallelize.

Bowers et al. [BLW11] proposed a ray tracing approach to evaluate the color contributions of each DC for a given pixel. Additionally, by generalizing curve colors into curve shaders, Bowers et al. provide an elegant and flexible way to incorporate both traditional gradient fills and raster textures into a DC framework. However, the solution is computed on the pixel grid, which breaks the vector graphics paradigm and leads to relatively slow computation, even when parallelized on the GPU. More recently, Pang et al. [PQC*12] used a triangulation to sparsely sample the solution, computed using rasterization instead of ray tracing. Unfortunately, their system discards most of the new flexibility introduced by Bowers et al., and the benefits of rasterization over ray tracing diminish with increasing complexity. Our approach combines the benefit of Pang et al.'s sparse triangulation with the flexibility of Bowers et al.'s method while additionally incorporating several other features.

Recently Sun et al. [SXD*12] used a boundary element method for the Laplace equation, which formulates the solution as a sum of Green's functions. This approach is quite fast and enables integrating the solution over a square region, which allows anti-aliasing. On the other hand, for complex boundary conditions such as diffusion curves, analytic formulas of Green's functions are not known, and need to be approximated. Moreover, Sun et al. approximated occlusion with a culling heuristic which could lead to visual artifacts in the resulting images.

**Vectorial Representation.** Despite optimized multi-grid methods to solve the Laplace equation or GPU implementations for computing color interpolation, both approaches become too slow for high resolution output when computing the solution on the pixel grid. Furthermore, such per-pixel solutions need to be recomputed even after basic operations, such as panning or zooming. However, since the color variations are relatively smooth, one can hope to more sparsely sample the solution.

The main difficulty is to find a representation able to respect discontinuities across curves. One strategy is to discretize the domain using a constrained Delaunay triangulation (CDT), in which the curves are now divided into tiny segments which then become a subset of the edges of the triangulation. In such a representation, we can compute the values only at the vertices, and then interpolate inside the triangles. Takayama et al. [TSNI10] were the first to use such a triangulation in the context of diffusion curves. They compute a CDT in a 2D cut-plane, where the diffusion surfaces (3D equivalent version of DCs) are now curves. They evaluate the color at each vertex by rasterizing the diffusion surfaces, and then interpolate the values inside the triangles using barycentric coordinates. The aforementioned method by Pang et al. [PQC*12] proposed a very similar method in the case of 2D DCs. Previously, Boyé et al. [BBG12] used a triangulation to compute an FEM-based solution to the PDE approach with a biharmonic equation. They solve and interpolate the solution using a quadratic basis.

The benefits of such a vectorial representation against a per-pixel solution are numerous. In particular, many basic editing operations (e.g. panning, zooming) do not require recomputing the solution. Moreover, with a well-defined alpha blending framework, it enables multi-layering and instancing support. Unfortunately, sparse CDT sampling cannot be trivially applied to Bowers et al.'s method since their general curve shaders can introduce arbitrarily high-frequency variation. We show how to overcome this problem. Furthermore, we use a cubic interpolation scheme within each triangle to produce high-quality images that faithfully approximate the per-pixel solution.

**Artistic Control.** Several researchers have investigated additional artistic control for DCs. The already mentioned method of Bowers et al. provides a way to incorporate raster graphics and texturing into a DC framework using curve shaders. Winnemöller et al. [WOBT09] and Jeschke et al. [JCW11], on the other hand, leveraged a DC framework for diffusing UV coordinates and noise parameters to create texturing effects in the context of DC images. Additionally, though

diffusion curves only allow creating sharp discontinuities between their two sides, it is often desirable to produce smooth transitions while controlling the continuity across curves. Orzan et al. originally proposed attaching blur values, corresponding to the radius of a blurring kernel, to the curves. After diffusing these values, the resulting blur radius map is apply as a post-processing step to the final image. Unfortunately, this approach requires the solution be rasterized first and incurs a steep performance cost; hence, it is often disabled during the creation stage. Other strategies consists of somehow changing the PDE, either by adding cleverly chosen soft constraints [BEDT10] or by moving to higher-order equations [FSH11, BBG12], in order to define constraints not only on the values but also on the gradients. Perhaps the most general and sophisticated of these extensions is Finch et al.'s [FSH11] method; however, their approach relies on the PDE formulation with its associated performance limitations. Additionally, while the method offers a rich grammar of possible curve continuity constraints, these additional controls increase the complexity presented to the user. We describe a novel approach to allow continuity control, which seamlessly fits into the explicit formulation without computational overhead.

**Contributions.** In this paper, we present a new framework to create diffusion curve images. We show how to efficiently combine a triangular representation with an extended ray tracing formulation. Sampling the solution only at a sparse set of locations on a triangular mesh allows to efficiently reconstruct the color function, simultaneously capturing sharp discontinuities across curves while avoiding oversampling of smooth regions. The explicit approach additionally makes the diffusion process a local computation which is trivially parallelizable. Furthermore, these choices allow us to naturally extend upon this framework, and support features which were impossible to combine in previous work: general curve shaders, curve continuity control, spatially-varying blur without any post-processing, multi-layering with instancing, and free-hand strokes similar to diffusion curves but with local influence. In the end, we achieve high-quality results with high performance while improving artistic expressiveness.

## 3. System Setup

**Input.** A diffusion curve, as introduced by Orzan et al. [OBW*08] is geometrically described by a 2D spline $\mathcal{C}(t)$ with some colors attached along each side. Colors are usually defined only at a discrete set of control points along the curve for each side and then interpolated in the parameter domain of the curve.

**Color Computation.** While diffusion curves traditionally serve as constraints in a Laplace equation, we chose the alternate formulation proposed by Bowers et al. [BLW11] as a starting point. Based on ray tracing, this approach numerically integrates the contribution of the surrounding visible curves (see Figure 1).

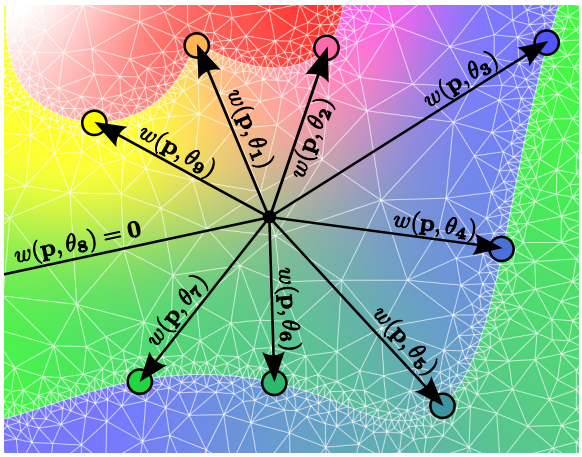



**Figure 1:** *The curves are discretized into segments which serve as input to the constrained Delaunay triangulation. Colors are computed with raytracing in order to gather and weight the contributions of the surrounding curves.*

Given a point $\mathbf{p}$ and an angle $\theta \in \Theta = [0; 2\pi]$, $(\mathbf{p}, \theta)$ fully defines a 2D ray for which we can find the closest intersection (if any) with the set of diffusion curves:

$$\text{hit}(\mathbf{p},\theta) = \mathcal{C}(t) = \mathbf{p} + r(\mathbf{p},\theta)\begin{pmatrix}\cos\theta\\ \sin\theta\end{pmatrix} \tag{1}$$

where $r(\mathbf{p},\theta)$ is simply the distance between $\mathbf{p}$ and the hit point.

The attribute (color or opacity) value $Z(\mathbf{p})$ at the point $\mathbf{p}$ is then computed by the following normalized integral:

$$Z(\mathbf{p}) = \frac{1}{W(\mathbf{p})}\int_\Theta w(\mathbf{p},\theta) z(\mathbf{p},\theta)\mathrm{d}\theta, \quad \text{with} \tag{2}$$

$$W(\mathbf{p}) = \int_\Theta w(\mathbf{p},\theta)\mathrm{d}\theta \tag{3}$$

where $z(\mathbf{p},\theta)$ is the value returned by the attribute shader attached to the curve side hit by the ray, and the weight of the ray $w(\mathbf{p},\theta)$ is traditionally given by the inverse squared distance $r(\mathbf{p},\theta)^{-2}$ (though none of our derivations restrict us to this particular weighting function).

Like Bowers et al. [BLW11], we support two general types of shaders:

- ***Curve-domain shaders*** return an attribute value depending on the point $\mathcal{C}(t)$ of a curve: $\mathcal{C}(t) \to z(\mathcal{C}(t))$. This includes the default shaders where the user defines attributes along the curves which are then interpolated in the curve parameter domain.
- ***Image-domain shaders*** return an attribute value depending on the point $\mathbf{p}$ of the 2D image domain: $\mathbf{p} \to z(\mathbf{p})$. This includes the additional shaders introduced by Bowers et al.,

namely the texture shaders and gradient fill shaders, where $z$ is a texture lookup or a gradient interpolation.

To summarize, the ray contribution $z(\mathbf{p},\theta)$ will be $z(\text{hit}(\mathbf{p},\theta))$ if the ray hits a curve-domain shader, and $z(\mathbf{p})$ if it hits an image-domain shader.

We numerically estimate the integral using 2D Monte Carlo ray tracing with uniformly jittered rays distributed over the circle. In this formulation, the attribute computation is completely independent at each point and can be trivially parallelized.

In our framework, a curve can have shaders for color and/or shaders for opacity (one for each side of the curve). Optionally the user can also attach blur radii, otherwise the blur radius assumed zero for the whole curve side.

## 4. A Sparse Vectorial Framework

**Triangulation.** Computing the solution on the pixel grid is expensive and often unnecessary, since the final image presents smooth color variations between the curves. Instead, we rely on a constrained Delaunay triangulation (CDT) constructed from the diffusion curves (see Figure 1). The curves are discretized into segments which serve as constrained edges for the triangulation algorithm. While computing the values just on the vertices and using barycentric interpolation is an obvious candidate, this can lead to distracting Mach banding artifacts due to the limited continuity. We also tried quadratic interpolation, but found cubic interpolation necessary to ensure good smoothness (see Figure 2 for a close-up comparison). This interpolation scheme requires computing ten values per triangle (one per vertex, two per edge and one at the barycenter), as shown in Figure 3.

We reconstruct the final image by rasterizing the triangles with cubic polynomial interpolation implemented in a fragment shader. This captures the complex variations of the analytic solution, while avoiding ray tracing for every pixel.

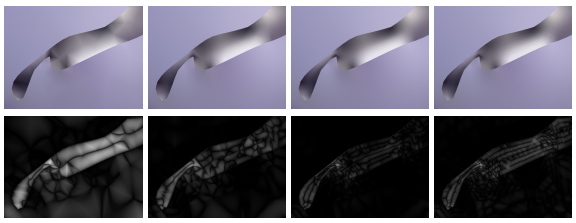
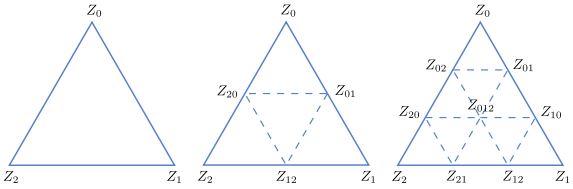



**Figure 2:** *Barycentric and quadratic interpolation result in visible artifacts. Cubic interpolation produces smooth results and provides better continuity than using barycentric interpolation on the nine sub-triangles (see Figure 3(c)). We also show the difference to the ground truth per-pixel solution in the bottom row (the values were scaled 8 times for better legibility).*




(a) Barycentric (b) Quadratic (c) Cubic

**Figure 3:** *Triangular patches of different interpolation order and the evaluation points.*

Moreover, evaluation points shared by multiple triangles at vertices and on the edges are only computed once, resulting in an effective average of four to five evaluation points per triangle (instead of ten). We provide the complete set of interpolation formulas in Appendix A.

The constrained Delaunay triangulation allows to naturally represent the discontinuities across the curves by keeping different values for vertices lying on constrained edges. For example, a vertex of the triangulation lying on a black/white curve will be black or white depending on the triangle we are considering. The only ambiguity arises at the endpoint vertices. Whereas Boyé et al. [BBG12] used a special interpolation scheme for these triangles to have radially varying colors, we obtain a similar result by assigning an angle-dependent value to these vertices. For example, if the two colors of the singularity are black and white, then, in each of the one-ring triangles, the color of the endpoint vertex is a different shade of grey depending on the angle between the triangle and the curve. Figure 4 provides a visual explanation of this process. While this solution theoretically creates small $C^0$-discontinuities between the radial edges, in practice their small size makes them hard to notice. Additionally, it allows us to use a single interpolation scheme for all the triangles without having to deal with special cases in the rendering.

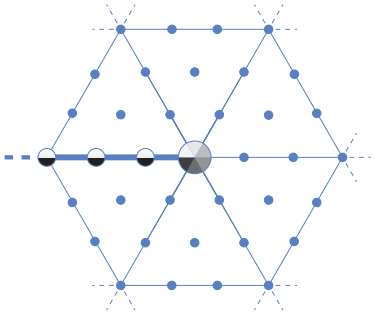


**Figure 4:** *The bold curve is constrained to white color on one side and black color on the other side, creating a singularity at the endpoint. Using the angle between the triangles' barycenters and the curve segment, we radially interpolate and use shades of grey for the endpoint vertex in each triangle. The rest of the points (in blue) are unconstrained, so they are computed using ray tracing.*

This representation is very well-suited for vector graphics, since it is fully vectorial and thus naturally supports operations such as panning, zooming, rotating, and instancing without any recomputation. To ensure sufficient quality of the reconstructed image, we can adjust triangulation quality parameters such as maximal area or minimal angle. Since the ray tracing computation is local, only the viewport needs to be triangulated, in contrast to PDE approaches which require prescribed boundary conditions to avoid discretizing the entire 2D plane.

**Curve Shaders Support.** One of the benefits of the ray tracing approach lies in the introduction of additional curve shaders by Bowers et al. [BLW11], namely gradient fill and texture shaders. In these cases, the attribute values are not defined along the curve but in the image domain, as described in Section 3. Unfortunately, sparse interpolation of such shaded values would result in severe under-sampling artifacts, especially for texture shaders with high-frequency details. Luckily, though image-domain shaders may have arbitrarily high frequency content, the total weight of each shader varies more smoothly across the CDT mesh, suggesting a potential for interpolation.

To support sparse sampling in the presence of image-domain shaders, we keep track of and interpolate the weights for each shader, while looking up image-domain shader values per-pixel during the rendering stage. This can be achieved by splitting the integral (2) depending on the shader hit:

$$Z(\mathbf{p}) = \frac{1}{W(\mathbf{p})}\left(\overbrace{\int_{\Theta_C} w(\mathbf{p},\theta) z(\mathbf{p},\theta)\mathrm{d}\theta}^{Z_C(\mathbf{p})} + \sum_{s=1}^{S}\int_{\Theta_s} w(\mathbf{p},\theta) z(\mathbf{p},\theta)\mathrm{d}\theta\right) \tag{4}$$

where $\Theta_C \bigcup \left(\cup_{s=1}^{S}\Theta_s\right) \subset \Theta$ partitions the angular domain between rays hitting a curve-domain shader and rays hitting a specific image-domain shader of index $s$.

Since for image-domain shaders the attribute value $z(\mathbf{p},\theta) = z_s(\mathbf{p})$ is independent of the hit point we can pull it out of the integral,

$$\int_{\Theta_s} w(\mathbf{p},\theta) z(\mathbf{p},\theta)\mathrm{d}\theta = \underbrace{\int_{\Theta_s} w(\mathbf{p},\theta)\mathrm{d}\theta}_{W_s(\mathbf{p})}\ z_s(\mathbf{p}), \tag{5}$$

which results in the following simplified formulation:

$$Z(\mathbf{p}) = \frac{Z_C(\mathbf{p})}{W(\mathbf{p})} + \sum_{s=1}^{S}\frac{W_s(\mathbf{p})}{W(\mathbf{p})} z_s(\mathbf{p}). \tag{6}$$

Equation (6) demonstrates that by keeping the accumulated weights $W_s(\mathbf{p})$ we can postpone the evaluation of the image-domain shaders. During the numerical integration, each ray will only contribute to:

- $Z_c(\mathbf{p})$ and $W(\mathbf{p})$ if it hits a curve-domain shader
- $W_s(\mathbf{p})$ and $W(\mathbf{p})$ if it hits the image-domain shader with index $s$.

For each evaluation point of the triangulation, we compute and store $Z_C, W, W_1, ..., W_S$. To reconstruct the value $Z(\mathbf{p})$ inside the triangles, we use Equation (6) where the ratios $Z_C/W(\mathbf{p}), W_1/W(\mathbf{p}), ..., W_S/W(\mathbf{p})$ are interpolated with cubic coordinates and $z_1(\mathbf{p}), ..., z_S(\mathbf{p})$ are evaluated per-pixel from the shaders (for example by texture lookup). Figure 5 shows an example with two curves that have varying colors on one side and a texture on the other side.

## 5. Artistic Control

In Sections 3 and 4 we presented our sparse vectorial framework for ray traced diffusion curve. In this section, we show how we leverage this design in order to provide artistic control in a single unified framework.

**Continuity Control.** When a ray hits a curve, by default we only consider the shader located on the front side of the curve. This leads to sharp discontinuities, if both sides have different shader values. We now generalize our previous formulation. By blending the front side and back side shaders depending on the hit distance we can provide an intuitive way to control interpolation continuity across curves.

More formally, we have

$$z(\mathbf{p},\theta) = \beta(\mathbf{p},\theta)\, z_{\mathrm{F}}(\mathbf{p},\theta) + (1-\beta(\mathbf{p},\theta))\, z_{\mathrm{B}}(\mathbf{p},\theta), \tag{7}$$

with blending coefficient

$$\beta(\mathbf{p},\theta) = \mathrm{smoothstep}\left(\mathrm{clamp}\left(\frac{r^{\perp}(\mathbf{p},\theta)+R}{2R}, 0, 1\right)\right) \tag{8}$$

where $r^{\perp}(\mathbf{p},\theta)$ is the projected distance between $\mathbf{p}$ and $\mathrm{hit}(\mathbf{p},\theta)$ along the normal direction, and $R$ is a specified maximum blur radius. Basically, this function smoothly goes from 0.5 near the curve to 0 at distance $R$ in the normal direction, such that if $\mathbf{p}$ is further that $R$ it will only use the front side shader and if it lies near the curve it will blend both the

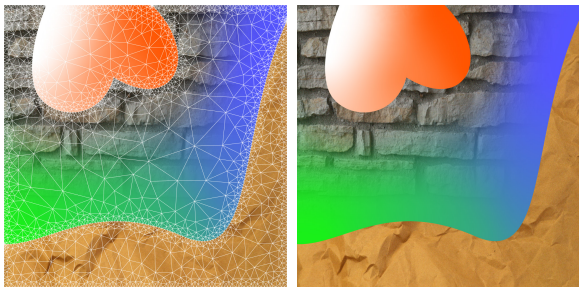


**Figure 5:** *When image-domain shaders are attached to a curve, we gather only their weights. This example shows that this allows arbitrarily high frequency details to be representated with a sparse computation and interpolation scheme.*

no blur | varying blur | two-sided blur | with a texture shader | EG logo

**Figure 6:** *Our system supports having no blur for sharp discontinuities, as well as varying blur radii along curves for smoothness control, and different blur radii on either side for full control over the normal derivative – for both image-domain and curve-domain shaders. On the far right we show a simple example where the blur radii help to create smooth color transitions.*

front side and the back side shaders. The default behaviour corresponds to a blur radius equal to zero, i.e. $\beta(\mathbf{p},\theta) = 1$, which means that $z(\mathbf{p},\theta)$ always returns the front side shader, as expected.

If we plug Equation (7) inside the integral formula (2), we obtain:

$$Z(\mathbf{p}) = \frac{1}{W(\mathbf{p})}\left(\int_\Theta \beta(\mathbf{p},\theta)w(\mathbf{p},\theta)z_\mathrm{F}(\mathbf{p},\theta)\mathrm{d}\theta + \int_\Theta (1-\beta(\mathbf{p},\theta))\,w(\mathbf{p},\theta)z_\mathrm{B}(\mathbf{p},\theta)\mathrm{d}\theta\right) \tag{9}$$

which can be split similarly to the derivation in Section 4, finally leading to the exact same formula:

$$Z(\mathbf{p}) = \frac{Z_C(\mathbf{p})}{W(\mathbf{p})} + \sum_{s=1}^{S}\frac{W_s(\mathbf{p})}{W(\mathbf{p})}\,z_s(\mathbf{p}), \tag{10}$$

but now with:

$$\begin{aligned} Z_C(\mathbf{p}) &= \int_{\Theta_C^\mathrm{F}} \beta(\mathbf{p},\theta)w(\mathbf{p},\theta)z_\mathrm{F}(\mathbf{p},\theta)\mathrm{d}\theta \\ &\quad + \int_{\Theta_C^\mathrm{B}} (1-\beta(\mathbf{p},\theta))\,w(\mathbf{p},\theta)z_\mathrm{B}(\mathbf{p},\theta)\mathrm{d}\theta, \\ W_s(\mathbf{p}) &= \int_{\Theta_s^\mathrm{F}} \beta(\mathbf{p},\theta)w(\mathbf{p},\theta)\mathrm{d}\theta \\ &\quad + \int_{\Theta_s^\mathrm{B}} (1-\beta(\mathbf{p},\theta))\,w(\mathbf{p},\theta)\mathrm{d}\theta \end{aligned} \tag{11}$$

Note that we must now partition the angular domain $\Theta$ to consider separate front side and the back side contributions. We denote this with a superscript $\Theta^\mathrm{F}$ and $\Theta^\mathrm{B}$ for the front and back side respectively. This handles the situation where a ray $(\mathbf{p},\theta)$ might hit a curve-domain shader on the front and an image-domain shader on the back, or two different image-domain shaders, and therefore should contribute to two different integrals.

The user can attach blur radii to each side of a curve using a curve-domain shader, allowing for varying the blur radius along the curve. In this case, we replace $R$ in Equation (8) by $R_\mathrm{F}(\mathbf{p},\theta)$, i.e. the blur radius on the front side of the curve at the hit location. Though small blur radii can lead to rapid color variations, we found that our cubic interpolation scheme can reliably handle these cases (see Figure 6).

The blur radii provide for intuitive control of the type of continuity across the curve, as well as the normal derivatives at the curves. In particular, we show in Appendix B that for any point on a curve the normal derivative is equal to $3(z_\mathrm{F}-z_\mathrm{B})/4R_\mathrm{F}$ and is therefore controlled by the blur radius. As shown in Figure 6, the user is able to create $\mathcal{C}^{-1}$, $\mathcal{C}^0$, and $\mathcal{C}^1$ transitions across curves with various transition speeds.

**Opacity, Instancing, and Multi-Layering.** To create complex images, multiple layers are often required. But, to our knowledge, the only multi-layering system for diffusion curves proposed attaching RGBA colors to the curves. In such a framework, every curve influences both opacity and color. We instead propose a decoupled formulation in which curves can specify only color, only opacity or both. This approach gives the user more freedom and flexibility to design complex opacity masks.

In our system, each layer has its own set of diffusion curves and its own Delaunay triangulation constrained over all its curves. Each evaluation point of the triangulation is now either on a curve with color, on a curve with opacity, on a curve with both, or completely free in space. We thus distinguish

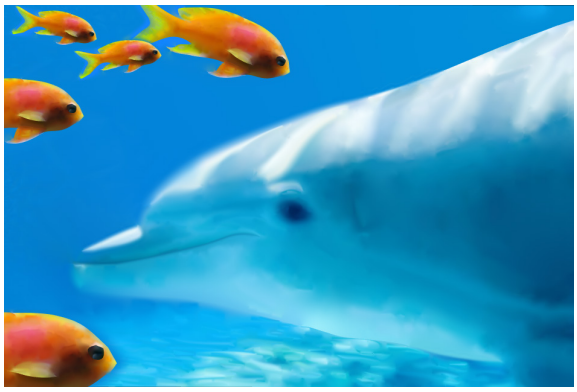


**Figure 7:** *The yellow fish are several instances of the same layer, which is possible thanks to our vectorial representation.*

between these four cases and only compute the missing attribute values. This is done using our ray tracing framework twice: once for colors, and once for opacity. At this point, all the color and opacity information is stored on the evaluation points of the triangulation, and the image can be synthesized.

Concerning multi-layering, our vectorial representation offers a direct benefit. In fact, it allows to instantiate layers without any recomputation. Figure 7 shows an example where the yellow fish is drawn on a separate layer, and then instantiated with several scales and rotations. In this case, the opacity is determined by a single curve enclosing the fish with opacity one on the inside, zero on the outside, and some blur radius to create a smooth transition.

**Local Curves.** The influence of a diffusion curve is global and hence sometimes hard to control. Simple tasks such as adding a highlight can be quite difficult to achieve in a diffusion curves framework. Though one possible solution would be to combine several curves with multi-layering, this can become cumbersome. Instead, we incorporate local curves, which we define very similarly to our global diffusion curves, but with a local influence controlled by the user.

A local curve $\mathcal{C}(t)$ has its own independent layer (invisible to the user), and is defined with:

- curve-domain color shaders $z(t)$ (one for each side)
- blur radii $R(t)$ to blend shaders
- influence radii $Q(t)$ to choose the size of our new locally-supported weighting functions

If we ignore visibility, we can rewrite Equations (2) and (3) as integrals over a single curve. We can then formulate the integration over the parameter domain $t$ of the single curve:

$$Z(\mathbf{p}) = \frac{1}{W(\mathbf{p})} \int_{\mathcal{C}} w(\mathbf{p},t) z(\mathbf{p},t) \mathrm{d}t \tag{12}$$

$$W(\mathbf{p}) = \int_{\mathcal{C}} w(\mathbf{p},t) \mathrm{d}t \tag{13}$$

where we also defined a new weighting function with smooth local support:

$$w(\mathbf{p},t) = 1 - \mathrm{smoothstep}\left(\min\left(\frac{r(\mathbf{p},t)}{Q(t)}, 1\right)\right) \tag{14}$$

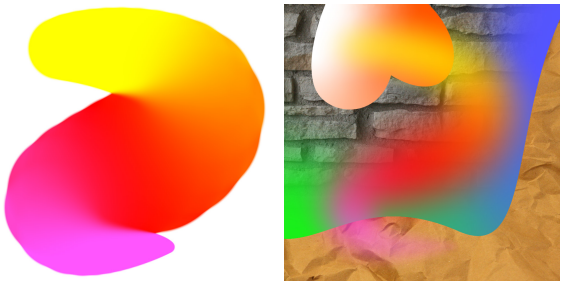


**Figure 8:** *Example of local curves with varying influence radius and color. (Left) Only $Z(\mathbf{p})$ without opacity, (Right) Local curve blended on top of an other layer.*

where $Q(t)$ is the influence radius varying along the curve.

The shader value $z(\mathbf{p},t)$ stays the same blending between front and back side, but written as a function of $t$:

$$z(\mathbf{p},t) = \beta(\mathbf{p},t)\, z_{\mathrm{F}}(t) + (1 - \beta(\mathbf{p},t))\, z_{\mathrm{B}}(t) \tag{15}$$

$$\beta(\mathbf{p},t) = \mathrm{smoothstep}\left(\mathrm{clamp}\left(\frac{r^{\perp}(\mathbf{p},t) + R(t)}{2R(t)}, 0, 1\right)\right) \tag{16}$$

where $R(t)$ is the blur radius varying along the curve.

With these new definitions, we use $Z(\mathbf{p})$ for the color, and $\alpha W(\mathbf{p})$ as opacity, where $\alpha$ is a scale factor. Doing so, produce complex strokes similar to diffusion curves but with local support (see Figure 8). Since, in this case we ignore visibility, ray tracing is not needed anymore, and using a stochastic sampling of the curve, we can compute the solution per-pixel in a fragment shader. Since the curve has local support, we only evaluate this for pixels where the contribution can potentially be non-zero (within the influence radius of the curve).

This technique holds some similarities with the work by Sun et al. [SXD*12], but whereas their weighting functions are approximated Green's functions of the Laplace equation, ours are specifically designed for smooth local support.

## 6. Results and Discussion

**Implementation Details.** We implemented our framework in C++ using CUDA for the ray tracing and parallel color computation. The images are rendered with a resolution of about 1 megapixel. The CDT is generated using the `Triangle` library [She02]. To obtain a high-quality triangulation, we choose by default a minimal angle of 22 degrees, and a maximal triangle area of 4% of the artwork size. We found these values to work well, but they can also be adjusted by the user. The ray tracing is accelerated using a simple $192 \times 192$ regular grid, constructed on the CPU, and then transferred as a texture to the GPU.

The final image is rasterized via the graphics pipeline. In order to do so, we aggregate the values for each evaluation point in Texture Buffer Objects. We send the triangulation as a Vertex Buffer Object, which contains the indices to look up in the texture. The vertex shader proceeds with the lookup and sends the values to the fragment shader. Finally, the fragment shader runs the cubic interpolation to compute the color of the fragment.

**Performance & Quantitative Results.** We measured performance on a 12-core 3.4 GHz Intel® Core™ i7-4930K CPU with a NVidia® GeForce GTX780 3GB. Table 1 gathers timings for the different parts of the algorithm. Our framework is able to handle complex images with hundreds of curves at interactive rates while not only computing color but also opacity. Computing the constrained Delaunay triangulation is very fast, which, in addition to quality reasons,

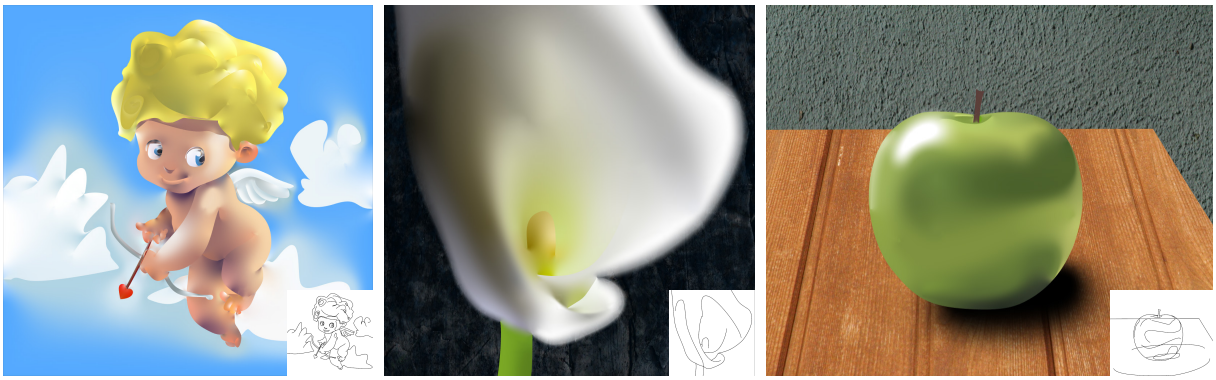


**Figure 9:** *(Left)* CUPID *with our sparse vectorial framework, (Middle) A* LILY *with a depth of field effect, (Right) An* APPLE *on a wood table with textures and a local curve for the highlight.*

confirms it is a good candidate for an intermediate sparse representation of diffusion curves. In the end, we are able to render high quality images using 64 rays per evaluation point with timings ranging from tens of milliseconds for simple examples to hundreds of milliseconds for very complex examples. Moreover, we support features such as blur radii and texture shaders, whereas previous frameworks usually need to disable these to maintain interactive performance.

Without access to previous work's source code, it is difficult to draw very precise conclusion, especially since performance is highly correlated to the choice of examples, the features enabled, as well as the choice of parameters for the solvers (number of iterations, resolution, etc.); however, we can still draw some rough conclusions about performance. Compared to Bowers et al.'s demo, our sparse representation seems to allow a significant speedup. In fact, we are able to maintain interactive rates even for high-resolution output with texture shaders, which in their case would require many more rays to trace. More precisely, for the TOMATOES and LADYBUG examples, our sparse representation provides a 16× speed improvement compared to an evaluation at very pixel. Our triangulation also enables efficient hardware anti-aliasing, providing further speedup compared to the expensive supersampling required by Bowers et al.'s approach. Jeschke et al.'s state-of-the-art Laplacian multi-grid solver, and Pang et al.'s sparse rasterization algorithm, are one order of magnitude faster than our framework. However, this speed comes at the cost of reduced artistic control; in particular, sacrificing support for image-domain shaders or continuity control. Boyé et al. report timings similar to ours. Our implementation is not highly optimized, and we believe a significant speedup could be achieved by more cleverly choosing the number of rays to trace for each evaluation point, as well as avoiding unnecessary CPU/GPU data transfers.

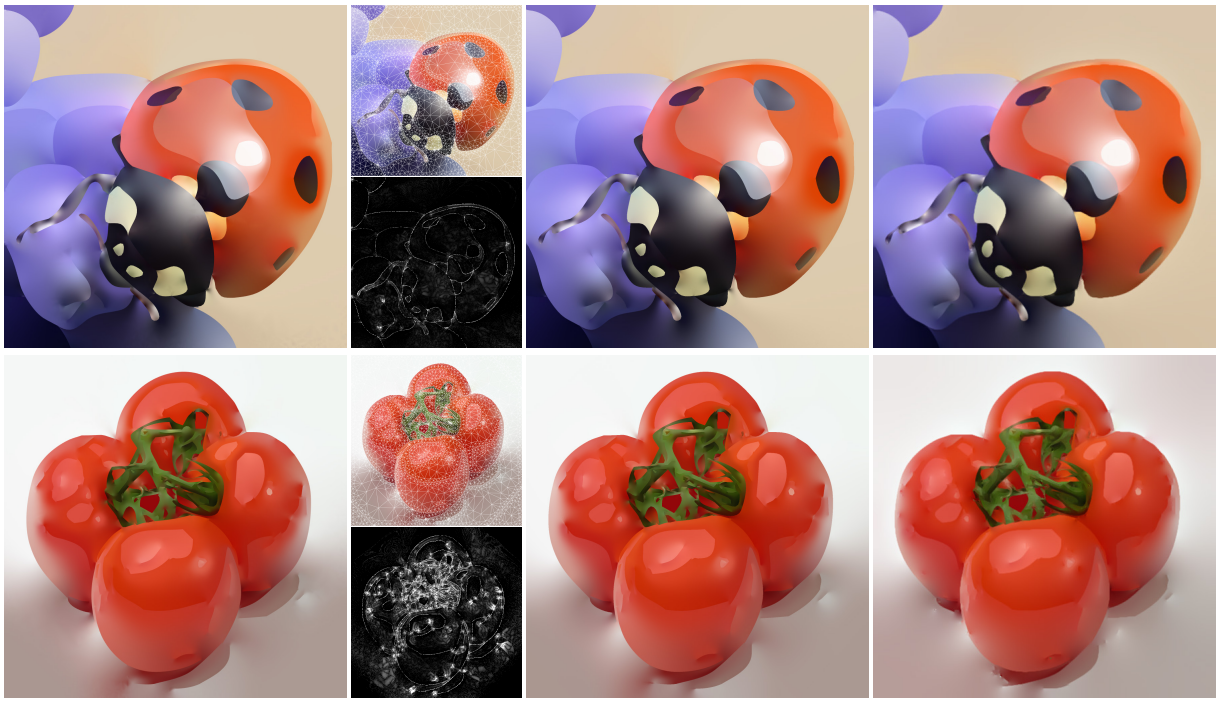
**Qualitative Results & Comparisons.** We tested our framework both with previous work examples and new examples to evaluate robustness, quality and additional artistic control. All our results are rendered as high-resolution bitmaps in the paper, but we also refer the reader to our supplemental material for vector images displayed in a WebGL viewer.

Figure 9 shows a few examples created with our framework. CUPID was created by an artist, and shows a more complex example of a diffusion curves image. LILY demonstrates how our continuity control feature allows to produce blurry transitions across diffusion curves, and in this case creates a convincing depth-of-field effect. APPLE is a two-layered example, which makes use of textures for highly detailed parts of the image, as well as a local curve for the highlight.

Figure 10 presents a qualitative comparison of diffusion curves images generated with our framework and Orzan's Poisson solver. Our explicit approach produces visually similar results even though the solutions are theoretically different. Moreover, comparison to ground truth (per-pixel computation, equivalent to Bowers' solver) shows that our sparse computation leveraging the CDT mesh is sufficient to reliably reconstruct the image. The difference images demonstrate that, even when the values are scaled 128 times, the differences primarily lie in how edges are antialiased by the graphics hardware. Figure 11 shows how our continuity control feature can produce similar results to the spatially varying post-process blur applied in some previous works. Our

**Table 1:** *Timings and statistics.*

| | LADYBUG | TOMATOES | FISH | LILY |
|---|---|---|---|---|
| #curves | 72 | 395 | 1859 | 13 |
| #triangles | 9k | 10k | 53k | 6k |
| #vertices | 5k | 5k | 27k | 3k |
| #evaluation points | 37k | 38k | 113k | 23k |
| #rays/evaluation points | 64 | 64 | 72 | 72 |
| Geometry | 18ms | 19ms | 61ms | 41ms |
| Triangulation | 7ms | 5ms | 16ms | 9ms |
| Color computation | 49ms | 53ms | 277ms | 56ms |
| Resolution | $1024^2$ | $1024^2$ | $944 \times 633$ | $1024^2$ |
| Rendering | 0.6ms | 0.8ms | 5.5ms | 8.9ms |



Our method | Triangulation Difference 128× | Ground truth (equiv. to Bowers et al.) | Orzan et al.

**Figure 10:** *Comparison between our sparse reconstruction and the ground truth per-pixel solution (corresponding to Bowers et al.). The second column shows the underlying triangulation as well as the difference image (128×). Differences primarily lie in how edges are antialiased by the graphics hardware. Orzan et al. is also included for qualitative comparison.*

method, however, comes nearly for free in terms of computation, whereas applying the blur map is quite expensive, and often needs to be disabled during editing.

**Figure 11:** *(Left) Our method with continuity control, (Right) Result from Orzan et al. with blur as a post-process*

**Limitations and Future Work.** Using the OpenGL pipeline with specialized shaders offers a simple way to display the triangulation by passing colors and various weights as Texture Buffers. We leverage the vertex shader to optimize this when possible, but since the graphics hardware imposes a maximum number of varying floats (48 in our case), we postpone this lookup to the fragment shader if we need to interpolate more values. In those cases (FISH and LILY), this can lead to suboptimal performance (see Table 1). Optimizing data usage in our rendering pipeline would allow to maintain high performance in these cases.

Texture Buffers are only supported by modern versions of OpenGL. In particular, WebGL does not support this feature yet. While some tricks could be used to circumvent this problem, we instead preferred using vertex attributes and barycentric interpolation inside the nine subtriangles, which widely extends the range of graphics cards able to handle our WebGL viewer.

We also make use of GLSL for rendering local curves. We chose to sample the curves with their parameters (colors, influence radii, normals, etc.) on the CPU and pass these values as arrays of uniform variables. This restricts the number of samples and can compromise the quality of the rendered local curve. To circumvent this issue, one could adaptively sample the local curves directly on the GPU, which would additionally allow for a more intelligent sampling of the integrals.

On the theoretical side, the endpoints of the curves are usually problematic to handle. In our case, the problem is twofold. First, as mentioned in Section 4, our solution for the endpoint vertices potentially creates small $C^0$-discontinuities. Deriving and using special patches around the singularities as proposed by Boyé et al. [BBG12] could remove the dis-

continuities if these small artifacts are deemed objectionable. Secondly, due to the rapid change of the visibility, our solution differs from the PDE approaches. This problem is inherent to most explicit approaches, and improving this behaviour is an interesting avenue for future work. For instance, one could incorporate the endpoints with angle-dependent color lookups in the integration.

Finding the best triangulation to represent the image is an interesting question. Though we currently focus on still images, diffusion curve animations would be an interesting extension. In this case, our current triangulation approach could lead to flickering artifacts if applied independently per frame. It would be interesting to incorporate some form of temporal coherence directly into the triangulation.

Taking inspiration from Jarosz et al.'s work on 2D global illumination [JSKJ12], we investigated computing color gradients in an alternative evaluation scheme to perform higher-order interpolation. In fact, using 4 values and 3 gradients per triangle instead of our current 10 values would require less rays. However, obtaining good gradient estimates proved to be challenging, compromising the smoothness of the interpolation. This led us to abandon this direction. Further investigation may nonetheless prove fruitful.

## 7. Conclusion

We have presented a new framework for computing diffusion curves images, which produces high quality results at interactive rates. Combining an explicit approach based on ray tracing with a sparse triangulation discretization, we are able to efficiently synthesize fully vectorial diffusion curves images. We do this while not sacrificing support for advanced features such as image-domain shaders, and we also improve artistic control by introducing a new continuity control technique, a multi-layering system, as well as curves with local influence.

**Acknowledgements.** We thank Maurizio Nitti for creating CUPID and LILY. We also thank our colleagues from DRZ and IGL, in particular Alec Jacobson, for insightful discussions. We are also grateful to the anonymous reviewers for their extensive help in improving this paper.

## References

[BBG12] BOYÉ S., BARLA P., GUENNEBAUD G.: A vectorial solver for free-form vector gradients. *ACM Trans. Graph. (Proc. SIGGRAPH Asia) 31*, 6 (Nov. 2012). 2, 3, 4, 9

[BEDT10] BEZERRA H., EISEMANN E., DECARLO D., THOLLOT J.: Diffusion constraints for vector graphics. In *Proc. Intl Symp. on NPAR* (2010). 2, 3

[BLW11] BOWERS J. C., LEAHEY J., WANG R.: A ray tracing approach to diffusion curves. *Computer Graphics Forum (Proc. EGSR)* (2011), 1345–1352. 2, 3, 5

[FHL*09] FARBMAN Z., HOFFER G., LIPMAN Y., COHEN-OR D., LISCHINSKI D.: Coordinates for instant image cloning. *ACM Trans. Graph. (Proc. SIGGRAPH) 28*, 3 (July 2009). 2

[FLW02] FATTAL R., LISCHINSKI D., WERMAN M.: Gradient domain high dynamic range compression. *ACM Trans. Graph. (Proc. SIGGRAPH) 21*, 3 (July 2002). 2

[FSH11] FINCH M., SNYDER J., HOPPE H.: Freeform vector graphics with controlled thin-plate splines. *ACM Trans. Graph. (Proc. SIGGRAPH Asia) 30*, 6 (Dec. 2011). 2, 3

[JCW09] JESCHKE S., CLINE D., WONKA P.: A GPU Laplacian solver for diffusion curves and Poisson image editing. *ACM Trans. Graph. (Proc. SIGGRAPH Asia) 28*, 5 (Dec. 2009). 2

[JCW11] JESCHKE S., CLINE D., WONKA P.: Estimating color and texture parameters for vector graphics. *Computer Graphics Forum (Proc. Eurographics) 30*, 2 (Apr. 2011). 2

[JSKJ12] JAROSZ W., SCHÖNEFELD V., KOBBELT L., JENSEN H. W.: Theory, analysis and applications of 2D global illumination. *ACM Trans. Graph. 31*, 5 (Sept. 2012). 10

[LHM09] LAI Y.-K., HU S.-M., MARTIN R. R.: Automatic and topology-preserving gradient mesh generation for image vectorization. *ACM Trans. Graph. (Proc. SIGGRAPH) 28*, 3 (July 2009). 1

[MP08] MCCANN J., POLLARD N. S.: Real-time gradient-domain painting. *ACM Trans. Graph. (Proc. SIGGRAPH) 27*, 3 (Aug. 2008). 1

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

[PGB03] PÉREZ P., GANGNET M., BLAKE A.: Poisson image editing. *ACM Trans. Graph. (Proc. SIGGRAPH) 22*, 3 (July 2003). 2

[PQC*12] PANG W.-M., QIN J., COHEN M., HENG P.-A., CHOI K.-S.: Fast rendering of diffusion curves with triangles. *IEEE Computer Graphics and Applications 32*, 4 (2012). 2

[She02] SHEWCHUK J. R.: Delaunay refinement algorithms for triangular mesh generation. *Computat. Geom. 22*, 1-3 (2002). 7

[SJTS04] SUN J., JIA J., TANG C.-K., SHUM H.-Y.: Poisson matting. *ACM Trans. Graph. (Proc. SIGGRAPH) 23*, 3 (Aug. 2004). 2

[SLWS07] SUN J., LIANG L., WEN F., SHUM H.-Y.: Image vectorization using optimized gradient meshes. *ACM Trans. Graph. (Proc. SIGGRAPH) 26*, 3 (July 2007). 1

[SXD*12] SUN X., XIE G., DONG Y., LIN S., XU W., WANG W., TONG X., GUO B.: Diffusion curve textures for resolution independent texture mapping. *ACM Trans. Graph. (Proc. SIGGRAPH) 31*, 4 (July 2012). 2, 7

[TSNI10] TAKAYAMA K., SORKINE O., NEALEN A., IGARASHI T.: Volumetric modeling with diffusion surfaces. *ACM Trans. Graph. (Proc. SIGGRAPH Asia) 29*, 6 (Dec. 2010). 2

[WOBT09] WINNEMÖLLER H., ORZAN A., BOISSIEUX L., THOLLOT J.: Texture design and draping in 2D images. *Computer Graphics Forum (Proc. EGSR) 28*, 4 (2009). 2

## Appendix A: Triangular patch interpolation

We denote $u, v, w$ as the barycentric coordinates ($u+v+w=1$ and $u, v, w \geq 0$). The rest of the notation refers to Figure 3.

In the linear case, the formula is simply the well-known barycentric interpolation from 3 vertex values: $Z(u,v,w) = w\,Z_0 + u\,Z_1 + v\,Z_2$. Quadratic interpolation requires 3 additional values chosen at the edge centers. The interpolated

value is then computed as:

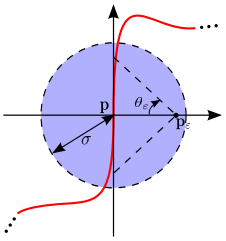
$$\begin{aligned} Z(u,v,w) = {} & w(2w-1)\,Z_0 + u(2u-1)\,Z_1 + v(2v-1)\,Z_2 \\ & + 4wu\,Z_{01} + 4uv\,Z_{12} + 4vw\,Z_{20}. \end{aligned} \tag{17}$$

The cubic interpolation requires 10 values. We choose the 3 vertices, the $1/3$- and $2/3$-points of the edges, as well as the triangle center. This choice leads to the following interpolation formula:

$$\begin{aligned} Z(u,v,w) = {} & 0.5w(3w-1)(3w-2)\,Z_0 \\ & + 0.5u(3u-1)(3u-2)\,Z_1 \\ & + 0.5v(3v-1)(3v-2)\,Z_2 \\ & + 4.5wu(3w-1)\,Z_{01} + 4.5wu(3u-1)\,Z_{10} \\ & + 4.5uv(3u-1)\,Z_{12} + 4.5uv(3v-1)\,Z_{21} \\ & + 4.5vw(3v-1)\,Z_{20} + 4.5vw(3w-1)\,Z_{02} \\ & + 27wuv\,Z_{012}. \end{aligned} \tag{18}$$

## Appendix B: Normal derivatives

Let us consider a point $\mathbf{p}$ on a curve (see figure below). We consider a coordinate system where the X-axis is aligned with the normal $\mathbf{n}(\mathbf{p})$. We will consider the point $\mathbf{p}_\varepsilon = \mathbf{p} + \varepsilon\,\mathbf{n}(\mathbf{p})$, and angles $\theta$ will be taken with respect to $-\mathbf{n}(\mathbf{p})$ in the clockwise direction.




For simplicity, we will assume that there exists a sufficiently small neighborhood (representated in blue) of radius $\sigma$ around $\mathbf{p}$ inside which:

- the only geometry is the local neighborhood of the curve around $\mathbf{p}$ and can be approximated by the segment $(0,\lambda)$ with $\lambda \in [-\sigma;+\sigma]$,
- the curve shader values on both sides $z_F$ and $z_B$ are constant,
- the curve has a constant blur radius $R_F$ greater than $\sigma$

Since this neighborhood can be as small as we want these assumptions are not very restrictive. Some of them could be relaxed, but it would require a more extensive proof to carefully study the asymptotic behavior.

To find the normal derivative, we will study the limit of the following quantity:

$$\frac{Z(\mathbf{p}_\varepsilon) - Z(\mathbf{p})}{\varepsilon} = \frac{\displaystyle\int_\Theta w(\mathbf{p}_\varepsilon,\theta)\,(z(\mathbf{p}_\varepsilon,\theta) - Z(\mathbf{p}))\,\mathrm{d}\theta}{\varepsilon\,W(\mathbf{p}_\varepsilon)} \tag{19}$$

In order to do so, we split the integrals into 2 parts:

$$\Theta_\varepsilon = [-\theta_\varepsilon;+\theta_\varepsilon] \quad \text{and} \quad \overline{\Theta}_\varepsilon = \Theta \setminus \Theta_\varepsilon \tag{20}$$

where $\theta_\varepsilon = \arctan\left(\frac{\sigma}{\varepsilon}\right)$.

**Integration over $\overline{\Theta}_\varepsilon$.** Since the shader values are bounded (here we assume between 0 and 1) and there is no geometry near $\mathbf{p}_\varepsilon$ for these angles, we can easily show that the following integrals are trivially bounded:

$$\left|\int_{\overline{\Theta}_\varepsilon} w(\mathbf{p}_\varepsilon,\theta)\mathrm{d}\theta\right| \leq \frac{2\pi}{(\sigma-\varepsilon)^2} \tag{21}$$

$$\left|\int_{\overline{\Theta}_\varepsilon} w(\mathbf{p}_\varepsilon,\theta)\,(z(\mathbf{p}_\varepsilon,\theta) - Z(\mathbf{p}))\,\mathrm{d}\theta\right| \leq \frac{2\pi}{(\sigma-\varepsilon)^2}. \tag{22}$$

**Integration over $\Theta_\varepsilon$.** On this interval, the geometry is a segment between $(0,-\sigma)$ and $(0,+\sigma)$, so we can analytically evaluate the integral of the weighting function. We have:

$$w(\mathbf{p}_\varepsilon,\theta) = r(\mathbf{p}_\varepsilon,\theta)^{-2} = \frac{1}{y^2+\varepsilon^2} = \frac{1}{\varepsilon^2}\frac{1}{1+\tan^2\theta}, \tag{23}$$

so,

$$\begin{aligned} \int_{\Theta_\varepsilon} w(\mathbf{p}_\varepsilon,\theta)\mathrm{d}\theta &= \frac{1}{\varepsilon^2}\int_{-\theta_\varepsilon}^{+\theta_\varepsilon} \frac{\mathrm{d}\theta}{1+\tan^2\theta} \\ &= \frac{\theta_\varepsilon + \cos\theta_\varepsilon \sin\theta_\varepsilon}{\varepsilon^2} \underset{\varepsilon\to 0}{\sim} \frac{\pi}{2\varepsilon^2}. \end{aligned} \tag{24}$$

Moreover, we assumed that the shader values are constant with respect to the angle on this tiny segment, which leads to:

$$\begin{aligned} \int_{\Theta_\varepsilon} w(\mathbf{p}_\varepsilon,\theta)\,(z(\mathbf{p}_\varepsilon,\theta) - Z(\mathbf{p}))\,\mathrm{d}\theta \\ = (z(\mathbf{p}_\varepsilon) - Z(\mathbf{p}))\int_{\Theta_\varepsilon} w(\mathbf{p}_\varepsilon,\theta)\mathrm{d}\theta \end{aligned} \tag{25}$$

We already have the asymptotic behavior of the integral. The first part can be determined by plugging in the definitions:

$$\begin{aligned} z(\mathbf{p}_\varepsilon) &= \beta(\varepsilon)\,z_F + (1-\beta(\varepsilon))\,z_B, \\ \beta(\varepsilon) &= \mathrm{smoothstep}\left(\frac{R_F+\varepsilon}{2R_F}\right), \\ Z(\mathbf{p}) &= \frac{z_F+z_B}{2}. \end{aligned} \tag{26}$$

After simplification, we obtain:

$$(z(\mathbf{p}_\varepsilon) - Z(\mathbf{p})) \underset{\varepsilon\to 0}{\sim} \frac{3\varepsilon}{4R_F}(z_F - z_B), \tag{27}$$

and therefore:

$$\int_{\Theta_\varepsilon} w(\mathbf{p}_\varepsilon,\theta)\,(z(\mathbf{p}_\varepsilon,\theta) - Z(\mathbf{p}))\,\mathrm{d}\theta \underset{\varepsilon\to 0}{\sim} \frac{3\pi}{8\varepsilon R_F}(z_F - z_B). \tag{28}$$

**Final Result.** While the integrals over $\overline{\Theta}_\varepsilon$ are bounded, the ones over $\Theta_\varepsilon$ tend to infinity when $\varepsilon$ tends to 0, so they dictate the asymptotic behavior of the full integrals:

$$\frac{Z(\mathbf{p}_\varepsilon) - Z(\mathbf{p})}{\varepsilon} \underset{\varepsilon\to 0}{\sim} \frac{\frac{3\pi}{8\varepsilon R_F}(z_F - z_B)}{\varepsilon\,\frac{\pi}{2\varepsilon^2}} = \frac{3(z_F - z_B)}{4R_F}, \tag{29}$$

which demonstrates that the blur radius controls the normal derivative at the curve.