Eurographics Symposium on Rendering 2011
Ravi Ramamoorthi and Erik Reinhard
(Guest Editors)

Volume 30 (2011), Number 4

A Ray Tracing Approach to Diffusion Curves

John C. Bowers Jonathan Leahey Rui Wang

University of Massachusetts Amherst

Abstract

*Diffusion curves [OBW*08] provide a flexible tool to create smooth-shaded images from curves defined with colors. The resulting image is typically computed by solving a Poisson equation that diffuses the curve colors to the interior of the image. In this paper we present a new method for solving diffusion curves by using ray tracing. Our approach is analogous to final gathering in global illumination, where the curves define source radiance whose visible contribution will be integrated at a shading pixel to produce a color using stochastic ray tracing. Compared to previous work, the main benefit of our method is that it provides artists with extended flexibility in achieving desired image effects. Specifically, we introduce generalized curve colors called* shaders *that allow for the seamless integration of diffusion curves with classic 2D graphics including vector graphics (e.g. gradient fills) and raster graphics (e.g. patterns and textures). We also introduce several extended curve attributes to customize the contribution of each curve. In addition, our method allows any pixel in the image to be independently evaluated, without having to solve the entire image globally (as required by a Poisson-based approach). Finally, we present a GPU-based implementation that generates solution images at interactive rates, enabling dynamic curve editing. Results show that our method can easily produce a variety of desirable image effects.*

1. Introduction

Vector graphics, in which images are represented by geometric elements such as curves, continues to serve as a powerful tool for creating resolution-independent images and drawings. Such representations are compact, scalable, and easy to edit and animate. The *diffusion curve*, as introduced by [OBW*08], is a new vector-based representation that enables the easy creation of smooth-shaded images. A diffusion curve is defined with different colors on each side of the curve. Given a set of such curves, an image can be computed by solving a Poisson equation that smoothly diffuses the colors from the curves to the image interior. As diffusion curves allow for arbitrarily shaped curves, including open ones, they provide a convenient and intuitive way to generate a variety of shaded images. Since the introduction of the original diffusion curves, several papers have significantly improved this technique towards higher performance [JCW09a], more expressive control [BEDT10], and additional applications domains [JCW09b, WOBT09, TSNI10, JCW11].

Despite these advances, the main technique for constructing an image from diffusion curves remains solving a Poisson equation. This solution imposes several limitations. First, it requires rasterizing the analytic curves into a discrete pixel buffer as input, which is prone to aliasing and numerical computation artifacts [JCW09a]. Second, the default Poisson diffusion process requires the whole image to be solved globally, and provides little flexibility in controlling how the resulting image is generated. Although the diffusion constraints technique proposed by [BEDT10] can be used to enable flexible control such as diffusion barriers and anisotropic diffusion, it is still expensive to compute and does not yet run at interactive speed. Third, and most importantly, as the diffusion process requires a gradient vector field which is fundamentally different from the way classic vector graphics is represented, enabling diffusion curves often means a substitute, instead of a complement, to existing tools. As a result, users have to redefine certain familiar operations in a less convenient way. One example is the radial gradient fill, which is a frequently used operation but rather cumbersome to define and manipulate using diffusion curves. While it's possible to combine different techniques using image layer compositing, doing so often introduces unnecessary complexity, making it difficult to achieve desired image effects. Thus the lack of seamless integration between classic vector graphics and diffusion curves has become an obstacle for reusing existing content and skills that users are already familiar with.

Figure 1: *A depiction of a barn owl. The curves are shown in the inlay on the left. Both images are computed using our method. Left: curves are assigned only colors as in standard diffusion curves; right: curves are defined with radial gradient fills and textures.*

In this paper we present a new method for solving diffusion curves using raytracing. We formulate the problem into an alternative form equivalent to final gathering in global illumination. Specifically, the curves define source colors whose contributions (accounting for occlusion) will be integrated at a shading pixel to produce a smoothly interpolated color. The integration is achieved using stochastic raytracing performed on the GPU. By leveraging a uniform grid acceleration structure and a two-stage image adaptive sampling algorithm, we achieve interactive speed for generating solution images of complex diffusion curves. This enables the user to dynamically create and edit curves on the fly.

Compared to existing solutions, the main benefit of our method is that it provides artists with extended flexibility in achieving desired image effects. Specifically, building upon the raytracing solution, we introduce generalized curve colors called *shaders* that can seamlessly integrate diffusion curves with classic 2D graphics including both vector graphics (e.g. gradient fills) and raster graphics (e.g. patterns and textures). This is achieved by associating each curve with a user specified shader that defines the shading contribution when a ray intersects that curve. We further extend the curves with additional attributes such as custom weighting function and a transparency factor, providing flexible control of each curve's contribution. In addition, our method allows any pixel in the image to be independently evaluated, without having to solve the entire image at once, as required by a Poisson-based approach. This enables local and output-driven image computation, suitable for view-dependent rendering. Finally, using raytracing allows curves to be represented algebraically, eliminating the numerical computation artifacts caused by curve rasterization in a diffusion based approach. It can also easily achieve anti-aliasing in the final generated image by using spatial stochastic sampling.

In its base form, our idea is closely related to coordinates image cloning [FHL*09], which replaces the diffusion process within a curved boundary by a smooth interpolation using the mean-value coordinates [Flo03]. However, unlike them, our method explicitly considers the visibility of a curve to a shading pixel, which is crucial for constructing complex diffusion curve images but negligible for the purpose of image cloning. Moreover, our curves are defined with custom shaders and attributes, thus the contribution of each curve must be numerically integrated and cannot use the analytic mean-value coordinates.

As an alternative to raytracing, one could also use rasterization to compute interpolated colors. Such a method has been adopted by the recent work of diffusion surfaces [TSNI10], which extends diffusion curves to 3D to generate solid textures. Theoretically, both raytracing and rasterization would work, and the choice goes back to the classic debate between them. However, in our case, we have found raytracing to provide several clear benefits. First, it enables the definition of arbitrary curve attributes including transparency, which would be harder to achieve using rasterization. Second, our experiments show that using raytracing for diffusion curves is in fact faster than rasterization, mainly because the number of curve elements in a complex image is typically larger than the number of sample rays required to integrate colors. This causes rasterization to under-utilize the GPU, slowing down performance. Finally, rasterization is prone to aliasing artifacts, which often have to be solved via supersampling. In contrast, raytracing can efficiently achieve anti-aliasing by using stochastic sampling.

In sum, our contributions are: 1) a raytracing approach to diffusion curves that allows the seamless integration with classic vector and raster graphics; 2) extended curve attributes such as weighting functions and transparency to provide flexible control; 3) a GPU-based implementation achieving interactive frame rates. Figure 1 shows an example of our rendered images. Note how the textures and gradient fills add pleasing details to the rendering.

2. Related Work

Curves are basic drawing elements in vector-based tools such as Adobe Illustrator and CorelDraw. Typically vector curves are defined with various attributes such as color, gradients, and patterns, which collectively define a shaded image. For complex shading effects, the gradient mesh is a powerful tool which represents an image as a lattice mesh, with smooth color transitions controlled by mesh vertices. While techniques exist to automatically optimize complex gradient meshes [SLWS07], it remains difficult and time-consuming to create gradient meshes free-hand from scratch.

In [OBW*08], the diffusion curve was proposed as a new vector-based primitive which allows arbitrarily shaped curves to shade an image through the diffusion process. Compared to classic vector graphics, this can provide a more convenient and intuitive way to design smooth-shaded images. The underlying computation, which is solving a Poisson equation to simulate the diffusion, is also a fundamental element in a variety of other applications such as tone

mapping [FLW02], image stitching and cloning [PGB03], and image matting [SJTS04]. Because solving the Poisson equation is computationally expensive for large images, most existing methods employ an efficient multigrid solution [BFGS03, KH08]. In addition, GPU-based multigrid solvers have also been studied [MP08, JCW09a] to facilitate interactive applications. In [Aga07], a quadtree method was proposed, which exploits adaptive subdivision to significantly improve the computation speed of solving large-scale diffusion problems.

As the diffusion process requires a gradient vector field that is fundamentally different from how classic vector graphics is represented, employing diffusion curves often means certain familiar operations, such as the radial gradient fill, must be redefined (e.g. by using an inner and outer circular curves) to match the new representation. This makes such operations inconvenient to edit. Other operations such as the pattern fill are currently unsupported by diffusion curves.

The standard diffusion curve formulation does not allow the diffusion process to be modified. Recently, Bezerra et al. [BEDT10] addressed this issue by introducing several flexible diffusion constraints, including the diffusion barrier, color strength, anisotropic diffusion, and non-local constraints. While some of these flexibilities are shared with our method, their underlying model is still the diffusion process, thus they do not support gradient fills or textures. In addition, they still require the image to be solved globally, thus the constraints can significantly increase the computation cost, requiring 4∼5 seconds to generate a 512×512 image.

For image cloning, Farbman et al. [FHL*09] proposed to replace the diffusion process with smooth interpolation using the analytic mean-value coordinates [Flo03, JSW05]. Their method is based on the observation that the Poisson cloning essentially constructs a minimal surface which is well captured by a harmonic-like interpolant such as mean-value coordinates. Although the interpolation does not necessarily produce identical results with the Poisson equation, they are typically indistinguishable, and there is no perceptual evidence that either solution produces better quality than the other. Therefore interpolation becomes an attractive alternative that provides benefits in both computation speed and memory cost. However, this method cannot be directly used for solving diffusion curves, as it ignores the visibility of curves, causing colors to spread across curve boundaries.

Diffusion curves have also been applied in [JCW09b] to define surface details as vector-based textures. This allows sharp features on the surface to be preserved upon closeup examination. However, as discussed in their paper, several issues must be carefully handled, including the view-dependent nature of surface texturing and the aliasing of curve details upon extreme closeups. Both issues have to do with the fact that a diffusion-based solution must solve the entire image globally and on a finite resolution pixel grid. In contrast, our raytracing based solution avoids these issues as it can compute solutions for an arbitrary sets of pixels.

Using diffusion curves for texturing has also been studied in [WOBT09], where they use diffusion curves to define uv coordinates across the textured region. Their method focuses on texturing closed regions that do not interact with the rest of the image. In contrast, our method defines texture shaders for curves, which can interact with other curves and provide more flexibility without relying on multiple layers. Another related work is a technique for estimating color/texture parameters for vector graphics, as proposed by [JCW11]. Their goal is to employ a procedural noise function to generate textures that mimic natural images.

In [TSNI10], Takayama et al. extended diffusion curves to diffusion surfaces, which can be used to easily construct 3D textures. They compute rendering result on a cross section of the texture by using GPU-based positive mean-value coordinates [LKCOL07]. This approach is similar to ours, but uses rasterization instead of raytracing. It is currently not interactive, requiring seconds to half a minute to compute each frame at a sparse set of sample points. As discussed in Section 1, for the purpose of solving diffusion curves, we have found raytracing to provide clear advantages over rasterization, not only in flexibility and quality, but also in speed.

3. Algorithms and Implementation

Overview. As with previous work, our images are described by curves with shading information defined on each side of the curve. The simplest shading information is color, as in standard diffusion curves; but we extend it to a more general concept called *shaders*, which allow us to incorporate classic vector and raster graphics tools such as gradient fill and textures. Below we first describe our raytracing formulation and explain shaders; we then describe additional attributes that can customize the contribution of each curve; finally, we present implementation details, and a two-stage image adaptive sampling algorithm to achieve interactive framerates.

Raytracing Formulation. Our basic algorithm is quite intuitive to understand: we treat the curves as virtual light sources that emit radiance energy to the surrounding space. Given this, the color at any 2D point $\mathbf{p}$ is simply assigned as the total radiance I (accounting for occlusion) received at that point, which is is calculated as:

$$I(\mathbf{p}) = \int_0^{2\pi} L(\mathbf{x}_i(\mathbf{p},\theta))\, w(\mathbf{x}_i(\mathbf{p},\theta))\, d\theta \qquad (1)$$

Here $\mathbf{x}_i(\mathbf{p},\theta)$ is the curve point that a ray originating from $\mathbf{p}$ intersects in direction θ. It accounts for visibility, thus a curve point only makes contribution to $\mathbf{p}$ if there is no other curve blocking the path. L is the source radiance of $\mathbf{x}_i$ obtained from the shading information stored at $\mathbf{x}_i$. w is the normalized weight that determines the contribution of the source. It is typically based on the distance between $\mathbf{x}_i$ and $\mathbf{p}$

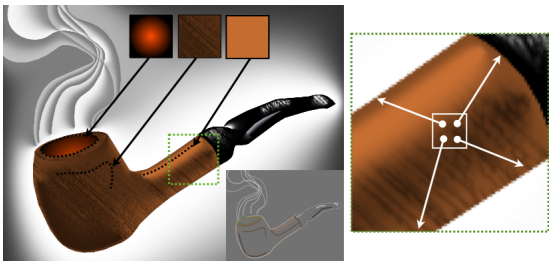


Figure 2: *Our algorithm works by tracing stochastic rays from a pixel (shown on the right) into the 2D scene of curves, computing intersections, and integrating the results (Eq. 1). Each curve is assigned a shader defining the source radiance (indicated by the squares in the left image), a weighting function and a transparency factor.*

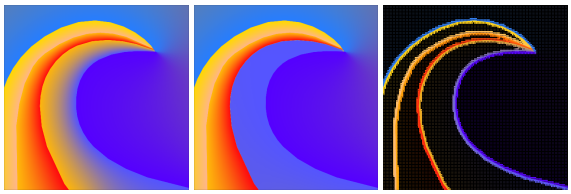
but can include other factors. Finally, the integration is over the range from 0 to 2π.

Note that this formulation is similar to the rendering equation in global illumination [Kaj86], thus can be evaluated using stochastic ray tracing at each receiver point $\mathbf{p}$. By picking different weighting functions w, we can influence how a curve contributes to its nearby space. In some special cases, the integral can be analytically computed. For instance, if w is set to be inversely proportional to the distance between $\mathbf{x}_i$ and $\mathbf{p}$ (i.e. $w \sim \frac{1}{|\mathbf{x}_i - \mathbf{p}|}$), L is defined by linear interpolation and occlusion is ignored, then the integral has an analytic solution known as the mean-value interpolation [Flo03, JSW05]. In the general case, however, the integral has no analytic solution and therefore must be numerically evaluated.

Shaders. Our raytracing formulation allows for the development of a set of *shaders* to define the source radiance L. The first is a basic *color shader* which provides an interpolated color along the curve, and the value is invariant to the location of the receiver point. This is equivalent to the definition of color in standard diffusion curves.

The second shader, called the *gradient fill shader*, allows the user to define a classic gradient fill operation attached to the curve. The most commonly used fill operations include the linear, radial, and angular fills, which are currently supported in our implementation. Other choices can be easily added, including arbitrary procedurally defined gradient fills. To create a gradient fill shader, the user will place down two control points, define their colors, and select the type of fill operations. When a ray hits the curve, the source radiance is calculated from the location of the receiver point relative to the two control points. Note that in this case, although the source color is not defined along the curve, the curve serves as a boundary that confines the influence of a fill operation to only one side of the curve.

The third shader, called the *texture shader*, allows the user to define a raster texture attached to the curve. This shader



(a) (b) (c)

Figure 3: *(a) and (b) compare the effect of changing the relative curve weight w_c from 1 to 0 on the right side of the gold curve. At $w_c = 0$, the curve stops making contribution to the space on its right side. (c) shows the adaptive sample points computed during the interactive mode of our algorithm.*

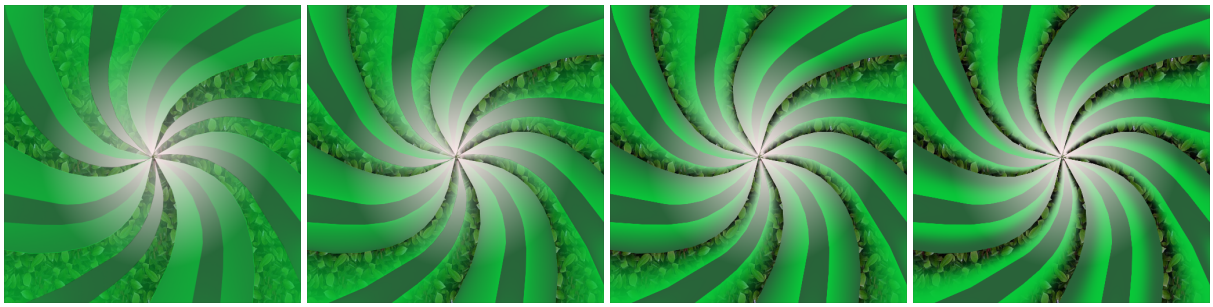
is specified by giving two control points indicating the lower left and upper right corners of a texture image. When a ray hits the curve, the source radiance is calculated by a texture lookup using the location of the receiver point relative to the two control points.

Fig. 2 shows an illustration of the three shaders described above. Note that by using a raytracing formulation, we unify the treatment of all shaders and allow them to interact in the same layer. This integrates diffusion curves with vector and raster graphics tools seamlessly, each of which is best suited for certain tasks.

Weighting function. We define the weighting function w in Eq. 1 as the product of two terms: $w = w_c \cdot w_d$. The first term w_c is a constant that controls the overall influence of a curve relative to other curves. A large w_c increases the relative contribution of a curve. On the contrary, setting $w_c = 0$ effectively makes a *barrier* curve which does not contribute to its surrounding space. This is similar to the concept of diffusion barrier in [BEDT10]. As each side of the curve can define a different weighting function, a zero weight is useful when only one side of the curve needs to specify colors but not the other side. Fig. 3 shows an example of setting a curve's weight $w_c = 0$.

The second term w_d is a distance based weighting defined as $w_d(\mathbf{x}_i, \mathbf{p}) = |\mathbf{x}_i - \mathbf{p}|^{-p}$, which is the negative p-th power of the distance between $\mathbf{x}_i$ and $\mathbf{p}$. Typically $p > 0$, thus this term attenuates the contribution of a source point that is far away from $\mathbf{p}$. The default value of p is 1. As p increases, the influence of a curve becomes increasingly more concentrated around itself; whereas when p decreases, the distance plays a less important role, leading to a more uniform shading. Fig. 4 (a-d) shows a comparison example.

Transparency. An additional attribute of the curve is the transparency α_c, which allows a curve's influence to penetrate through other curves. This can be easily achieved in raytracing by allowing the ray to continue traveling after the first intersection. The total return value of the ray is calculated as $(1 - \alpha_c) \cdot I_f + \alpha_c \cdot I_b$, where I_f is the contribution of the current intersection point (foreground), and I_b is the contribution beyond it (background), which can be recursively



(a) $p = 0$ (b) $p = 1/2$ (c) $p = 1$ (d) $p = 2$

Figure 4: *(a)-(d) compare different distance weighting functions w_d applied on the pinwheel image.*

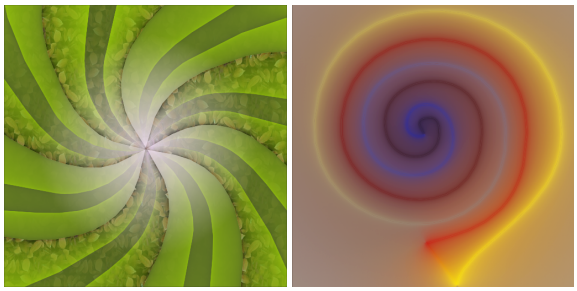


Figure 5: *These two images demonstrate the effect of transparency. The left example is the same pinwheel as shown above but defined with different colors and with curve transparency enabled. Note the difference in the transparent spirals, especially the dark ones. The right example shows how the curve transparency leads to a translucency effect.*

defined. Fig. 5 show two examples of transparency effect. Note that compared to Fig. 4, the pinwheel image in this example shows transparent spirals.

GPU Implementation. We have implemented our algorithm on the GPU using the Thrust library [HB10]. There are three main steps. For simplicity, all curves are defined as cubic Bézier curves. The first step is to subdivide the Bézier curves into a set of line segments using a standard subdivision algorithm. For each line segment we store the curve parameter values t of the two end points. Second, we build a uniform grid using a parallel algorithm from the set of line segments, and use this structure to accelerate raytracing. We also experimented with a BVH but found the uniform grid to work better in practice, because it is inexpensive to compute, and is well-suited for scenes with many curves distributed over the entire image. We set the grid resolution to $2\sqrt{N} \times 2\sqrt{N}$ for a scene with N segments. When more curves are added, the performance using the uniform grid does not degrade significantly. The final step is to perform raytracing in parallel for every pixel of the target image.

Our default image resolution is 512×512, and we trace 128 rays per pixel using stratified random sampling in the angles. The origin of each ray is assigned using a 2×2 subpixel jittered sample pattern, which provides reasonable antialiasing in the rendering without increasing the computation cost. Note that instead of discretizing the curves into line segments, we could also trace Bézier curves directly by solving cubic polynomials. This can potentially help reduce the ray-curve intersection cost.

Image adaptive sampling. Since all computations are performed in 2D and on the GPU, it is relatively fast to trace a large number of rays. Still, to provide more interactivity while the user creates and modifies curves, we exploit the smoothness of a diffusion curves image to enable simple image adaptive sampling. While the user interacts with the software, our algorithm performs raytracing in two stages. The first stage partitions the image into 8×8 pixel blocks, and computes shading at all block corner pixels with 64 rays per pixel. For each sampled pixel we store its shading value as well as the shortest intersection distance among all the traced rays. This shortest distance value indicates whether a pixel is sufficiently close to a nearby curve. Then in the second stage, we collect all blocks that have at least one corner pixel with a shortest distance less than ε, and perform raytracing at all 8×8 pixels belonging to that block. For the other blocks, the interior pixels are bilinearly interpolated from the corner pixels. We currently set ε to 8 pixels wide, which works quite well in practice. This two-stage adaptive sampling algorithm can perform at 14∼25 fps with reasonably good quality (refer to the paper video). We can optionally perform a 3×3 median filtering to reduce the sampling noise. The right image of Fig. 3 shows the adaptive sample pixels computed for the wave image. These pixels will be raytraced. Other pixels, which are interpolated, are not shown. As soon as the user has been inactive for a certain period of time (1 second by default), our algorithm will invoke a full rendering pass that traces 128 rays at every image pixel. This will provide a high-quality rendered image at 2∼3 fps.

4. Results and Discussions

In this section we present results highlighting the flexibility of our method in integrating different types of shaders and customizing the curve contributions. We demonstrate how this flexibility allows artists to easily create interesting and

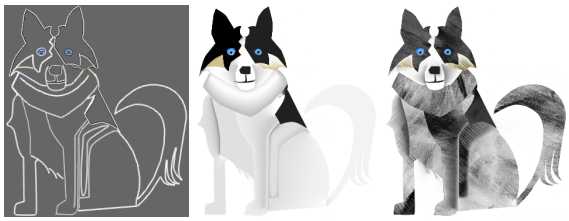


Figure 6: *A cartoon of a dog. Left: the set of curves. Middle: result from applying only color shaders. Right: result from applying both color and texture shaders on the curves, achieving a particular aesthetic similar to painted paper collage. Note the interaction between the color and texture shaders, particularly in the shading of the dog's leg in the center of the right image.*

desirable image effects without resorting to compositing. All images shown in this paper are rendered at 512×512 resolution, and results were collected on a PC with Intel Core i7 2.67GHz CPU and an NVIDIA GTX470 GPU.

Shaders. Fig. 1 shows a drawing of a barn owl. Both images are computed using our method. In the left image, curves are assigned only color shaders, simulating standard diffusion curves. In the right image, we applied radial gradient fill shaders in several places to introduce shading details. For example, the right outer boundary curves of the owl has two radial gradient shaders defined to create the red highlights in the sky; and the moon is defined by a white to dark blue radial gradient shader to create a perfectly circular halo. In addition, texture shaders are added to the curves defining the owl and the tree branch to produce a stylized effect. The effect of two different texture shaders interacting in an open region can be seen in this image as well. The top of the wing's checkerboard pattern fades into the slate texture of the grey feather. This sort of blending would require more careful masking to achieve using a layered approach.

Fig. 2 highlights the use of several different shaders, indicated by the squares pointing to their corresponding curves. Fig. 6 shows an example of texture shaders applied on a cartoon image of a dog. In the middle image, curves are assigned only color shaders; whereas in the right image, some curves are assigned texture shaders, simplifying the creation of fine details in the image and providing a stylish textured rendering. Note the interactions between the standard colored curves and textured curves, which would be difficult to reproduce using image layer compositing techniques.

Fig. 7 shows another example: the bird is drawn with only a few curves, but our method produces a detailed painterly style rendering achieved by applying texture shaders to the curves. Note that on the bird's head, the dark gray curve along the left side is used to add darker shading providing a three-dimensional feel, and the yellow curves above the eye provide additional shading around the eyes and interpolate smoothly with the textures. The specular highlights on the eyes are defined by vector radial fills.

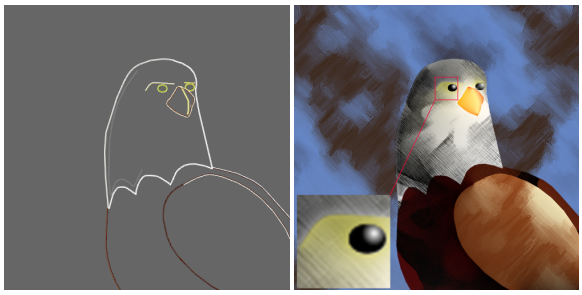


Figure 7: *A cartoon of a bird. Texture shaders were applied to the curves to produce a painted effect. Note how the yellow shading defined on the brow diffuses into the texture to add highlights. The eyes are defined by vector gradient fills.*

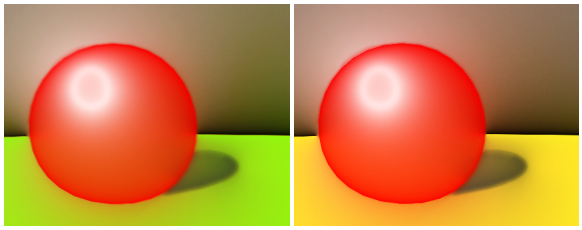


Figure 8: *In this example, the shadow curve and the horizontal curve are defined with transparency. Thus edits to the ground color (from green towards yellow) results in corresponding color changes in the shadowed region and the vertical wall. This makes it easy to preserve the desired image effects without having to edit any other curve.*

Fig. 4 (c) shows an image of a spiral pinwheel. Some curves are defined with a texture shader that shows a detailed leaf texture; other curves are attached with different radial gradient fill shaders. These shaders interact with each other to create the smooth-shaded image. Each radial gradient is white at the center and transforms to a different green color on each curve. To create such a radial gradient, the user only has to specify the center and radius of a circle. In contrast, reproducing such effect using standard diffusion curves would be difficult: a user would have to create several carefully chosen color constraints along the side of each spiral in order to generate a perfectly circular gradient. Using a compositing based technique would also be difficult as it requires carefully defining the transparent mask of each layer.

Weighting functions and transparancy. Fig. 4 shows the effects of applying different distance weighting functions w_d on the curves. As the power p becomes smaller, the effect of distance decreases, leading to a more uniformly shaded image. In comparison, as p becomes larger, the influence of a curve is more concentrated around itself, creating an image with higher contrast. Fig. 5 shows two examples where the curves are assigned transparency values. Note how the effect of transparency softens the colored and textured regions, and in some cases (e.g. the image on the right) mimics the appearance of a translucent object.

(a) Original diffusion curves (b) Our method

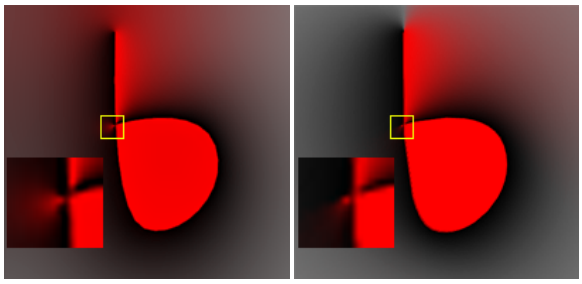
Figure 9: *A comparison of the original diffusion curves and our method on the Zephyr image [OBW*08]. The inlets show zoomed-in comparisons of the details around the eyes. The two images look qualitatively similar. Differences are noticeable at the end points of the curves that define the eyes.*

Fig. 8 shows an example where the transparency feature can be used to define colored shadows and indirect color bleeding effects. Here the shadow curve and the horizontal curve both have transparency turned on. As the user edits the color of the ground, the colors inside the shadow and on the vertical wall change correspondingly due to transparency. This provides a convenient and intuitive way to edit the image. If standard diffusion curves were used, achieving the same effect would have required editing several other curves.

Comparison to [OBW*08]. In Fig. 9, 10 and 11 we show comparisons between the original diffusion curves method [OBW*08] and our method. Fig. 11 is the pipe image we created, and for this example we have removed textures and gradient fills in order to apply their method. While subtle differences are noticeable, the two images look qualitatively the same. The image on the right shows a 2× difference image of the two renderings. The primary differences are around the curve edges.

Fig. 9 is an example taken directly from their paper. Note that the two images look qualitatively similar, but differences are noticeable around the end points of the curves. To examine these differences more carefully, in Fig. 10 we have constructed a simple example consisting of a curve that is shaped like the letter 'b', and one side of the curve is colored red while the other side black. First, we observe that at the top end of the curve, the two methods diffuse colors differently. In (a), the red color is diffused all the way to the left side of the image due to the opening on the top, while our method in (b) generates a sharp boundary between the two sides (as pixels on the left do not receive contribution from the right side of the curve). Second, at the other end point, where the curve slightly crosses itself to the left, we observe that (a) exhibit more pronounced artifacts than (b).

This example simply shows that the two methods behave differently in certain cases, and there is no evidence that either method is inherently better or worse than the other. We believe that since the motivation for these methods is artistic, the Poisson-based approach is simply one among many possible ways for generating smooth shaded images.



(a) Original diffusion curves (b) Our method

Figure 10: *A comparison of the original diffusion curves and our method on the 'b'-shaped curve. At the top end point of the curve, we observe that the colors are diffused in different ways; at the other end point (where the curve slightly crosses itself), we observe that the original diffusion curves generate more pronounced artifacts than our method.*

Performance. As described in Section 3, our algorithm switches between a fast adaptive sampling mode during interaction and a high quality full frame rendering mode once the user has been inactive for a second. Using an NVIDIA GTX470 GPU, we achieve 14-25 fps for the adaptive sampling mode, and 2-3 fps for the high quality mode for all the images included in the paper. We found the speed to be sufficient for interaction and is relatively insensitive to the number of curves. This insensitivity is a result of the fact that as the number of curves increases the rays tend to be terminated more quickly during ray tracing using a uniform grid structure. Note that enabling transparency is more expensive because each ray must continue traveling after intersecting a curve. With global transparency turned on, the spiral image in Figure 4 took about 2 seconds to render in high-quality mode. All images included in the paper were made with between 15 and 50 curves, resulting in between 300 to 1000 line segments. All high-quality full frame renderings were computed within 300∼500 ms per frame.

Noise and Sampling Artifacts. Since our method uses stochastic ray tracing, an insufficient number of rays can lead to numerical issues such as noise and sampling artifacts in the rendering. This will become especially noticeable during an animation, where the sampling artifacts lead to temporal flickering and aliasing. So far by using 128 rays per pixel we haven't noticed any obvious sampling artifacts in the high-quality rendered images. Furthermore, our current implementation pre-generates the set of rays for every pixel using pseudo-random numbers, and they remain the same through successive frames. Therefore, during an animation when the user edits curves locally, the pixels that are not influenced by the edited curves will return the same values as the previous frame. Hence in practice the animations we

(a) Original diffusion curves (b) Our method (c) 2× difference

Figure 11: *A comparison between images generated using [OBW*08] (left) and our method (middle); and a 2× difference image is shown on the right. For this comparison we used the pipe image from Figure 2 but with the textures and gradient fills removed. Note how the two images look qualitatively the same. The primary differences are on the edges of the curves.*

generated using the high-quality renderer appear stable and are not prone to temporal artifacts. Refer to the supplemental video for an example animation.

5. Conclusions and Future Work

To summarize, we have presented a raytracing solution to diffusion curves that provides the following benefits: 1) it allows a unified treatment of diffusion curves with classic vector and raster graphics by using shaders; 2) it is easy to extend with additional curve attributes such as weighting functions and transparency; 3) it provides interactive frame rates using a GPU-based implementation.

In future work, we would like to extend our method to diffusion surfaces. It is also possible to extend our method to render surface details defined by textures, similar to [JCW09b]. In fact, the 'local' computation nature of our method makes it suitable for many view-dependent rendering scenarios. One limitation of our work is that it is currently unclear how a ray tracing based approach can achieve anisotropic diffusion effects as presented in [BEDT10]. We would like to investigate possible ways to extend our method along this direction. Finally, our GPU implementation is not highly optimized, and there is opportunity for significant improvement towards real-time framerates.

Acknowledgments. We would like to thank Copper Giloth, Zijun Guo, and Fred Zinn for discussions and feedback on the project. This work was supported by NSF grant CCF-0746577 and an NSF graduate student fellowship.

References

[Aga07] AGARWALA A.: Efficient gradient-domain compositing using quadtrees. *ACM Trans. Graph. 26* (2007). 3

[BEDT10] BEZERRA H., EISEMANN E., DECARLO D., THOLLOT J.: Diffusion constraints for vector graphics. In *Proc. of NPAR* (2010), pp. 35–42. 1, 3, 4, 8

[BFGS03] BOLZ J., FARMER I., GRINSPUN E., SCHRÖODER P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph. 22* (2003), 917–924. 3

[FHL*09] FARBMAN Z., HOFFER G., LIPMAN Y., COHEN-OR D., LISCHINSKI D.: Coordinates for instant image cloning. *ACM Trans. Graph. 28* (2009), 67:1–67:9. 2, 3

[Flo03] FLOATER M. S.: Mean value coordinates. *Comput. Aided Geom. Des. 20* (2003), 19–27. 2, 3, 4

[FLW02] FATTAL R., LISCHINSKI D., WERMAN M.: Gradient domain high dynamic range compression. *ACM Trans. Graph. 21* (2002), 249–256. 3

[HB10] HOBEROCK J., BELL N.: Thrust: A parallel template library, 2010. 5

[JCW09a] JESCHKE S., CLINE D., WONKA P.: A GPU Laplacian solver for diffusion curves and Poisson image editing. *ACM Trans. Graph. 28* (2009), 116:1–116:8. 1, 3

[JCW09b] JESCHKE S., CLINE D., WONKA P.: Rendering surface details with diffusion curves. *ACM Trans. Graph. 28* (2009), 117:1–117:8. 1, 3, 8

[JCW11] JESCHKE S., CLINE D., WONKA P.: Estimating color and texture parameters for vector graphics. *Computer Graphics Forum 2*, 30 (2011), to appear. 1, 3

[JSW05] JU T., SCHAEFER S., WARREN J.: Mean value coordinates for closed triangular meshes. *ACM Trans. Graph. 24* (2005), 561–566. 3, 4

[Kaj86] KAJIYA J. T.: The rendering equation. *SIGGRAPH Comput. Graph. 20* (1986), 143–150. 4

[KH08] KAZHDAN M., HOPPE H.: Streaming multigrid for gradient-domain operations on large images. *ACM Trans. Graph. 27* (2008), 21:1–21:10. 3

[LKCOL07] LIPMAN Y., KOPF J., COHEN-OR D., LEVIN D.: GPU-assisted positive mean value coordinates for mesh deformations. In *Proc. of SGP* (2007), pp. 117–123. 3

[MP08] MCCANN J., POLLARD N. S.: Real-time gradient-domain painting. *ACM Trans. Graph. 27* (2008), 93:1–93:7. 3

[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. 27* (2008), 92:1–92:8. 1, 2, 7, 8

[PGB03] PÉREZ P., GANGNET M., BLAKE A.: Poisson image editing. *ACM Trans. Graph. 22* (2003), 313–318. 3

[SJTS04] SUN J., JIA J., TANG C.-K., SHUM H.-Y.: Poisson matting. *ACM Trans. Graph. 23* (2004), 315–321. 3

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

[TSNI10] TAKAYAMA K., SORKINE O., NEALEN A., IGARASHI T.: Volumetric modeling with diffusion surfaces. *ACM Trans. Graph. 29* (2010), 180:1–180:8. 1, 2, 3

[WOBT09] WINNEMÖLLER H., ORZAN A., BOISSIEUX L., THOLLOT J.: Texture design and draping in 2d images. *Computer Graphics Forum 28*, 4 (2009), 1091–1099. 1, 3