A GPU Laplacian Solver for Diffusion Curves and Poisson Image Editing

Stefan Jeschke*
Arizona State University

David Cline†
Arizona State University

Peter Wonka‡
Arizona State University

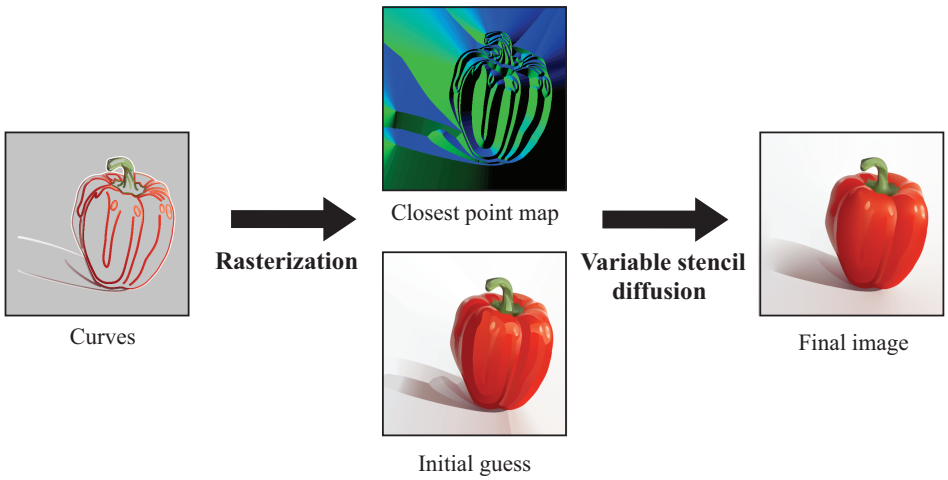

Figure 1: *Diffusion curve rendering in our system. Analytical curves (left) are rasterized into a closest point map (distance map plus information about the closest curve point) and an initial guess image (middle). The initial guess is diffused by our variable stencil size solver, producing the final image (right).*

Abstract

We present a new Laplacian solver for *minimal* surfaces—surfaces having a mean curvature of zero everywhere except at some fixed (Dirichlet) boundary conditions. Our solution has two main contributions: First, we provide a robust rasterization technique to transform continuous boundary values (diffusion curves) to a discrete domain. Second, we define a *variable stencil size* diffusion solver that solves the minimal surface problem. We prove that the solver converges to the right solution, and demonstrate that it is at least as fast as commonly proposed multigrid solvers, but much simpler to implement. It also works for arbitrary image resolutions, as well as 8 bit data. We show examples of robust diffusion curve rendering where our curve rasterization and diffusion solver eliminate the strobing artifacts present in previous methods. We also show results for real-time seamless cloning and stitching of large image panoramas.

CR Categories: I.3.3 [COMPUTER GRAPHICS]: Picture/Image Generation —Display algorithms; I.4.10 [IMAGE PROCESSING AND COMPUTER VISION]: Image Representation—Multidimensional

Keywords: Line and Curve rendering, Diffusion, Poisson equation

1 Introduction

A *minimal surface*, also known as a "rubber sheet", is a function that has zero mean curvature everywhere, except at a few fixed points, called *Dirichlet* boundary conditions. Given a set of boundary points, the corresponding minimal surface can be found by solving the equation which minimizes the Laplacian ($\nabla^2 G = 0$) of the solution while maintaining the boundary values. This equation shows up repeatedly in engineering contexts, and is referred to variously as the *homogenous Poisson equation*, the *Laplace equation*, the *heat equation* or the *diffusion equation*. We will use these terms interchangeably in the paper.

One application of minimal surfaces in computer graphics is the recently introduced Diffusion Curve Image (DCI) [Orzan et al. 2008] representation. A DCI is defined by a set of curves with colors on either side that are meant to be viewed as a rubber sheet; i.e. the colors diffuse out from the curves filling the plane. Thus, the fundamental rendering operation is solving the diffusion equation. Diffusion curves provide an intuitive and efficient method for image manipulation, and are at the same time general in the sense that a wide variety of images can be generated. However, proper treatment of diffusion curve rendering remains an unsolved problem. Existing multigrid solutions can lead to noticeable artifacts, as seen in figure 2 (left) and the accompanying video.

A second important application of minimal surfaces is *seamless image cloning* [Pérez et al. 2003]. Seamless cloning allows a source image patch to be pasted into a destination image without visible

*e-mail:jeschke@cg.tuwien.ac.at
†e-mail:clinedav@gmail.com
‡e-mail:pwonka@gmail.com

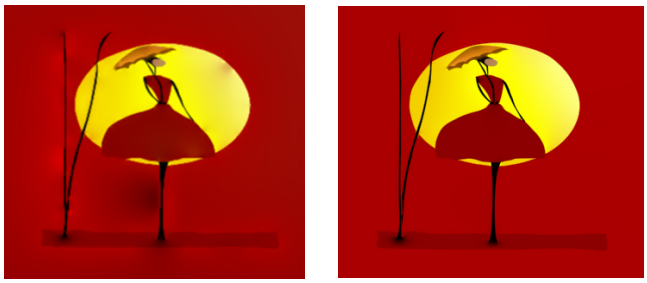

Figure 2: *Comparison of the rendering of Orzan et al. (left) to our rendering (right). Note the color bleeding at small features in the left image. These artifacts are even more prominent in animations, where they result in visible strobing (see accompanying video).*

seams. The process works by exchanging the source and destination rubber sheets within the patch. As with diffusion curves, the rubber sheet difference between the source and destination is a solution to the homogenous Poisson equation, which in the original work was solved using either Gauss-Seidel iterations with successive overrelaxation, or a multigrid solver.

Contributions. This paper provides two important contributions that make diffusion curve rendering faster and more robust, and allow for real-time seamless cloning of large image patches.

The first contribution is a rasterization technique for diffusion curve rendering that provides high quality for interactive applications. A major challenge in the diffusion curve rendering process is that the nicely defined analytical curves have to be discretized in a regular grid to provide the input for the diffusion solver. Due to the global nature of the diffusion process, even small errors in the initial setup can lead to noticeable errors affecting thousands of pixels. Additionally, the location of the curves relative to image pixels may change during every frame in an interactive application, for example, if the image is viewed at multiple resolutions, or if the curves are edited. Our solution gets rid of these fragile dependencies by rasterizing the Voronoi diagram of the curves as an initial state, coloring each pixel with the color of the closest curve point before diffusion begins.

The second contribution of the paper is to define a *variable stencil size* diffusion solver that quickly computes the minimal surface for any initial conditions within a few iterations. Typically 8 iterations suffice to produce visually smooth output. The new solver works by performing Jacobi-like iterations, except with a larger stencil size than usual diffusion. This simple modification has a profound effect on the convergence rate of the solver, making it faster than existing multigrid solutions for computing minimal surfaces. Furthermore, the output of our variable stencil size solver is stable under subpixel translation and general curve animations.

2 Background

The Poisson equation and minimal surfaces. The Poisson equation is a very successful model of a number of different problems in science and engineering. In computer graphics, Perez et al. [2003] were the first to provide a rigorous treatment of the Poisson equation as an image editing tool. A number of different methods have been suggested to solve the Poisson equation. Perez et al. suggested the use of either a conjugate gradient or a multigrid solver, with the multigrid method being the more efficient of the two. Later solutions have also mostly relied on the multigrid method [McCann and Pollard 2008], [Grady et al. 2005], [Ian et al. 2003], [Goodnight et al. 2003], [Kazhdan and Hoppe 2008], [Kimmel and Yavneh 2002], [Grady 2008].

Several applications allow a reformulation to a *minimal surface problem* instead of the general Poisson equation. This reformulation is an instance of the *homogeneous* Poisson equation (or Laplace equation), meaning that the value of the Laplacian is zero everywhere except at some Dirichlet boundary conditions, or in other words, fixed values at specified pixel locations. The solution to this subset of the Poisson equation in 2D is called a *minimal surface*, or more colloquially, a *rubber sheet*. Seamless cloning and diffusion curve rendering are two applications of minimal surfaces.

Seamless cloning. Perez et al. [2003] described a number of image manipulation techniques that relied on different solutions of the Poisson equation, the most successful of which they named seamless cloning. Here, the goal is to paste a source image patch into a destination image without leaving a visible seam. To do this the source patch is modified to conform to the corresponding boundary in the destination image.

An appealing formulation of seamless cloning is to find the *difference* between the source patch and its seamless counterpart, which results to a minimal surface problem. Observing that the according rubber sheet function is extraordinarily smooth away from the boundaries, Agarwala [2007] proposed a quadtree decomposition to solve for very large rubber sheets for use in stitching image panoramas.

In work simultaneous to ours, Farbman et al. [2009] dispense with the Poisson equation altogether. As an alternative, they note that a true Laplacian membrane is not required for seamless cloning and many other applications — almost any surface will do as long as it has the correct boundary and a smooth interior. Thus, they define an alternate membrane based on mean value coordinates that is suitable for seamless cloning. While this new membrane can be calculated fairly quickly, it requires a lengthy setup time, up to several seconds, whenever the boundary shape changes, so it is not suitable for interactive applications.

Diffusion curves. A more recent application of minimal surfaces is the diffusion curve image (DCI) [Orzan et al. 2008]. The diffusion curve description extends the ideas of Elder et al. [2001], who showed that edges are a near complete image representation. A DCI represents an image as a set of curves with associated color values on either side that diffuse outwards to fill the image plane, creating a minimal surface. A DCI is therefore just a nice encoding of a rubber sheet. This vector encoding has a number of appealing properties. It is compact, and the curves have an intuitive meaning, which makes them easy to work with. Also the gradients possible in a diffusion curve are more expressive than those available in most other vector graphics formats. However, a major question is how to efficiently render diffusion curves, since they do not have a closed-form solution as do other vector graphics.

Orzan et al. [2008] designed a multigrid solution to render the diffusion curves. While fast, this solver suffers from strobing artifacts. The sources of the artifacts are two-fold. First, the curves that define the rubber sheet surface are not rasterized in a robust manner, and the diffusion amplifies these rasterization errors. Second, the solver itself is not able to converge to within visual tolerance within a few iterations. This might not necessarily be a problem, except that the artifacts align with coarse levels in the multigrid. Consequently, small translations can result in large changes in the image, leading to more strobing.

We present solutions to both problems above: a discretization tech-

nique that robustly transforms the continuous curve colors onto a regular grid for the diffusion process, as well as an easy to implement diffusion solver that is faster than existing multigrid solvers. Our solution is based on variable stencil sizes instead of a hierarchy, as is multigrid. It is also related to "morphological interpolation" [Salembier et al. 1996].

3 A Minimal Surface Solver for Diffusion Curves

3.1 Overview

In this section we present the two main parts of our solution for diffusion curve rendering: the curve rasterizer and the variable stencil size diffusion solver. The rasterizer takes as input a set of curves with associated colors, and produces a distance map of the curves along with an initial guess of the solution. The diffusion solver then diffuses the guess image, using the distance map as a guide to prevent color mixing over curve boundaries. Together, the rasterizer and diffusion solver allow us to render diffusion curves quickly and without the strobing artifacts present in previous solutions.

3.2 Robust Curve Rasterization

The first step in rendering a diffusion curve is to rasterize the continuous curves into a discrete image. It is essential that rasterization be handled robustly, as even a single wrong pixel in the input can result in large artifacts after the diffusion process. This process is made more difficult when one considers even simple manipulations such as subpixel panning and curve scaling.

Our rasterizer avoids fragile single pixel dependencies by rasterizing curve colors over the *entire* image and not solely relying on local information during the diffusion. First, we divide the curves into a number of small linear segments, which are fed to the rasterizer. These segments can be sampled from any curve description, in our case Bezier splines as described in Orzan et al. [2008]. The rasterizer creates a discrete Voronoi diagram of the segments, initializing pixels to the color of the closest curve point, as shown in figure 1 (middle). For this step we implemented the algorithm of Hoff et al. [1999] which we briefly review here.

The algorithm of Hoff et al. creates a Voronoi diagram by rendering "slanted" polygons around each curve segment, letting the z-buffer comparison generate the respective Voronoi regions. For each linear segment we construct a *tent* as two quads starting with depth zero at the segment and growing perpendicularly outward towards depth one. The size perpendicular to the segment, s, is the diagonal of the output texture. This ensures that each segment will stretch across the whole texture. On the outer side at each junction between adjoining segments we define a *triangle fan* that closes the gap between adjacent tents: starting at the junction at depth zero and growing towards depth one at size s. Fan triangles are generated so that their angle at the junction is no larger than 45 degrees. Similarly, at each curve endpoint we form a 180 degree fan consisting of four 45 degree triangles. In our implementation all polygons are generated in the geometry shader. To form the fans between segments, neighborhood information about the predecessor and successor segments are stored per vertex.

When the slanted polygons are rasterized, the Voronoi regions emerge automatically due to z-buffer comparison. We store the following per-pixel information in addition to the depth: (1) the curve color of the closest point on the same side of the curve segment as the point. (2) the *opposite* side curve color at the same point. This will later be needed for anti-aliasing. (3) the distance of the pixel to the curve, which results in a distance map (i.e. we store the actual distance in addition to the depth, rather than just using the depth as the distance). Together, we call the 3 values the "closest point map".

Note that a 45 degree fan introduces a distance error of up to $(1 - \cos\frac{45}{2}^\circ) \approx 8\%$ [Hoff et al. 1999] where triangle fans overlap, since depth is not exactly equal to distance on the fans. We will compensate for this distance error in the diffusion step in the next section. The error could be eliminated by setting depth in the pixel shader but unfortunately, the performance hit is unacceptably large as it disables early z-culling on current hardware, resulting in an extremely high fill rate. Table 1 shows some performance figures with and without early z culling. As one can see, early z culling decreases the rendering time by factors between 8 and 55 with an increasing acceleration factor for more complex models. Fortunately, future APIs (DirectX 11) will provide early z-culling even if depth is changed in the pixel shader.

# Curve segments	109	297	338	656	1193	1779
With early z (FPS)	333	243	211	139	96	73
Without early z (FPS)	42	19	7.1	3.75	1.97	1.32
Acceleration factor	7.9	12.8	29.7	37.07	48.7	55.3

Table 1: *Curve rasterization speed comparison in frames per second (*800×800 *pixels) with and without early z culling for curve models with different complexity.*

3.3 A Variable Stencil Size Diffusion Solver

The Jacobi method. Perhaps the simplest way to solve the diffusion equation is to use Jacobi iterations. The end goal of the diffusion equation is to minimize the Laplacian $\nabla^2 G$ of an image. Adding fixed pixels, such as boundary curves, imposes color constraints on the system:

$$\begin{aligned} G(x,y) &= B(x,y) \quad \text{if } (x,y) \text{ is a boundary value, and} \\ \nabla^2 G(x,y) &= 0 \qquad\quad \text{otherwise.} \end{aligned} \tag{1}$$

In the equation (x, y) denotes an image pixel, and B the Dirichlet boundary values. The Laplacian operator is discretized as

$$\nabla^2 G(x,y) = \sum_{i=1}^{4} G(n_i) \; - \; 4G(x,y), \tag{2}$$

where n_i denotes the pixels in the 4 connected neighborhood of (x, y). A standard Jacobi solver iterates toward the solution by setting each pixel to the average of its direct neighbors:

$$G(x,y) = \frac{1}{4} \sum_{i=1}^{4} G(n_i). \tag{3}$$

Boundary constraints are maintained by fixing pixel colors near the curves. This basic approach converges quite slowly, however, taking more than 20,000 iterations for a modest 512×512 pixel image.

Varying the stencil size. The main reason for the slow convergence of Jacobi is that color cannot quickly travel from one part of the image to another. Here we present a more efficient approach that applies similar iterations but with *larger stencil sizes* to more quickly transport colors over the image. We start our discussion with the 1D case and common Jacobi iterations. In this case, the

Initial guess | 1 step | 2 steps | 8 steps | 400 steps

Figure 3: *Demonstration of our diffusion solver using the "shrink always" strategy. After just 8 diffusion steps, the image is visually smooth, and very similar in appearance to the converged result using 400 steps.*

converged solution between two fixed boundary points is clearly always linear. Let

$$Ag = b \tag{4}$$

be the 1D analog of equation 1 in matrix form. Here, g is the function that we want to find. b gives the desired Laplacian and boundary values; it is zero everywhere except at the boundary points. The matrix A is tridiagonal, and has two types of rows: one row type corresponds to the boundary values, and it contains only 1's on the diagonal, and 0's elsewhere. The other row type, corresponding to the Laplacian operator, contains a -2 on the diagonal and two 1's next to it. The matrix A is *irreducibly diagonally dominant*. Consequently, by the Gerschgorin circle theorem, the matrix has full rank and the Jacobi method is guaranteed to converge to an unique solution.

The interesting point is that A remains irreducibly diagonally dominant no matter where the off-diagonal elements are placed (and Jacobi is still guaranteed to converge). Thus, we can modify A to place the off diagonal elements at distance $\pm h$ from the diagonal, instead of right next to it. In particular, we choose h to be the distance to the closest fixed boundary point. Now we show that the modified system has the same solution as the original as follows: we set g to the solution of the original system (linear between boundary points) and run a single Jacobi iteration on the new system. Any non boundary point g_i will be set to $\frac{g_{i-h}+g_{i+h}}{2}$ during the Jacobi iteration. However, this is equal to g_i, since all three points lie on the same line by construction. Thus, the system is in a converged state, and the solution to the modified system is the same as for the standard system.

An interesting point is that the spectral radius of the Jacobi iteration matrix J compared to standard diffusion is appreciably smaller, which makes the matrix easier to solve. For example, in 1D, with seven values and Dirichlet boundary conditions on the end, the iteration matrix for the standard Laplacian stencil is

$$\mathsf{J} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.5 & 0 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

J has a spectral radius of 0.87, and this value quickly approaches 1 for larger problem setups. On the other hand, with the large stencil sizes the iteration matrix becomes

$$\mathsf{J} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.5 & 0 & 0.5 & 0 & 0 & 0 & 0 \\ 0.5 & 0 & 0 & 0 & 0.5 & 0 & 0 \\ 0.5 & 0 & 0 & 0 & 0 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix},$$

with spectral radius 0.5. Larger problem sizes retain this small spectral radius so that the diffusion happens faster.

It seems natural that the best choice for h is the distance to the closest boundary value as this provides fast convergence. Figure 4 shows this process. Starting from two Voronoi regions (red and blue), one can observe how plateaus of equal color subdivide, cutting the error in half with each diffusion step.

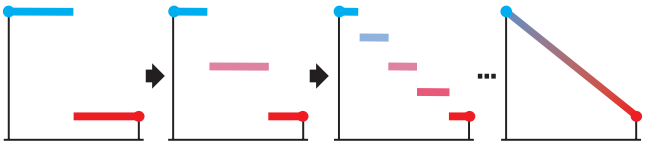

Figure 4: *0, 1, 2 and ∞ variable stencil size diffusion iterations in one dimension.*

The 2D case. The direct 2D extension of the above setup is to use *circles* around each sample that do not cross any boundary curve (we can ensure this by using the distance map). As in the 1D case we again have an irreducibly diagonally dominant matrix by construction, which guarantees convergence to an unique solution. The mean value theorem for harmonic functions states that in a minimal surface the value at the center of a circle is equal to the average value on the circle boundary, which proves that this solution is identical to the minimal surface that we are looking for.

A practical 2D solver. The inputs to our solver are the Voronoi color image and the distance map obtained in the rasterization step (section 3.2). The former serves as an initial solution guess, and the latter determines the radius of the sampling circle around each pixel, which ensures that no boundaries are crossed. (We reduce all of the radii by 8% to compensate for errors in the distance map, as described in section 3.2.) Averaging all samples from a large circle during diffusion steps is clearly too slow for practical use, so in practice we reduced the number of samples to only four values in axis aligned directions. Unfortunately, since each sample now has a great influence on the result, artifacts similar to mach banding can

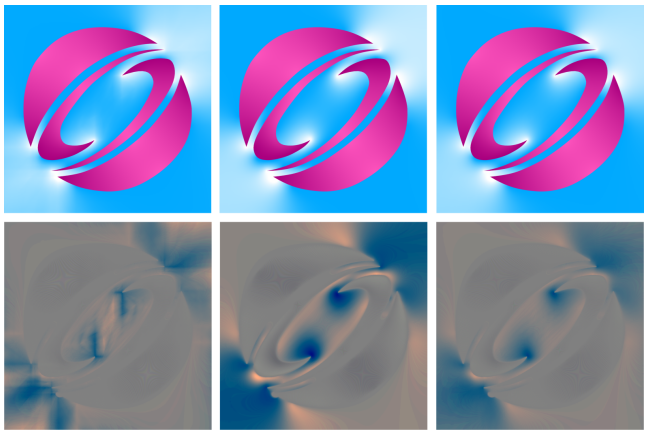

Figure 5: *Errors resulting from different shrinking strategies: (left) no shrinking, (middle) shrink always, (right) shrink half. The top row shows images after 8 diffusion iterations, and the bottom row shows the difference compared to a converged image, magnified 5 times.*

result. Figure 5 (left) shows such horizontal and vertical banding artifacts, mostly visible at the tips of the Siggraph logo. Our solution is to successively shrink the stencil radius during the diffusion.

We have identified two different shrinking strategies suitable for different situations, which we will refer to as "shrink always" (SA) and "shrink half" (SH). In SA, we shrink the stencil size linearly at each step. In other words, if the solver will perform n iterations, the radius on iteration i will be scaled by $1 - \frac{i}{n}$. This lets pixel values align more and more with their *local* neighborhood at later iterations, producing smoother behavior. The appeal of SA is that it removes visible banding artifacts and color plateaus quickly, almost always within 8 iterations. However, the color distribution is typically not converged at that point, which makes the image appear with slightly higher contrast. This difference is very minor, though, and difficult to see even with a side-by-side comparison to the converged solution. Figure 3 shows an example result of SA over a number of iterations. Note the very subtle differences between the solution with 8 steps and the converged solution with 400.

The SH strategy is similar to SA, except that the sampling radii are kept at full size for the first half of the diffusion iterations. SH converges towards the correct result faster than SA, but visible banding artifacts may remain for up to 12 or 14 iterations. Figure 5 compares the kinds of errors that result after 8 iterations with different shrinking strategies. At this number of iterations, SH exhibits less than half the error of SA. Even so, the proper shrinking strategy for a given application depends on the desired accuracy and smoothness of the result. Of course, both SA and SH converge to the right solution with an increasing number of steps, simply by the fact that the last iterations are identical to standard diffusion. In addition, in-between solutions are visually appealing because all errors have very low frequency. Figure 5 demonstrates that this type of error is hardly visible for the human eye: the left image with smaller but high frequency error looks less appealing. We also maintain temporal coherence, i.e., flickering is avoided as can be seen in the accompanying video.

3.4 Reblurring

In order to support unsharp edges, curves can be blurred after the diffusion. To facilitate this process, blur values are defined along curves. These values get diffused over the image along with the colors, resulting in a *blur map*. To complete the blur, Orzan et al. [2008] apply a spatially varying convolution to the diffused image, with a kernel width specified by the blur map. This process can be quite slow, however, and Orzan et al. report times of near a full second per frame.

Figure 6: *(Center) Unblurred images. (Left) The blur of Orzan et al. (Right) Our fast separable blur, showing little visual difference in the output.*

To increase the speed of the reblurring step, we simply pretend that the spatially varying convolution is separable, and perform repeated box filtering in each direction, with a different filter radius at each pixel. (Recall that if n box filtering passes are performed, the kernel radius should be divided by $\sqrt{n}$ to achieve approximately the same amount of blur as a single pass.) We fix the number of samples taken at each pixel to 10 to even out the rendering time. Our experience is that we can achieve good blur results using only two passes, which is effectively a tent filter. When using only two passes, we have found that we can achieve a smoother blur by decreasing the kernel radius by 5% on the second pass to stagger the sample locations. Although the separable approach is not entirely accurate, in practice the method is visually difficult to distinguish from the brute force result, and it runs at more than 200 Hz on an 800×800 image. Figure 6 compares our separable blur to the approach of Orzan et al.

4 Performance and Applications

We have implemented our curve rasterizer and variable stencil size diffusion solver in HLSL. Our tests were rendered on a Geforce 8800 GTX with 768 MB of video memory. As shown in this section, the algorithm is quite fast compared to other diffusion solvers. The high performance is not surprising as our diffusion process basically consists of a few Jacobi-like iterations (8 for most of the results shown in this paper). Our algorithm also works well for 8 bit data and non power of two texture resolutions. The rendering pipeline, including rasterization, diffusion, blurring, and display is entirely done in graphics hardware.

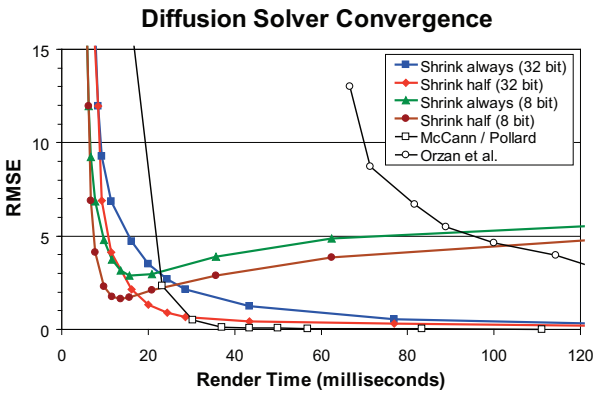

Figure 7: *A plot of render time vs. error (# of gray levels) for different variants of our solver, compared with the muligrid solutions of Orzan et al. and McCann and Pollard. All times are plotted for a 1024^2 image and the curve points correspond with different numbers of solver iterations. Times for our solver include distance map creation as well as diffusion.*

4.1 Speed and Convergence.

Figure 7 compares the convergence rates of different versions of our algorithm to two different multigrid implementations, the diffusion curve renderer of Orzan et al. [2008] (using their application), and McCann and Pollard's gradient domain solver [2008] (which we reimplemented for testing purposes). The graphic used for the tests is the "Zephyr" image, although slightly more zoomed out than is shown in figure 3.

The ground-truth image for our algorithm was created using 50,000 standard diffusion steps at 32 bit accuracy, which takes about one minute for a 1024^2 image on our hardware. The results for our algorithm include the time to create the distance map and initial guess image, which was about 7.6 ms. for 32 bit (the same as 7 diffusion steps), and 5.8 ms. for 8 bit (10 diffusion steps). See Table 1 for more rasterization results. SH converges appreciably faster than SA in both the 8 and 32 bit modes. In 32 bit mode, SH achieves an RMSE of one gray level twice as fast as SA, but once again, SH may require more iterations to create a visually smooth membrane, which is all that is really required for diffusion curve rendering and seamless cloning. As might be expected, the 8 bit variants run almost twice as fast as full floating point. They also converge at a similar rate for the first few (~ 16) iterations. Afterwards the error increases since rounding errors begin to dominate. Consequently, the diffused image gets slightly darker as the number of iterations increases. But the image quality of the 8 bit solver can still be considered high since low frequency RMSE errors of 2-5 are practically not noticeable by the human eye.

Comparison with Orzan et al. The solver of Orzan et al. does not perform multiple V cycles. Rather, it simply increases the number of smoothing steps to achieve a more converged result. The accepted image that we used for this solver was the image created with the maximum iterations allowed in their renderer. Their application produces a 512^2 (or 513^2) texture and rescales it for display, so we render at the default size and multiplied the render times by 4 for comparison. Roughly speaking, this solver converges between six and ten times slower than our solver, depending on settings. Furthermore, the output of their solver is not stable under translation even with a very large number of smoothing steps, whereas our output is visually stable, even under subpixel translations.

Figure 8: *Examples of animated diffusion curves. Left: animated water waves. Right: a moving magnifying glass.*

It is interesting to note that in a fully converged state, even our solver produces some strobing during subpixel movements. The problem is that there is a fragile dependency of the solution on the Dirichlet boundary conditions. All of the solvers we know about discretize the Dirichlet boundaries to specific pixels. The result is that even small changes lead to large changes in the output due to a changing input pixel mask. This is especially noticeable at curve endpoints where different fixed colors are spatially close.

Our solver is different than most solvers in that it starts with the whole image as an initial guess, instead of individual rasterized lines. This initial image dictates the result after the first few iterations rather than the boundary conditions, which makes these in-between results very stable and visually pleasing. However, as stated above, the fully converged solution still has a fragile dependence on fixed rasterized pixels near the curves, which produces strobing. In the same vein, if only pixels near curves are initialized with the right color, our solver is much slower. For that case it takes more than 50 iterations for the SH strategy to entirely fill the canvas and visually converge (SA takes even longer, as one might expect). The conclusion we draw from this is that the initial Voronoi rasterization supports a stable and visually pleasing (though not totally converged) solution, but cannot remedy the inherent problems of discretized Dirichlet boundary conditions.

To quantify the discretization error in our solver, we computed a number of different renderings of the same graphic with subpixel translations and different numbers of iterations. We then compared these in a pair-wise fashion to determine the magnitude of strobing that might occur. Even in extreme cases, we measured RMSE differences of less than 1.5 gray levels with 8 to 16 iterations of our solver. This was typically not visible. However with more than 16 iterations, the fixed boundary conditions start to assert themselves, and discernible strobing occurs by 32 iterations (RMSE 3.8) , and by 256 iterations, the error may be as high as 13 gray levels.

Comparison with McCann and Pollard. McCann and Pollard's multigrid solver for gradient domain painting runs multiple V cycles with no pre-smoothing and 2 post-smoothing steps. It handles non-zero Laplacian internal conditions and Neumann domain boundaries, rather than internal Dirichlet boundaries as does our solver. However, McCann and Pollard's solver can perform seamless cloning, as can ours, so it is reasonable to compare the two. To make a meaningful comparison, we created a test with non-zero Laplacian values at the same image pixels as our Dirichlet boundaries. In general it takes 4 V cycles for McCann and Pollard's solver to reach a stable state that does not flicker visibly under translation. This is about half as fast as our solver for smooth, visually stable results. Four V cycles is also the point at which their solver overtakes

Figure 9: *Rendering with different curve styles.*

our best variant (32 bit SH) in terms of error vs. time. However, this solver does not solve the same problem as ours, and it is not suitable for diffusion curve rendering.

4.2 2D Diffusion Curve Rendering

In general the input to a diffusion curve solver has to be very accurate near curves as even single pixel errors can lead to large visible errors in the final image. For animated images such as those shown in figure 8, small errors are prominently visible as flickering or strobing artifacts. The problem becomes even more complicated for minified viewing where multiple curves might cover a single pixel. In contrast to the solution by Orzan et al. [2008] that poses problems for features smaller than 3 pixels, our rasterization method is very consistent, even under minified viewing. If multiple curves fall into a pixel, that pixel itself may get an arbitrary color, but pixels that are further away get the correct colors from the nearest curve. Consequently, our approach produces no visible color leaking or strobing artifacts during animations, as can be observed in the accompanying video.

Since we have the sampled distance function and the parametrization and colors at the closest curve point we can easily anti-alias edges or apply effects based on curve distance, such as outlining, procedural strokes, embossing, color gradients, or drop shadows as Qin et al. [2008]. Figure 9 shows examples of different stoke styles. In addition, it is interesting to note that closed curves with only a single color defined on each side are already correctly represented after the rasterization step. This means, images only consisting of such curves with a constant exterior color can be rendered without diffusion, which is an interesting subclass of DCIs, close to traditional vector graphics.

4.3 Real-time Seamless Cloning

A second application of our solver is seamless image cloning [Pérez et al. 2003]. In seamless cloning, the goal is to paste a given source image patch into another image without leaving visible seams. This is done by swapping the source rubber sheet, defined by the boundaries of the source patch, with the destination rubber sheet, defined by the corresponding pixels in the destination image. In other words, we add the *difference* between the destination and source rubber sheets over the patch to the source image:

$$F_{dest}(\mathbf{x},\mathbf{y}) = F_{src}(\mathbf{x},\mathbf{y}) + G_{dest}(\mathbf{x},\mathbf{y}) - G_{src}(\mathbf{x},\mathbf{y}) \qquad (5)$$

where F_{dest} is the final cloned image, F_{src} is the source patch, and G_{dest} and G_{src} are the destination and source rubber sheets defined by the patch boundaries. To perform the rubber sheet swap, we compute the difference rubber sheet $(G_{dest} - G_{src})$ as described in Section 3.3. However, we use the jump flood algorithm [Rong and Tan 2006] to create the distance map. A straightforward application of the rasterization routine described in Section 3.2 would rasterize a cone for each pixel on the cloning boundary which is a more expensive operation.

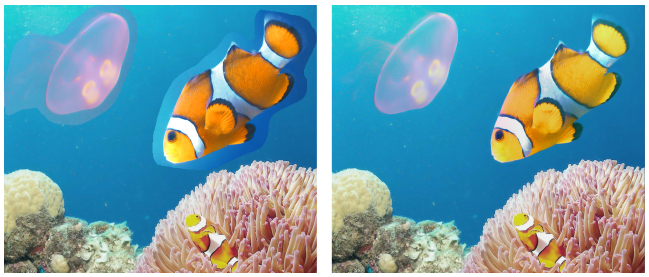

Figure 11: *Seamless cloning with our new solver.*

Figure 11 demonstrates seamless cloning using our diffusion solver. This example contains about 350 thousand cloned pixels, and runs at 104 fps with a destination image size of 1024^2, including the time for jump flooding. This time could be somewhat improved by confining the distance map and diffusion renderings to the bounding box of the cloned area. Figure 10 shows a second cloning example, stitching together a large panorama. Here, every second photo was seamlessly pasted into the final image, yielding a seamless composite. This example runs at 26 fps at 4096×1024 pixels and 3.6 fps at 8192×2048.

Comparison with mean value cloning. The work of Farbman et al. [2009], developed at the same time as ours, constructs a difference membrane to perform seamless cloning based on mean-value coordinates. Their application achieves a peak cloning rate of 11 M pixels/sec. on the CPU and 44 M on the GPU. This does not include the significant preparation time required whenever the shape of the cloned patch is modified (0.5 sec. for an example comparable to figure 11, and 3.6 sec. for an example comparable to figure 10). The long preparation times make mean value cloning impractical for interactive editing of large patch boundaries. By contrast, our cloning rate for figure 11 is 36 M pixels/sec. including preparation time, and 63 M without.

5 Conclusions

This paper presented a variable stencil size diffusion solver for the discrete calculation of minimal surfaces in two dimensions. We showed that the new solver is both simple to implement, produces stable renderings, and is more efficient than competing multigrid solvers. We demonstrated the solver on several applications, including the rendering of diffusion curves, and real-time seamless cloning.

There are still a number of open questions related to the solver that could bear further study. For example, it is unclear how to achieve the best convergence rate per iteration. We tried several diffusion strategies besides the ones mentioned in the paper, such as performing a separable blur for some of the iterations, and gathering 8 samples instead of 4, but the simple 4 sample model worked best in our tests. Also, we experimented with different shrinking strategies, but did not find any that were substantially better than SA and SH. However, it would be interesting to attempt to determine an optimal shrinking function. Other optimization ideas include computing the distance map or some of the diffusion steps at lower resolution, essentially hybridizing our solver with a multigrid. The solver itself could also be extended in a number of ways, such as computing 3D diffusions and adding the capability to handle nonzero Laplacian boundary conditions. Finally, we believe that the variable stencil size idea, as opposed to hierarchical decomposition, has the potential to be a useful general technique for computer graphics. For example, the jump flooding that we used for seamless cloning is

Figure 10: *Image panorama taken at Delicate Arch near Moab, Utah, USA. (Top) Aligned photographs. (Bottom) Final seamless composite removing illumination changes between the photos.*

very much in this vein. A number of other search problems may fit well into the mold of variable stencil size search and processing.

References

AGARWALA, A. 2007. Efficient gradient-domain compositing using quadtrees. In *ACM Trans. Graph. (SIGGRAPH 2007)*, vol. 26, 1–5.

ELDER, J. H., AND GOLDBERG, R. M. 2001. Image editing in the contour domain. In *PAMI '01*, IEEE Computer Society, Washington DC, USA, 291–296.

FARBMAN, Z., HOFFER, G., LIPMAN, Y., COHEN-OR, D., AND LISCHINSKI, D. 2009. Coordinates for instant image cloning. In *ACM Trans. Graph. (SIGGRAPH 2009)*, vol. 28, 1–9.

GOODNIGHT, N., WOOLLEY, C., LEWIN, G., LUEBKE, D., AND HUMPHREYS, G. 2003. A multigrid solver for boundary value problems using programmable graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 102–111.

GRADY, L., TASDIZEN, T., AND WHITAKER, R. 2005. A geometric multigrid approach to solving the 2D inhomogeneous Laplace equation with internal Dirichlet boundary conditions. In *ICIP '05: IEEE International Conference on Image Processing*, vol. 2, 642–645.

GRADY, L. 2008. A lattice-preserving multigrid method for solving the inhomogeneous Poisson equations used in image analysis. In *ECCV '08: Proceedings of the 10th European Conference on Computer Vision*, 252–264.

HOFF, K. E., KEYSER, J., LIN, M., MANOCHA, D., AND CULVER, T. 1999. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of SIGGRAPH '99*, 277–286.

IAN, J. B., FARMER, I., GRINSPUN, E., AND SCHROEDER, P. 2003. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *ACM Trans. Graph. (SIGGRAPH 2003)*, vol. 22, 917–924.

KAZHDAN, M., AND HOPPE, H. 2008. Streaming multigrid for gradient-domain operations on large images. In *ACM Trans. Graph. (SIGGRAPH 2008)*, vol. 27, 1–10.

KIMMEL, R., AND YAVNEH, I. 2002. An algebraic multigrid approach for image analysis. *SIAM Journal of Scientific Computing 24*, 4, 1218–1231.

MCCANN, J., AND POLLARD, N. S. 2008. Real-time gradient-domain painting. In *ACM Trans. Graph. (SIGGRAPH 2008)*, vol. 27, 1–7.

ORZAN, A., BOUSSEAU, A., WINNEMÖLLER, H., BARLA, P., THOLLOT, J., AND SALESIN, D. 2008. Diffusion curves: A vector representation for smooth-shaded images. In *ACM Trans. Graph. (SIGGRAPH 2008)*, vol. 27, 1–8.

PÉREZ, P., GANGNET, M., AND BLAKE, A. 2003. Poisson image editing. In *ACM Trans. Graph. (SIGGRAPH 2003)*, vol. 22, 313–318.

QIN, Z., MCCOOL, M. D., AND KAPLAN, C. 2008. Precise vector textures for real-time 3D rendering. In *Proceedings of I3D'08*, 199–206.

RONG, G., AND TAN, T. S. 2006. Jump flooding in GPU with applications to Voronoi diagram and distance transform. In *Proceedings of I3D '06*, 109–116.

SALEMBIER, P., BRIGGER, P., PARDS, J. R. C. M., AND CASAS, J. R. 1996. Morphological operators for image and video compression. *IEEE Trans. on Image Processing 5*, 881–897.