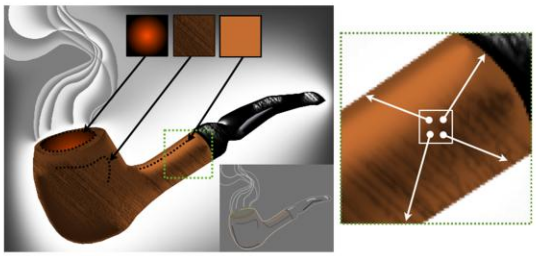
UMassAmherst

A Ray Tracing Approach to Diffusion Curves



John C. Bowers, Jonathan Leahey, and Rui Wang

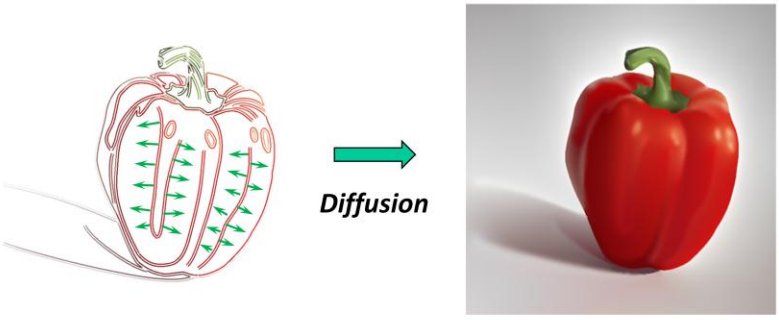
Univ. of Massachusetts Amherst

This work is about a new method for generating diffusion curve style images. Although this topic is dealing with non-photorealistic rendering, as you will see our underlying solution is based on two-dimensional ray tracing on the GPU, and it is analogous to computing final gathering in global illumination problems.

UMassAmherst

Introduction

- **Diffusion Curves** [OBW*08]
  - A vector representation that allows for easy creation of smooth-shaded images.



2

Diffusion curves is a vector-based image representation that can be used to easily create smooth-shaded images. Here, each curve is defined with generally different colors on each side; given a set of such curves, an image can be computed by simulating the diffusion of the curve colors into the interior of the image. Due to the diffusion process, the regions between curves are typically very smooth; and the curves themselves serve as diffusion boundaries which can be used to define edges, highlights, or other image features.

UMassAmherst

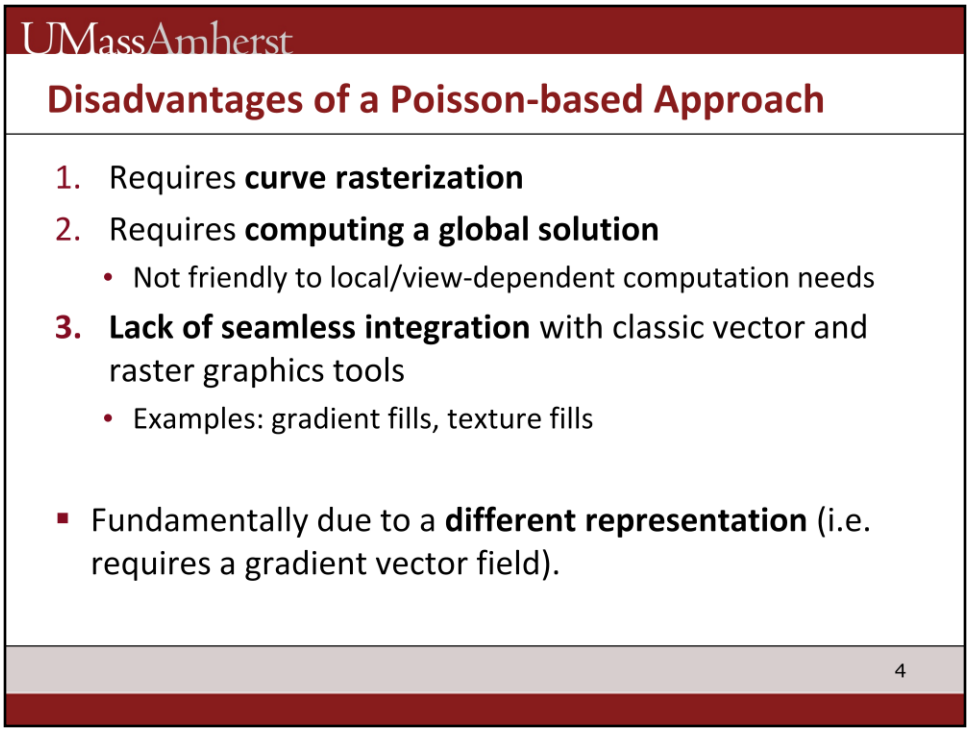
Introduction

- The standard way to compute a diffusion curves image is by **solving a Poisson equation**.

$$\Delta f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$$

- **Recent work on diffusion curves**
  - Fast Poisson solver [JCW09a]
  - Define surface details [JCW09b]
  - Expressive control via *diffusion constraints* [BEDT10]
  - Texture draping [WOBT09]

The standard way to compute a solution image is by solving a Poisson equation, more specifically a Laplace equation, which simulates the diffusion process. The way this works is that it solves for an image f whose Laplacian is zero everywhere except on the curve boundaries, where the Laplacian is equal to the divergence of the color gradient. Because diffusion curves provide an easy way to generate and manipulate smooth images, it has inspired many recent work which try to improve its speed and functionality.



However, the underlying solution in these works is still based on solving the Poisson equation, and this type of approach has several limitations. First, it requires rasterizing the curves into pixels that will be used as the source of diffusion. But curve rasterization is prone to aliasing artifacts. Second, the Poisson solver requires computing a global solution, which means all image pixels have to be solved at once. This is not friendly to local or view-dependent computation needs. Finally, the Poisson-based approach also lacks seamless integration with classic vector and raster graphics tools, such as gradient fills and texture fills. These are the tools that artists are already familiar with. But in order to use them in a diffusion curves framework, they would either have to be redefined or created on a different layer and composited afterwards. And this issue is because diffusion curves require a gradient vector field, which is fundamentally different from how the other tools are represented.

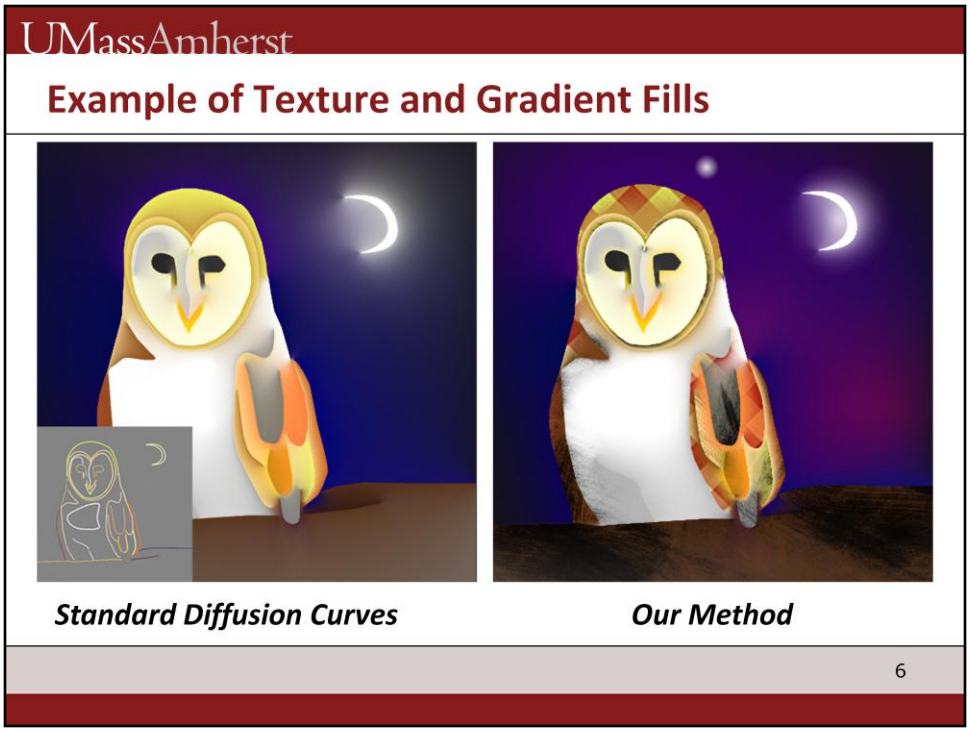
UMassAmherst

Our Contributions

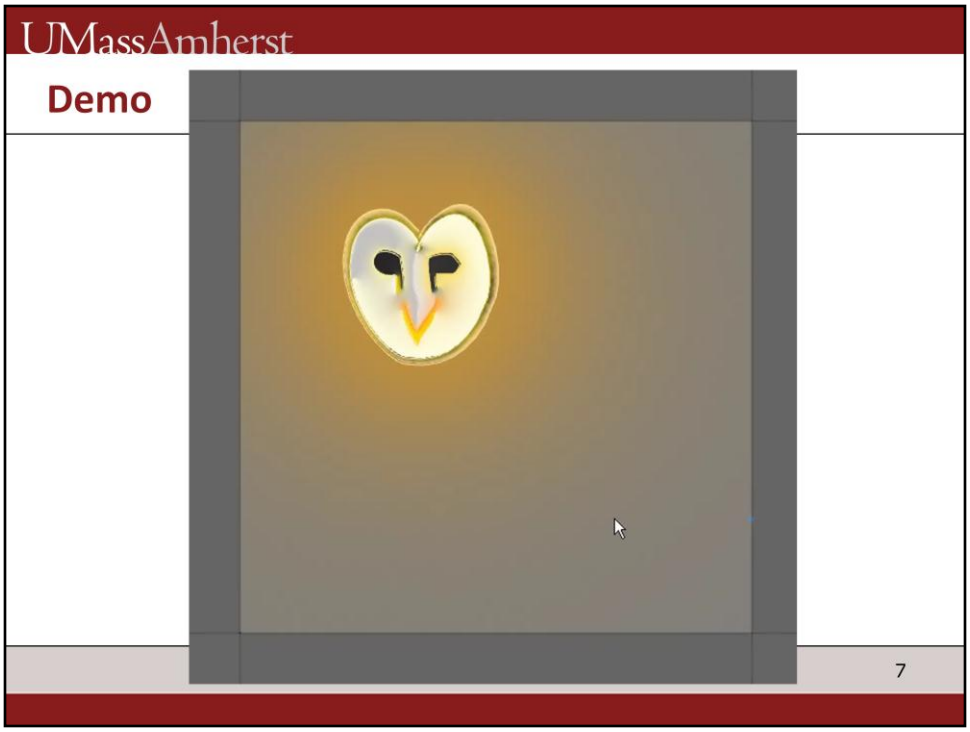
- A new, raytracing based solution to diffusion curves with the following advantages:
  - Requires **no curve rasterization**.
  - Can be computed **independently for any pixel.**
  - **Seamless integration** of diffusion curves with classic vector and raster graphics tools, such as ***gradient fills*** and ***textures fills***.

5

Our main contribution is a raytracing based approach that is aimed to address the above limitations. First, it is based on ray tracing analytic curves and hence it does not require curve rasterization. Second, it can be computed independently for any set of pixels, so it's very suitable for local or view-dependent computation needs. And finally, it seamlessly integrate diffusion curves with classic vector and raster graphics tools through the use of shaders, which I will describe shortly. This makes it easy to create different image effects all in a single layer using familiar operations.



This example demonstrates the effect of textures and gradient fills. In the left image, curves are defined only with colors, and this is what standard diffusion curves would output. The right image is our results that contains more interesting details, such as the textures on the owl, and the radial gradient fills on the glow of the moon and also the sky background.



This video shows the sketching session that created the owl image. The video has been sped up so you can see the entire process. The user starts by drawing the curves and assigning colors as in standard diffusion curves; then textures are attached to the curves to define the rendering details; and finally radial gradient fills are added to simulate the glow of the moon as well as on the background sky.

4:30 here

UMassAmherst

Related Work

- **Mean-Value Coordinates** [Flo03, JSW05]
  - **Coordinates Image Cloning** [FHL*09]

Solve Laplace equation: $\Delta f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$

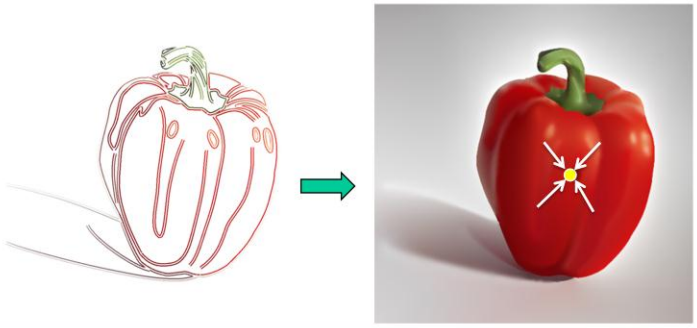
Mean-value interpolation: $f(x) = \sum_i f(\Omega_i) \cdot w_{mvc}(x, i)$

So what inspired this work is the idea that the solution to the Laplace equation can be converted into a much simpler and direct form. This idea was first introduced in a paper by Farbman et al. in the context of image cloning. They made a clever observation that the solution to the Laplace equation can be approximated by an interpolation from the boundary values using the mean-value coordinates, and this is a much more direct way to compute the result than solving a differential equation.

UMassAmherst

Related Work

- **Mean-Value Coordinates** [Flo03, JSW05]
  - **Coordinates Image Cloning** [FHL*09]
- We apply this idea to diffusion curves:

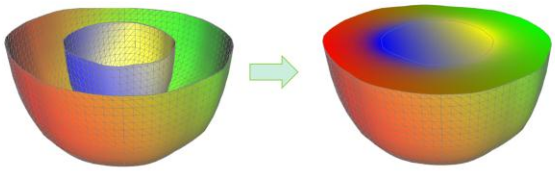


This method can be extended to diffusion curves and is the key of our work. And here is an intuitive explanation: to compute the color of this pixel >>>> in the resulting image, instead of solving a Poisson equation, we can directly interpolate it from the colors of nearby, unoccluded curves.

UMassAmherst

Related Work

- **Mean-Value Coordinates** [Flo03, JSW05]
  - **Coordinates Image Cloning** [FHL*09]
  - **Diffusion Surfaces** [TSNI10]
    - Compute interpolated colors by using rasterization.



In fact, this idea has been applied in the context of diffusion surfaces, which is an extension of diffusion curves to smooth shaded 3D textures. The biggest difference in their approach with ours is that they use rasterization instead of raytracing to solve the problem. Specifically, they project these colored surfaces to a rasterization buffer at each shading pixel, from which they can then compute the interpolated color.

UMassAmherst

Related Work

- **Mean-Value Coordinates** [Flo03, JSW05]
  - **Coordinates Image Cloning** [FHL*09]
  - **Diffusion Surfaces** [TSNI10]
    - Compute interpolated colors by using rasterization.
  - ***Advantages of our method over rasterization:***
    - Performance benefits of raytracing.
    - Allows for arbitrary curve attributes including transparency.
    - Easy to achieve anti-aliasing by spatial stochastic sampling.

You may be wondering, ok, rasterization sounds like a faster approach. But it's actually not the case as we've tested. It turns out that for diffusion curves, using ray tracing provides several clear benefits. The first is the performance, which sounds a bit counter-intuitive. But what's happening here is that the number of curve segments is much larger than the resolution of the rasterization buffer, and therefore this is a case where raytracing will win in terms of speed. The second benefit is that using raytracing allows for arbitrary curve attributes such as transparency, which will be much more difficult to achieve with rasterization. And finally raytracing makes it easy to achieve image anti-aliasing by using spatial stochastic sampling, while using rasterization cannot get this benefit.

UMassAmherst

Outline

- Raytracing Formulation
- Definition of Shaders
- Curve Attributes
- Implementation Details
- Results

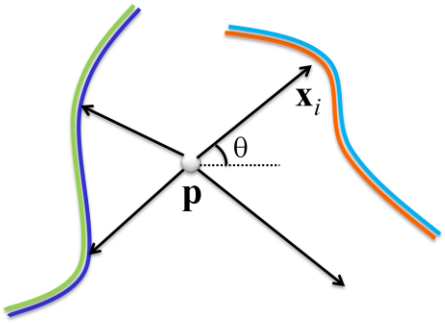
7:30 here

UMassAmherst

Algorithm Overview

- **Raytracing Formation**

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



$\mathbf{p}$:	shading pixel
$\mathbf{x}_i$:	visible curve point in direction θ
$\boldsymbol{L}$:	curve color
$\boldsymbol{w}$:	normalized weight

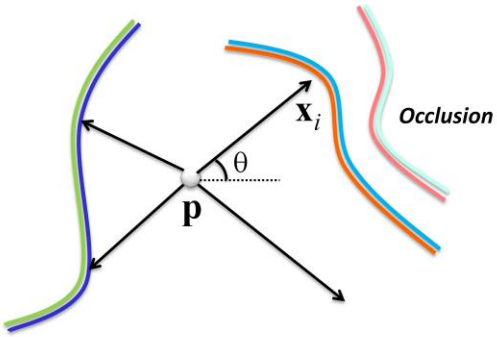
To begin, let's assume that a curve is defined with colors on each side. Given a set of such curves, we define the color of a shading pixel p as the directional integral over the visible curves. Specifically, imagine we shoot a ray >>>> from p towards the direction theta, and this ray's closest intersection point is xi. We then obtain the color of xi along the curve, and multiply it with a normalized weight to define the contribution of this ray. Finally, >>>>> we integrate the contributions of all rays from 0 to 2pi, and the result gives the color of p. As you can see, this definition is very much like treating the curves as light sources that emit radiance, and the integration is performed in the same fashion as final gathering, which can be evaluated using stochastic ray tracing.

UMassAmherst

Algorithm Overview

- **Raytracing Formation**

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



$\mathbf{p}$:	shading pixel
$\mathbf{x}_i$:	visible curve point in direction θ
L:	curve color
w:	normalized weight

Since we use ray tracing, the computation naturally accounts for occlusion. This way, the influence of a curve will not cross the boundary of other curves, which is what we have wanted.

UMassAmherst

Algorithm Overview

- **Raytracing Formation**

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

  - The weighting function $\boldsymbol{w}$ defines how the curve color influences its nearby space.
    - In a **special case**, $\boldsymbol{I}(\mathbf{p})$ can be analytically computed using **mean-value coordinates** [Flo03, JSW05]
    - In the general case, it must be numerically computed.

The weighting function $\boldsymbol{w}$ defines how the curve will influence its nearby space. It typically has to do with the distance between the intersection point and the shading pixel, but can include other factors as well. In a very special case, the integral can actually be analytically computed using mean-value interpolation. The special case has to satisfy three conditions: first, the weighting function is inversely proportional to distance; second, occlusion between curves must be ignored; third, colors are linearly interpolated along the curve. In the general case, however, the integral has no analytic solution and must be numerically computed.

UMassAmherst

Algorithm Overview

- **Raytracing Formation**

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

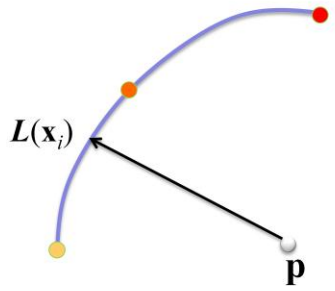
- **Shaders**
  - We can now extend the curve color $\boldsymbol{L}$ to a more general concept called curve **Shaders**.
    - ***Color shader***
    - ***Gradient fill shader***
    - ***Texture shader***

Given the basic raytracing formulation, we can now extend the color L to a more general concept called shaders. I will describe three types of shaders.

UMassAmherst

Shaders

- **Color Shader**
  - Colors are defined at the curve vertices.
  - The intersection point's color is linearly interpolated from the two nearby vertices.

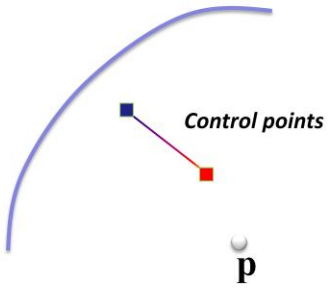


The first is a color shader, which is equivalent to standard diffusion curves. Here a color is defined at each vertex, and when a ray hits the curve, the color of the hit point is linearly interpolated from the two nearby vertices.

UMassAmherst

Shaders

- **Gradient Fill Shader**
  - Examples: **linear**, **radial**, and **angular** fills.
  - The user specifies two control points with colors, and the type of fill operations.

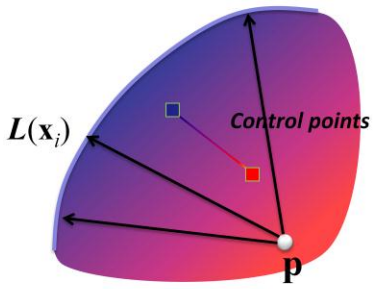


The second shader is the gradient fill shader. Examples include linear, radial, or angular fills. To define such a shader, the user specifies two control points, each with a different color, and also specifies the type of gradient fill operation.

UMassAmherst

Shaders

- **Gradient Fill Shader**
  - The intersection point's color is calculated based on the location of the pixel **p** relative to the two control points.

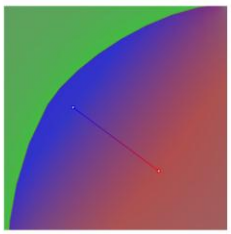


When a ray hits the curve, the color of the intersection point is calculated based on the location of the pixel p relative to the two control points. Notice that this is different from the previous shader in that the color is not defined along the curve, rather, it is decided by the two control points. As a result, >>>> any ray starting from the same pixel and hitting the same curve will return the same color. However, the curve >>>>> does serve as a boundary to confine the influence of the gradient fill to only one side of the curve.

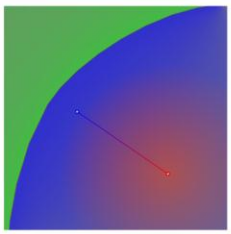
UMassAmherst

Shaders

- **Gradient Fill Shader**
  - Examples:



Linear gradient



Radial gradient

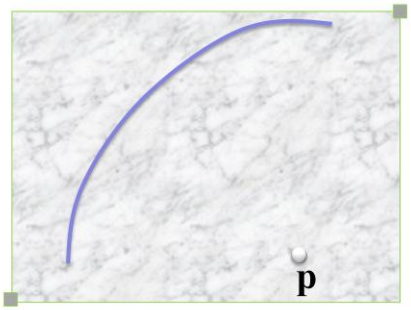
  - Other gradient fill operators can be easily included as well, including procedurally defined ones.

Here are two examples of the gradient fill shader. They are defined by the same two control points but one is a linear gradient fill and the other is a radial gradient fill. Other gradient fill operators can be included as well, such as procedually defined operators.

UMassAmherst

Shaders

- **Texture Shader**
  - Allows a texture image to be attached to a curve.
  - The user specifies two anchor points of the texture.

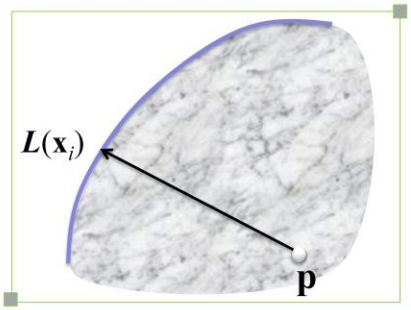


The third shader is the texture shader. It is defined in a similar way with gradient fill shader. Specifically, the user provides two anchor points that define the texture region.

UMassAmherst

Shaders

- **Texture Shader**
  - The intersection point's color is computed by a texture lookup using **p**'s coordinates relative to the anchor points.

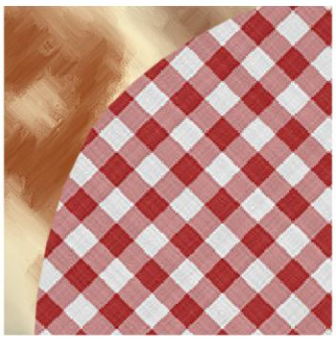


When a ray hits the curve, the color of the intersection point is returned by a lookup in the texture using p's coordinates relative to the two anchor points. This will give a shading result like this >>>>

UMassAmherst

Shaders

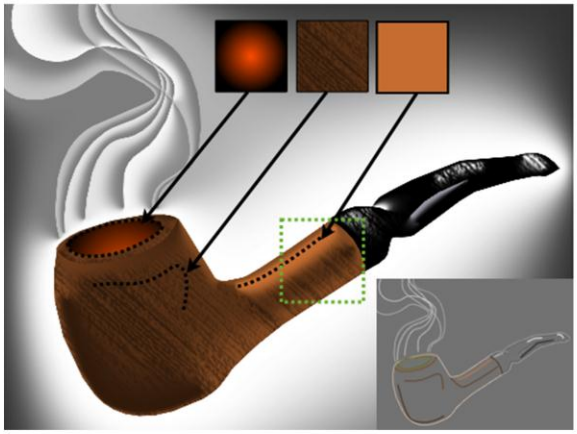
- **Texture Shader**
  - Example:



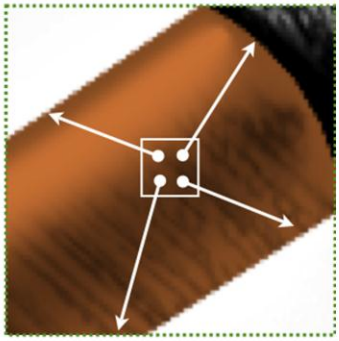
Here is another example. Here, each side of the curve is attached with a different texture shader.

UMassAmherst

Shaders



An image created with all three shaders



Anti-aliasing

Now, the power of our approach is that it allows all three types of shaders to appear in a single layer and intersect with each other through the weighted integral. Here we show a pipe image that's defined using all three shaders. For instance, the glow in the pipe is defined with a radial gradient fill shader, the pipe's body is defined with texture shaders, and the highlights are defined with color shaders.

Antialiasing??

UMassAmherst

Curve Attributes

- **Raytracing Formation**

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

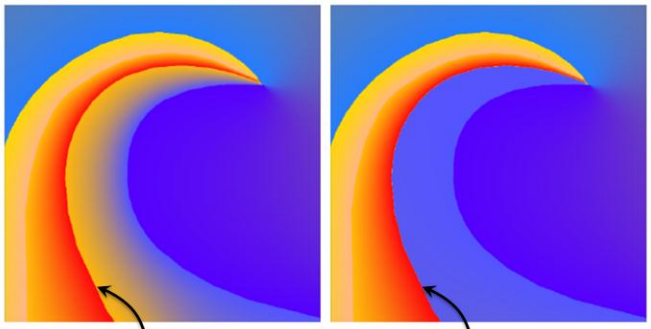
- The weighting function is the multiplication of the following two weights:
  - ***Relative curve weight***
  - ***Distance-based weight***

Now let me describe the weighting function w, which can be used to customize the influence of each curve. The weighting function is currently defined as the product of two weights.

UMassAmherst

Curve Attributes

- **Relative Curve Weight** w_c



$w_c = 1$ $w_c = 0$ ***Barrier curve***

The first is a relative curve weight wc, controls the influence of a curve relative to other curves. By default, every curve is given a weight of 1, so that they all count equally. By changing the weight of a curve to zero, we are essentially ignoring this curve and its nearby region is completely determined by other curves. This is equivalent to what's known as a barrier curve in previous work.

UMassAmherst

Curve Attributes

- **Relative Curve Weight**

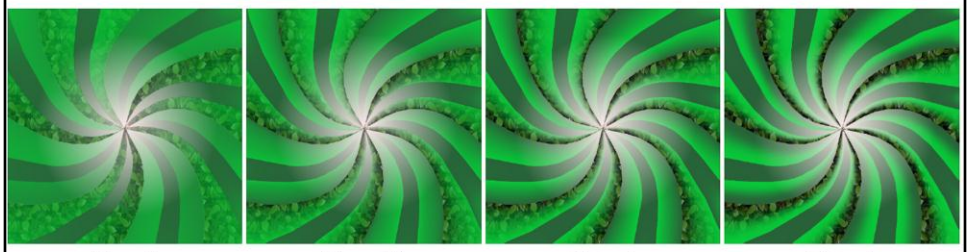
This video clip shows adjusting the weight of the golden curve from 1 to 0 and then back.

UMassAmherst

Curve Attributes

- **Distance-based Weight** w_d

$$w_d(\mathbf{x}_i, \mathbf{p}) = |\mathbf{x}_i - \mathbf{p}|^{-p}$$



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

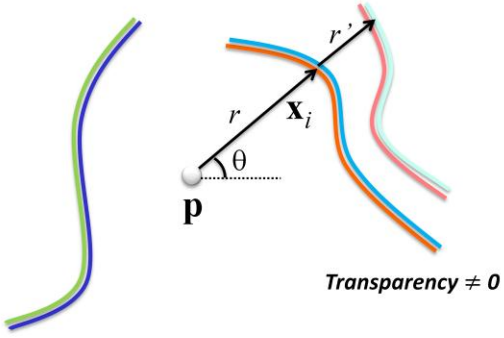
The second component of the weighting function is a distance-based weight, which is defined as the negative p-th power of the distance between an intersection point with the shading pixel. This basically controls how quickly the influence of a curve will fall off over distance. So if p is a small value, the influence of a curve can go very far; and if p is large, we can see that the influence of a curve will be increasingly more concentrated around itself.

UMassAmherst

Curve Attributes

- **Curve Transparency** α_c

$$I(r) = (1 - \alpha_c) \cdot I(\mathbf{x}_i) + \alpha_c \cdot I(r')$$

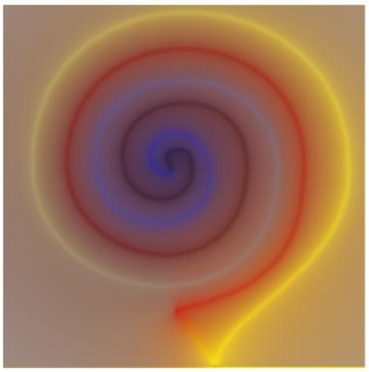


The last attribute that I will introduce is the curve transparency alpha. This is basically a concept that's borrowed from the rendering literature, and it allows the influence of a curve to penetrate through other curves. This effect can be achieved by simply allowing the ray to continue >>>>> traveling after the first intersection point, and the final color of the ray is calculated by an alpha blending of all the successive intersection points.

UMassAmherst

Curve Transparency

- **Curve Transparency Examples**

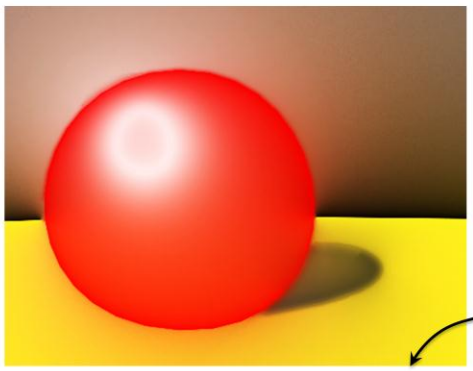


Here is an example that shows the effect of transparency. Initially the curve transparency is disabled. By turning it on and giving all curves a transparency value of 0.5, we can see that the colors are significantly diffused, giving a sense of translucency effect.

UMassAmherst

Curve Transparency

- Simulate ***colored shadows*** and ***color bleedings*** effects.



In practice, we can use transparency to some extent to simulate colored shadows and color bleeding effects in an image. For example, this image contains a shadowed region and a back wall, and we have assigned the curves defining these two regions a nonzero transparency value. Now, when the user edits the color of the floor from green towards >>>> yellow, the colors inside the shadow curve and the back wall will change accordingly, which simulate color shadows and color bleeding effects. This provides a convenient way to modify the image, without having to edit multiple curves.

UMassAmherst

Implementation Details

- Implemented on the GPU using CUDA and Thrust.
- All curves are defined as cubic Bézier curves.
- All images are rendered at 512x512 resolution.

Now let me briefly go through implementation details. The entire algorithm is implemented on the GPU using CUDA and Thrust library. All curves are defiend as cubic Beizer curves, and images are rendered at 512x512 resolution by default.

UMassAmherst

Implementation Details

- ***Step 1***: subdivide curves into line segments (N).
- ***Step 2***: build a uniform grid of $2\sqrt{N} \times 2\sqrt{N} = 4N$ resolution.
- ***Step 3***: classify line segments to the grid.
- ***Step 4***: for each pixel, trace 128 stratified rays; the ray origin is assigned using 2x2 sub-pixel jittered pattern.
- N is generally between 300 to 1000 in the examples shown.

The algorithm mainly contains 4 steps. The first step is to subdivide curves into a set of line segments using a standard subdivision algorithm. This is mainly for the simplicity of tracing line segments. Alternatively we can also trace cubic curves directly and avoid subdividing the curves. Assume there are N line segments, the second step is to build a uniform grid of 4 times N resolution; then step 3 is to classify line segments to the grid, and finally, we use the uniform grid to accelerate ray tracing. Specifically, for each pixel we trace 128 rays where the directions are randomly sampled with stratification, and the origins are assigned using a 2x2 jittered pattern to provide spatial antialiasing.

In our experiments, the number of line segments N is typically between 300 to 1000.

UMassAmherst

Image Adaptive Sampling

- Simple 2-stage adaptive sampling to exploit spatial coherence.
- ***Stage 1***: sparse sampling on 8x8 pixel block, with 64 rays per pixel; record the shortest intersection distance.
- ***Stage 2***: if a block is sufficiently close to any curve, all pixels in the block will be sampled; otherwise, the pixels will be bilinearly interpolated from the corner pixels.

Now, because a diffusion curves image is typically smooth in the interior, we can exploit the spatial coherence to speed up the computation during user interaction. This is achieved by using a simple 2-stage adaptive sampling algorithm. In the first stage, we perform sparse sampling on every 8x8 pixel block, and we trace a reduced number of 64 rays per pixel. For each pixel we calculate its color and record the shortest intersection distance. Then in stage 2, if a block is found to be sufficiently close to any curve, which we can tell from the shortest distances of the corner pixels, then every pixel in the block will be sampled. Otherwise, the pixels in a block will be linearly interpolated from the corner pixels.

This video shows the adaptive samples generated using the 2-stage algorithm. As you can see, the sampled pixels are mostly distributed nearby the curves, and the rest of the image contains only sparsely sampled pixels.

UMassAmherst

Image Adaptive Sampling

- On NVIDIA 470 GTX, the adaptive algorithm can perform at **15~25 fps** with acceptable quality.
- As soon as the user is inactive for one second, full frame rendering will be performed at every pixel with 128 rays.
  - This provides high quality results, at **2~3 fps.**

In terms of performance, on an NVIDIA 470 GTX, our adaptive sampling algorithm can perform at 15-25 fps with acceptable quality. As soon as the user is inactive for more than a second, the algorithm will start a full frame rendering which will compute at every pixel in the 512 square image, and with 128 rays. This provides high-quality rendering result at 2-3 fps.

UMassAmherst

Results – Shaders

Now I will show some additional results. This is a pinwheel image that I have shown before. These curves are defined with various shaders including texture shader, color shader, and a radial gradient fill shader which defines the center glow. Reproducing this image using standard diffusion curves would be very difficult, especially center glow because it would require carefully assigning the colors of multiple curves. In comparison, our method only requires one single radial gradient fill defined with two control points.

Results

UMassAmherst

Results – Comparison to [OBW*08]

- Pipe image

*[OBW*08]*

Our method

2x error

39

Here we compare the pipe image rendered using our method with the result obtained using the original diffusion curves. To make this comparison, we have removed the texture fills and gradient fills since their method does not support these features. The two images match quite well qualitatively, and the main differences are on the curve boundaries.

UMassAmherst

Results – Comparison to [OBW*08]

- Zephyr image

*[OBW*08]*

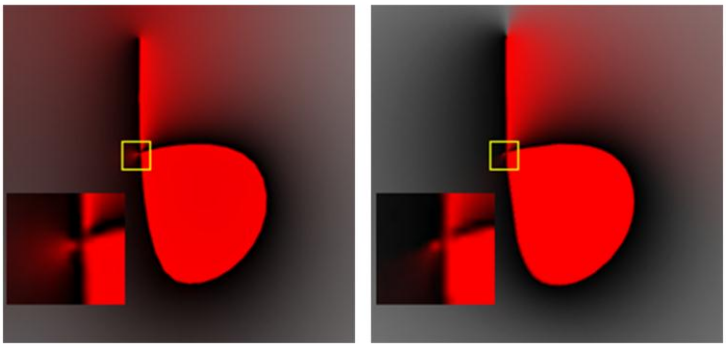
Our method

This is another comparison where the input is the Zephyr image from their paper. Again, the images look qualitatively similar, but on close examination, we can see some differences in the way that colors are diffused into each other. In particular, the regions around the eyes contain open curves which result in noticeable differences.

UMassAmherst

Results – Comparison to [OBW*08]

- The two methods spread colors differently at the ***open ends of a curve***.



*[OBW*08]* *Our method*

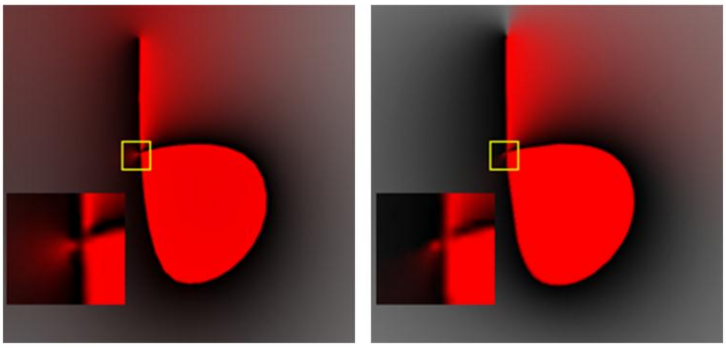
41

To study these differences more carefully, we have constructed a simple example with a curve shaped like the letter b. The first difference we can see clearly is that at the top end of the curve, the two methods spread colors differently. Specifically, the Poisson-based approach diffuses the red color all the way to the left side due to the opening at the top. On the other hand, our method creates a sharp boundary between the two sides due to the visibility of the curve. The second difference is that at the other end of the curve, where the curve slightly crosses itself, the Poisson based approach produces more pronounced artifacts than our method, as shown in the blow-up view.

UMassAmherst

Results – Comparison to [OBW*08]

- The two methods spread colors differently at the ***open ends of a curve***.



*[OBW*08]* *Our method*

Although these differences exist, there is no clear evidence that either method is inherently better or worse than the other. Sometimes you may want the behavior of one method, and sometimes the other. These are simply different options that artists can exploit and get adapted to.

UMassAmherst

Summary

- **A raytracing solution to diffusion curves with the following benefits:**
  - Unified treatment of diffusion curves with classic vector and raster graphics tools.
  - Simple to implement and easy to extend with custom attributes.
  - Suitable for parallel computation.

To summarize,

Future Work

- **Define view-dependent surface details** [JCW09b]
  - The local computation nature of the algorithm is suitable for view-dependent effects.
- **Optimize raytracing** to improve speed.
- Simulate **anisotropic diffusion effects**.

Acknowledgement

- Copper Giloth, Zijun Guo, Fred Zinn
- NSF Grant CCF-0746577
- NSF Graduate Student Fellowship