Interactive Depth of Field Using Simulated Diffusion on a GPU

Michael Kass, Pixar Animation Studios
Aaron Lefohn, U.C. Davis
John Owens, U.C. Davis

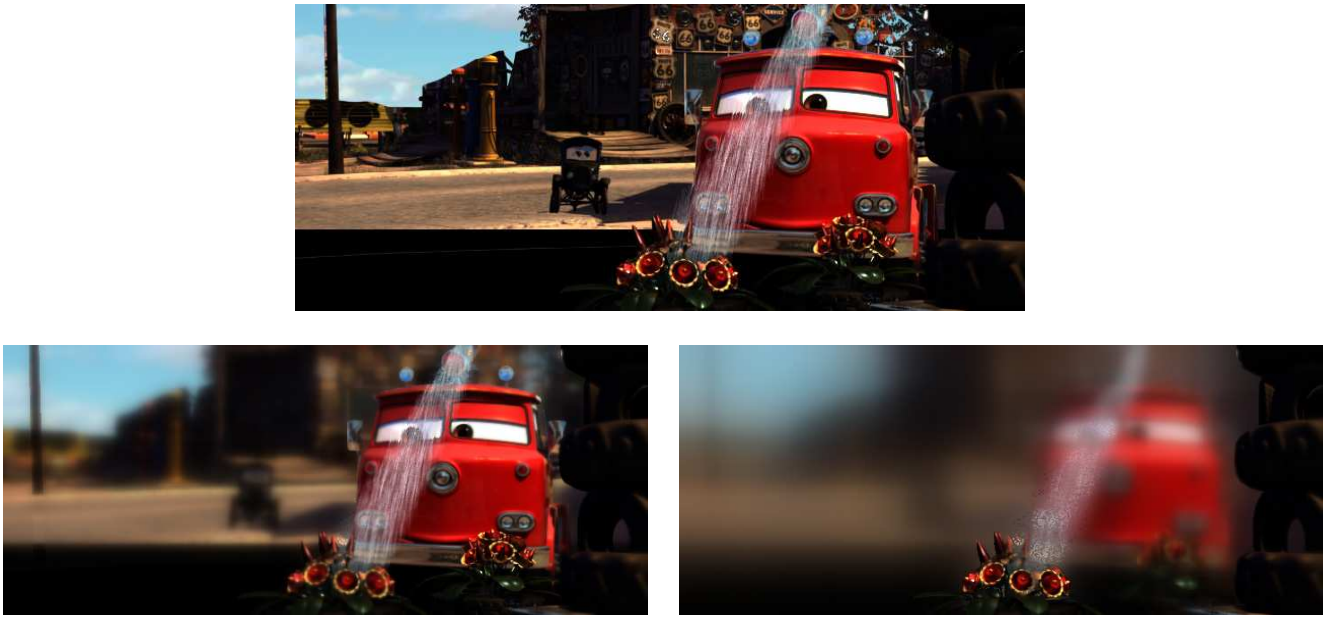

Figure 1: Top: Pinhole camera image from an upcoming feature film. Bottom: Sample results of our depth-of-field algorithm based on simulated diffusion. We generate these results from a single color and depth value per pixel, and the above images render at 23–25 frames per second. The method is designed to produce film-preview quality at interactive rates on a GPU. Fast preview should allow greater artistic control of depth-of-field effects.

Abstract

Accurate computation of depth-of-field effects in computer graphics rendering is generally very time consuming, creating a problematic workflow for film authoring. The computation is particularly challenging because it depends on large-scale spatially-varying filtering that must accurately respect complex boundaries. A variety of real-time algorithms have been proposed for games, but the compromises required to achieve the necessary frame rates have made them them unsuitable for film. Here we introduce an approximate depth-of-field computation that is good enough for film preview, yet can be computed interactively on a GPU. The computation creates depth-of-field blurs by simulating the heat equation for a non-uniform medium. Our alternating direction implicit solution gives rise to separable spatially varying recursive filters that can compute large-kernel convolutions in constant time per pixel while respecting the boundaries between in-focus and out-of-focus objects. Recursive filters have traditionally been viewed as problematic for GPUs, but using the well-established method of cyclic reduction of tridiagonal systems, we are able to vectorize the computation and achieve interactive frame rates.

CR Categories: I.3.7 [Computer Graphics]: Three Dimensional Graphics and Realism—Animation; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Physically-Based Modeling

Keywords: Depth of field, Diffusion, Heat Equation, Alternating-Direction Implicit Methods, GPU, Tridiagonal Matrices, Cyclic Reduction.

1 Introduction

Depth-of-field (DOF) effects are essential in producing computer graphics imagery that achieves the look and feel of film. Unfortunately, the computations needed to compute these effects have traditionally been very slow and unwieldy. As a consequence, the effects are both costly to create and difficult to direct. Here, we attempt to improve the process of creating DOF effects by introducing a high-quality preview that can be computed at interactive rates with a GPU. The image quality should be sufficient to allow a director of photography to specify, edit, and approve aperture settings and focus pulls interactively, offering a far greater level of artistic control than has previously been available. In time, as GPU speeds increase, the method may also become of interest to game developers desiring a good-quality interactive result.

Most high-end film renderers use the method of distributed ray tracing [Cook et al. 1984] to get accurate DOF results that take into account the varied paths of light through different parts of a lens. While the method certainly produces high-quality results, it is slow, and if the focal plane or aperture are changed, all the ray paths are altered, and the rendering process must be restarted from the beginning. Our goal is to use a much simpler computation to get a good-quality interactive preview that can be used to adjust and animate the relevant parameters. Once the parameters have been chosen, distributed ray tracing can be used to compute the final result.

Potmesil and Chakravarty [1981] introduced the idea that approximate DOF effects can be computed by post-processing an image rendered from a pinhole camera. Since the image from a pinhole camera only contains information about rays that pass through a single point, no amount of post-processing can make up for the lack of information about other rays and create a DOF computation good enough for high-end film rendering. Nonetheless, in most cases, the pinhole image does contain enough information to produce a result acceptable for film preview or game purposes. Following Potmesil and Chakravarty, we take the pinhole camera image with depth as our starting point.

Photographers have long known that a small spot at a given depth will be blurred out in a camera to a size known as the "circle of confusion." Moreover, for ordinary lenses, the circle of confusion depends in a simple way on the aperture and the focal plane. Noting this, Potmesil and Chakravarty proposed blurring each pixel by its circle of confusion to simulate DOF effects. While the basic approach is sound, the blurring must be done in a more careful way than presented in their original paper in order to avoid objectionable artifacts. In particular, care must be taken to keep sharp, in-focus objects and blurry backgrounds from bleeding into each other.

From a computational standpoint, one of the chief challenges of rapid DOF computation is that out-of-focus regions can require large kernel convolutions to blur appropriately. While there are efficient ways of computing large kernel convolutions that are well established in graphics [Burt and Adelson 1983; Williams 1983], they cannot easily be made to respect complex boundaries between in-focus and out-of-focus objects.

Our approach is to achieve the necessary blurring by simulating heat diffusion in a non-uniform medium. The basic idea is to convert the circles of confusion into varying heat conductivity and allow the pixel values in an image to diffuse as if they were a series of temperature samples. A key feature of this formulation is that when the heat conductivity drops to zero, the blurring will stop dead in its tracks, precisely respecting boundaries between sharp in-focus objects and neighboring objects that are very blurred. By casting the DOF problem in terms of a differential equation, it becomes much easier to deal with the spatially varying circles of confusion that cause so many problems with traditional signal-processing and filtering approaches. Our solution method is an alternating direction implicit method that leads to a series of tridiagonal linear systems. These linear systems effectively implement separable recursive filters, and can be computed in constant time per pixel, independent of the size of the circles of confusion.

The rest of the paper is as follows. In Section 2, we discuss relevant prior work. In Section 3 we introduce the heat equation and show that solving it with an alternating direction implicit method leads to a series of tridiagonal systems. In Section 4, we describe how to implement this computation efficiently on GPU hardware. Then in Section 5, we provide the results of running the algorithm on a variety of scenes.

2 Prior work

Approaches to computing DOF vary in the detail with which they model the lens and light transport, their performance-quality tradeoffs, and in their suitability to implementation on graphics hardware. Demers provides a recent survey of approaches to the DOF problem [2004].

In order to generate a high-accuracy result, a DOF computation must combine information about rays that pass through different parts of a lens. The accumulation buffer [Haeberli and Akeley 1990] takes this approach, simulating DOF effects by blending together the results of multiple renderings, each taken from slightly different viewpoints. Unfortunately, the method requires a large collection of renderings to achieve a pleasing result (Haeberli and Akeley use 23 to 66), and the enormous geometric complexity of film-quality scenes makes this prohibitive. It is not unusual for the geometry of film-quality scenes to exceed any available RAM, so doing multiple passes through the original geometry is out of the question for interactive film preview.

In order to achieve interactive performance, there is little choice other than to rely largely on the post-processing approach of Potmesil and Chakravarty [1981]. Their work has inspired a variety of algorithms which can be divided into two major categories:

Scatter techniques (also known as forward-mapping techniques) iterate through the source color image, computing the circle of confusion for each source pixel and splatting its contributions to each destination pixel. Proper compositing requires a sort from back to front, and the blending must be done with high-precision to avoid artifacts. Distributing energy properly in the face of occlusions is also a difficult task. Though scatter techniques are commonly used in non-real-time post-processing packages [Demers 2004], they are not the techniques of choice for today's real-time applications primarily because of the cost of the sort, the lack of high-precision blending on graphics hardware, and the difficulty of conserving total image energy.

Gather techniques (also known as reverse-mapping techniques) do the opposite: they iterate through the destination image, computing the circle of confusion for each destination pixel and with it, gathering information from each source pixel to form the final image. The gather operation is better suited for graphics hardware than scatter. Indeed, the most popular real-time DOF implementations today all use this technique [**?**; Riguer et al. 2003; Scheuermann and Tatarchuk 2004; Yu 2004]. Nonetheless, the gather operation is still not very well matched to today's SIMD graphics hardware because of the nonuniformity of the sizes of the circles of confusion. The method also has difficulty with edge discontinuities and edge bleed, neither of which is eliminated in any published algorithm.

Even if efficiently implemented on the target hardware, standard gather and scatter techniques have poor asymptotic complexity because the amount of work they do is the product of the number of pixels in the image and the average area of the circle of confusion. For an $n \times n$ image, these algorithms are $O(n^4)$ which is clearly problematic for high-resolution film-quality images.

In order to bring the computational cost down to a level that permits real-time performance, some implementations, such as Scheuermann and Tatarchuk [2004], compute large blur kernels by downsampling. While this is a perfectly sensible compromise to achieve adequate performance for games on current hardware, it comes at the expense of artifacts that are unacceptable for film preview. The problem is that existing techniques do not allow large-scale blurs to be computed efficiently in ways that respect the critical boundaries between in-focus objects and those that are out-of-focus. As a re-

sult, the acceleration methods will cause unacceptable color bleeding.

3 Heat Diffusion

The main requirement in DOF computation with a post-processing method is to blur the image with a spatially varying filter width given by the circle of confusion. This poses two key challenges. First, while blurring, we must maintain accurate and sharp boundaries between areas of the image that are in focus and those that are out of focus. Second, in order to achieve interactive speed, we must be able to compute large blurs efficiently. No existing method meets both of these challenges at once.

Our approach to these challenges is to compute the blurring by simulating the heat diffusion equation. The image intensities from a pinhole camera view in our method provides a heat distribution that diffuses outward to produce the DOF image. Where the circles of confusion are large, we model the thermal conductivity of the medium as high, so the diffusion will extend outward to an appropriate radius. Where the circle of confusion reaches zero, the thermal conductivity will correspondingly be zero, creating a perfect insulator that completely decouples the color of a sharp object from the color of an adjacent blurry object.

Consider an input image $x(u,v)$ which we want to diffuse into an output image $y(u,v)$. The basic heat equation can be written

$$\gamma(u,v)\frac{\partial y}{\partial t} = \nabla\cdot(\beta(u,v)\nabla y) \tag{1}$$

where $\beta(u,v)$ is the heat conductivity of the medium, $\gamma(u,v)$ is the specific heat of the medium and ∇ represents the del operator in terms of the spatial coordinates u and v. We will use the input image to provide the initial heat distribution for the diffusion, and then integrate the heat equation through time to get a blurred result.

There are a variety of numerical methods that can be employed to solve the heat equation, but only a well-suited method will provide adequate performance. The simplest of all methods is to begin with $y(0) = x$, evaluate the derivative $\partial y/\partial t$ at time zero, and then take a step where $y(\Delta t) = y(0) + \Delta t(\partial y/\partial t)$. Unfortunately, this method, known as forward Euler's method, is unacceptably slow. The method yields a step given by a small Fixed Impulse Response (FIR) filter, and with repeated applications, it takes $O(n^2)$ FIR convolutions to produce a filter with width proportional to n. In order to achieve better performance, we choose an alternating direction implicit (ADI) solution method [Press et al. 1992], which instead gives rise to very efficient separable Infinite Impulse Response (IIR) or recursive filters. ADI methods have been used previously in graphics to simulate shallow water [Kass and Miller 1990] and in that context achieve constant time per surface sample, independent of wave speed.

The basic idea of the ADI solution of the heat equation is to split the solution into two substeps. In the first substep, heat will diffuse along the u axis. During the second substep, the heat distribution will further diffuse along the v axis. While ADI is a well-established technique for solving differential equations, it always carries a risk that the existence of preferred directions in the solution, u and v, may produce objectionable anisotropies in the result. With the diffusion equation, however, these anisotropies turn out to be particularly small. The exact solution of the diffusion equation in a uniform medium after a fixed time is given by the convolution of the initial conditions with a 2D Gaussian. Since a 2D Gaussian convolution can be computed exactly by a horizontal 1D Gaussian convolution followed by a vertical 1D Gaussian convolution, the ADI approach is particularly well-justified in this case.

In each substep the ADI method must solve a 1D diffusion equation given by

$$\gamma\frac{\partial y}{\partial t} = \frac{\partial}{\partial u}\,\beta(u)\,\frac{\partial y}{\partial u}. \tag{2}$$

In an ADI approach, each of these substeps is computed with an implicit method. There is a vast literature in the numerical analysis community on implicit methods; Baraff et al. [2003] provide a tutorial as they relate to problems in CG. For our purposes, the simplest implicit scheme, known as backwards Euler, will suffice.

$$\frac{y(t+\Delta t)-y(t)}{\Delta t} = \left.\frac{\partial y}{\partial t}\right|_{t+\Delta t} \tag{3}$$

In contrast to the forward or explicit Euler method mentioned earlier, here the time derivative $\partial y/\partial t$ is evaluated at the end of the step, rather than the beginning. The result is a set of simultaneous linear equations for the solution which allows the diffusion to propagate arbitrarily far in a single step.

We are free to choose any units for time and space, so for simplicity, we will choose units in which $\Delta t = 1$ and the separation between pixels is unit distance. With these units, discretizing over space with finite differences yields

$$\gamma_i\frac{\partial y}{\partial t} \approx \beta_i(y_{i+1}-y_i) - \beta_{i-1}(y_i - y_{i-1}). \tag{4}$$

If we begin with the initial conditions $y_i = x_i$, and then take a single time step using the implicit Euler method of Equation 3, we get

$$\gamma_i(y_i - x_i) = \beta_i(y_{i+1}-y_i) - \beta_{i-1}(y_i - y_{i-1}) \tag{5}$$

where $\beta_0 = \beta_n = 0$, so that the boundary of the image is surrounded by insulators.

In order to set up Equation 5 from a DOF problem, we need to know the relationship between β and the size of a circle of confusion. To do this, consider the situation where γ is unit and β is uniform. Then, Equation 5 can be written:

$$y_i - x_i = \beta(y_{i+1} - 2y_i + y_{i-1}) \tag{6}$$

The right-hand-side of Equation 6 is the product of β and a finite difference approximation to the second derivative of y. Taking a Fourier transform of both sides and noting that taking n derivatives in space is the same as multiplying by $(i\omega)^n$ in frequency, we obtain

$$\tilde{y} - \tilde{x} = \beta(i\omega)^2\tilde{y} \tag{7}$$

which yields the frequency response

$$\tilde{y} = \frac{1}{1+\beta\omega^2}\tilde{x}. \tag{8}$$

of a Butterworth low-pass filter. Traditionally, Butterworth filters are described in terms of a cutoff frequency ω_c

$$\tilde{y} = \frac{1}{1+(\omega/\omega_c)^2}\tilde{x}. \tag{9}$$

and in these terms, $\beta = 1/\omega_c^2$. The spatial width corresponding to the diameter of the filter is just $1/\omega_c$, so we have $\beta = d^2$ where d is the diameter of the circle of confusion.

Figure 2: Top: original pinhole image. Bottom: image with simple DOF.

Potmesil and Chakravarty [1981] describe how to compute the circle of confusion for each pixel from its depth and a full set of camera parameters. From this circle, we can now compute β, and complete Equation 5.

Equation 5 describes a symmetric tridiagonal linear system of the form:

$$\begin{pmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & c_3 & \\ & & \ddots & \ddots & \ddots \\ 0 & & & a_n & b_n \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} \tag{10}$$

where $c_i = a_{i+1}$. The tridiagonal structure of the equations result from the fact that each sample in the 1D heat equation is coupled only to the next and previous sample.

Unlike general linear systems, tridiagonal linear systems can be solved very efficiently in constant time per sample [Press et al. 1992]. The traditional method for doing this is to factor the matrix into the product of a lower-diagonal matrix and an upper-diagonal matrix in a process known as LU decomposition. Having computed the factors L and U, we are left with the linear system $LUy = x$. Next, we compute $z = Uy$ from L and x by a process known to applied mathematicians as "forward substitution" and known to people in the signal processing world as the application of a recursive filter. Then, knowing z, we compute y from the equation $Lz = x$ by a process known to applied mathematicians as "back substitution," and to people in the signal processing world as the application of a backwards recursive filter. From a signal-processing point of view, the unusual thing about the filters being run forward and backwards to solve the linear system is that their coefficients change over space, properly taking into account the boundary conditions. Because they are IIR filters, the amount of work they do is independent of the size of the filter kernel. While LU decomposition is a perfectly reasonable way to solve tridiagonal systems on a CPU, it is poorly suited to a GPU, so our actual implementation uses an equivalent but different method called "cyclic reduction," described in Section 4.

The basic algorithm for heat diffusion can now be described very simply. We begin with an RGBZ image and calculate circles of confusion for each pixel from the camera parameters. Then we compute the horizontal diffusion by forming and solving the tridiagonal system of Equation 5, assuming that the specific heat γ is uniformly equal to one. Note that the value of β_i in this equation corresponds to the link between pixels i and $i+1$. In order to guarantee that pixels with zero circle of confusion will not diffuse at all, we use the minimum circle of confusion at the two pixels to generate β_i. Once the horizontal diffusion is complete, we use the result as the starting point for the vertical diffusion. Figure 2 shows the results after both diffusion steps.

3.1 Blurring Underneath

While the simple algorithm just described works reasonably well for a range of scenes, it became evident after some testing that it only solves part of the DOF problem. To describe both the problem and the solution, it is valuable to distinguish three ranges of depth. The furthest depth range, which we will refer to as background, consists of portions of the image that lie far enough behind the plane of focus to have large circles of confusion. The next closer depth range, which we will refer to as midground, consists of portions of the image with depths near enough to the plane of focus on either side to have relatively small circles of confusion. Finally, the closest depth range, which we will refer to as foreground, consists of portions of the image enough closer than the focal plane to have large circles of confusion.

In general, the heat diffusion algorithm as just described works reasonably well for objects in the midground. It successfully maintains the sharpness of in-focus objects and prevents color bleeding from taking place between in-focus objects and neighboring out-of-focus objects. Unfortunately, good performance in the midground is not sufficient for our purposes. In real optical situations it is not uncommon for background objects to have circles of confusion so large that they blur behind sharp foreground objects. Since in-focus objects in our diffusion approach act as heat insulators, all blurring due to the heat equation is blocked by in-focus midground objects, and severe artifacts can result. Figure 3 shows an example. In the upper right image, thin leaves of a tree block the blurring of the yellowish background plane in such a way that the originally straight outline of the plane becomes unacceptably distorted.

In order to address the problem of blurring underneath in-focus objects, we introduce a separate layer to process background portions of the image with large circles of confusion. The idea is essentially to matte out the in-focus midground objects, blur the background objects across the removed regions, and then blend between the original (midground) layer and the new background layer based on the matte.

Thus far, we have not used the specific heat γ, but for computing the background layer, it will become critically important. In effect γ acts as a coupling coefficient between the initial conditions and the diffusion. Where γ is large, the initial conditions will greatly influence the final result. Where γ is zero, the initial conditions become entirely irrelevant.

Let $\alpha(u,v)$ be a matte that separates background regions from midground regions. α will be zero for pixels with small circles of confusion and ramp up smoothly to one for pixels with circles of confusion equal to or greater than a level that identifies them as background pixels. By setting $\gamma = \alpha$, we will be able to make our diffusion take into account the matte and interpolate the proper information in the gaps.

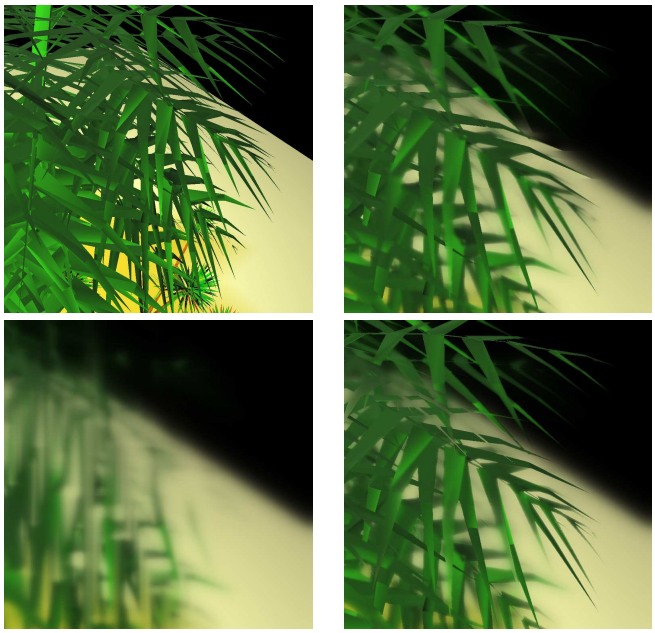

Figure 3: Our original image is at upper left. Single-layer diffusion (upper right) results in artifacts at the horizon adjacent to the leaves. We thus compute a separate background layer (lower left) and blend it with our single-layer diffusion for the final result (lower right).

Note that there are two kinds of information missing in the gaps where α is small. Clearly we are missing background colors, but equally importantly, we are also missing the corresponding circles of confusion. Before we can interpolate the colors appropriately, we need to estimate those circles.

To interpolate the circles of confusion, we have only to use our original circles of confusion instead of colors as the input to the diffusion computation. Setting $\gamma = \alpha$ ensures that circles of confusion from fully in-focus midground regions will be completely ignored. For this diffusion, it suffices to set β to be a constant such that the filter size is comparable to the blur-size threshold between midground and background. Figure 4 shows the original circles of confusion before matting and then the results after matting and smoothing.

Once we have interpolated the circles of confusion, we can run our original diffusion computation on the color information with $\gamma = \alpha$, and the colors will fill in the gaps. The lower left of figure 3 shows the result. The closest of the leaves have been removed by the α channel, and diffusion has filled in the background with a smooth interpolation. Matting our original midground computation over this background layer yields the result in the lower right of figure 3. The objectionable artifacts of the midground computation are all but gone, as the background layer provides blurring behind the long thin in-focus midground leaves.

There is one seemingly arbitrary choice in the algorithm: the threshold blur size that separates the midground from the background. For the highest possible quality, one can perform this background computation at a number of different thresholds chosen on a log scale, and then matte among all the different layers based on the circle of confusion of each pixel. While this approach does provide a slight improvement, the result with a single background layer works well enough in our experience that we consider extra background layers to be generally unnecessary.

Figure 4: These pictures show the sizes of the circles of confusion at each pixel, with dark regions corresponding to small circles (more in focus) and light regions corresponding to larger circles (less in focus). The left image here corresponds to Figure 3's top right image, and the right image corresponds to Figure 3's bottom left image.

3.2 Blurring On Top

There is a little-discussed, yet fundamental limitation of the post-processing method that arises with very blurry foreground objects. When a foreground object gets very blurry, it begins to become transparent because a wide enough aperture allows a camera to collect rays that go fully around the object. If the aperture is sufficiently wide, the transparency can be almost complete. The lower right image in figure 6 shows an example of this phenomenon. Even though we are looking through a fence, the camera aperture is wide enough that the foreground fence has become almost completely invisible.

The problem when a foreground object becomes transparent is what to show behind it. A postprocessing method as described by Potmesil and Chakravarty [1981] has no information about the scene behind the very blurry foreground object and cannot produce a high-quality result. In particular, if the blurry foreground object obscures a sharp midground object, there is no way for such an algorithm to create the high-frequency detail on the obscured object. A case in point is the American flag in figure 6. No postprocessing algorithm can be expected to invent the additional stripes in the flag that are obscured in the pinhole camera view in the upper left, yet are needed to produce a proper image in the lower right.

Only by supplementing the input to a postprocessing algorithm can we hope to achieve a high-quality result. Here we consider what can be done if the input can be separated into different layers. To produce the images in figure 6, we have taken as input, not only an RGBZ image as before, but also a separate RGBZα layer for the fence alone.

The extension of our algorithm to foreground objects in separate elements is relatively straightforward. As with our computation of the background layer, we begin by diffusing the circles of confusion. In this case, the weights γ are given by the α channel of the foreground input. Having calculated these new circles of confusion we diffuse not only the input colors, but also the input α, with γ again given by the foreground α channel. Finally, we take the diffused colors and alpha channel for the foreground layer and composite them over the previously described midground/background blend. Figure 6 shows the results.

4 Implementation

The key to a fast implementation of our DOF computation is the efficient solution of the tridiagonal system in equation 10. While previous authors have successfully developed solvers for a variety of linear systems on graphics hardware, none of the published algorithms are appropriate for this particular case. The general linear algebra framework of Krüger and Westermann [2003] supports memory layouts for both dense and sparse matrices, including banded matrices, but does not provide solvers specific to banded or tridiagonal matrices. Galoppo et al. support both Gauss-Jordan elimination and LU decomposition on the GPU, but only for dense matrices [2005]. GPU-based conjugate gradient solvers [Bolz et al. 2003; Goodnight et al. 2003; Hillesland et al. 2003] are iterative and do not take advantage of the special properties of banded or tridiagonal matrices. Moreover, for banded or tridiagonal systems, direct methods are generally much faster and better conditioned than iterative techniques.

We begin by describing the data structures necessary for our implementation, then describe our implementation of our solution to the heat diffusion equation, including our solver for tridiagonal systems, on the GPU.

4.1 Data Layout

In our implementation we must represent 2D arrays of input and output values (such as colors or depths, one per screen pixel) and a tridiagonal matrix. Representing a 2D array is straightforward: 2D arrays are stored as 2D textures, with individual entries (colors or depths) stored as texels (RGB for colors, floating-point scalars for depths). The structure of the tridiagonal matrix lends itself to storage as a 1D array. Each row of a tridiagonal matrix contains 3 elements (a_n, b_n, and c_n); those elements are stored in a single texel as R, G, and B. We can thus represent a 1D array of tridiagonal matrices in a single 2D texture, with entries in the tridiagonal matrix in the same texel positions as their corresponding entries in the 2D arrays of inputs or outputs.

We note that a tridiagonal system is a particular type of a recurrence equation and as such, can be efficiently solved in parallel using the *scan* primitive [Blelloch 1990]. In graphics, Horn recently used scan to implement an $O(n \log n)$ stream compaction primitive [2005]. The logarithmic forward-propagation-back-propagation structure of our cyclic reduction is also a type of scan; our implementation runs in $O(n)$ time.

4.2 Algorithm Implementation

Since we are using an alternating direction solver, our implementation will first solve for all rows and then use the results to solve all columns in parallel. Our algorithm requires four steps, all of which exploit the parallelism of the GPU: construct tridiagonal matrices for each row in parallel, solve the systems of matrices on each row in parallel, then repeat those two steps on the columns. In the discussion below, we use row terminology, but other than a necessary transpose, the procedure is the same for columns.

We begin by computing the tridiagonal matrix on the GPU. We can do so with a single GPU pass, computing all matrix rows in parallel; each row only needs the thermal conductivity and input coupling coefficient from itself and its immediate neighbors. At the end, for an $m \times n$ image, we have n $m \times m$ tridiagonal matrices, each corresponding to a row of the input image, stored as rows in a single $m \times n$ texture. We solve each of these n systems in parallel to produce n solutions to the 1D heat diffusion equation, each solution corresponding to a row of the input. For clarity below, we describe only the solution of a single row.

```
1:  for L = 1...log2(N+1) - 1 do
2:      for j = 0...2^-j (N+1) - 2 do
3:          α ← a^{L-1}_{2j+1} / b^{L-1}_{2j}
4:          γ ← c^{L-1}_{2j+1} / b^{L-1}_{2j+2}
5:          a^L_j ← -(α a^{L-1}_{2j})
6:          b^L_j ← b^{L-1}_{2j+1} - (α c^{L-1}_{2j} + γ a^{L-1}_{2j+2})
7:          c^L_j ← -(γ c^{L-1}_{2j+2})
8:          y^L_j ← y^{L-1}_{2j+1} - (α y^{L-1}_{2j} + γ y^{L-1}_{2j+2})
9:  y^{M-1}_0 ← y^{M-1}_0 / b^{M-1}_0
10: for L = log2(N+1) - 2...0 do
11:     for j = 0...2^-j (N+1) - 2 do
12:         j_p ← j/2
13:         if j is odd then
14:             y^L_j ← y^{L+1}_{j_p}
15:         else
16:             y^L_j ← (y^L_j - c^L_j y^{L+1}_{j_p} - a^L_j y^{L+1}_{j_p-1}) / b^L_j
```

Figure 5: Pseudocode for our GPU-compatible cyclic reduction tridiagonal solver. a^L_j, b^L_j, and c^L_j refer to the matrix entries for row j in level L of the hierarchy of solutions, and y^L_j is an element of the solution vector. Note that the above forward and backward substitution computations are both parallelizable and rely only on gather memory accesses.

As we noted in Section 3, LU decomposition is the traditional method for solving a tridiagonal system. Unfortunately, each step in the forward and back substitutions of a LU decomposition relies on the previous step and hence cannot be parallelized. Instead, we use the method of *cyclic reduction* [Hockney 1965; Karniadakis and Kirby II 2003], a parallel-friendly algorithm often used on vector computers for solving tridiagonal systems, as a basis for our implementation.

Cyclic reduction works by recursively using Gaussian elimination on all the odd-numbered unknowns in parallel. Figure 5 contains pseudocode for the cyclic reduction algorithm. During elimination, each of the odd-numbered unknowns is expressed in terms of its neighboring even-numbered unknowns, resulting in a partial solution and a new system, each with half the number of equations (Figure 5, lines 1–8). The process is repeated for $\log m$ steps until only one equation remains (Figure 5, line 9) along with a hierarchy of partial solutions to the system. Next, the solution to this equation is fed back into the partial solutions, and after $\log m$ steps to propagate the known results into the partial solutions (Figure 5, lines 10–16), the system is solved. While cyclic reduction requires more arithmetic than an LU solver, it still takes only a constant time per unknown and is amenable to an efficient GPU implementation.

In our implementation, in each row, each pixel is associated with one input element as well as to one row of the tridiagonal matrix. In the forward propagation step, a pass that begins with m unknowns will produce a new system with $m/2$ unknowns; because each new system produces two output matrices of half the size, we allocate a pyramid of textures at the outset of our computation, requiring an aggregate additional amount of storage equal to twice the size of the original tridiagonal matrix texture. We also refactor the traditional description of cyclic reduction so that the computation of an output

element k requires data from input elements $2k-1$, $2k$, and $2k+1$. Expressing our computation in this way enables the GPU to run the same program on every pixel, enabling high performance. Just as important is that it also allows the GPU to leverage its ability to read random memory locations (gather) without requiring writes to random memory locations (scatter).

5 Results

Figures 1 and 2 show several images generated with our DOF algorithm; the inputs to our algorithm were input images with color and depth information at every pixel. Figure 6 shows a series of images generated with the 3-layer variant of our algorithm, where we composite the near-field fence atop the mid- and far-range flags and mountains. These images are also part of the accompanying video.

5.1 Runtime and Analysis

We implemented our DOF system on a 2.4 GHz Athlon 64 FX-53 system running Windows XP with an NVIDIA GeForce 7800 GPU. Our implementation has image-space complexity so its runtime is strictly a function of the image size. Using just the background and midground layers, on a 256×256 image, we sustain 80–90 frames per second; on 512×512 image, we sustain 21–23 frames per second; and on a 1024×1024 image, we sustain 6–7 frames per second. At 1k by 768 resolution, with the separate foreground layer added for the flag images in the accompanying video, the frame rate drops to 3–4 frames per second. Note that the flag sequence in the video was computed and recorded at 1k by 768 resolution, and later downsampled to keep the video size reasonable. In all cases the performance scales approximately linearly with the number of pixels.

The performance of our algorithm is suitable for use in high-quality film preview applications such as those we target with our work, and we expect that further improvements in next-generation GPUs will soon allow this technique to be used in real-time entertainment applications such as games. The running time of our algorithm is limited by the performance of the fragment program that implements the tridiagonal solver. In general, increases in either the speed or the number of fragment units on graphics hardware will directly scale the performance of our system.

5.2 Limitations

As we discussed in Section 3, our ADI solution of the heat equation yields solutions that approximate a Gaussian point-spread function across the circle of confusion. The technical term for the point-spread function in this context is "bokeh"; choosing other distributions corresponds to different lens effects. Because the Green's function of the diffusion equation is a Gaussian, generating other distributions would be problematic for our technique. Fortunately, the Gaussian distribution is a physically meaningful and interesting one; Buhler and Wexler indicate that such a distribution of light produces a "smooth" or "creamy" effect "similar to a Leica lens" [2002].

Our ADI solution breaks up the computation into separate horizontal and vertical passes that could potentially produce anisotropies. Any issues with anisotropies could be mitigated by taking multiple diffusion steps; we have not seen any severe anisotropic effects in practice.

6 Conclusion

In this work, we have introduced a new depth-of-field post-process algorithm that uses a heat diffusion formulation to calculate accurate DOF effects at real-time rates. Unlike previous methods, our algorithm achieves high quality and interactive speed at the same time. It properly handles boundaries between in-focus and out-of-focus regions, while attaining interactive frame rates and constant computation time per pixel. Our implementation of the algorithm also introduces the use of cyclic reduction to the GPU world using GPU-friendly gather memory patterns. The real-time performance of our system makes it suitable today for interactive film preview, and continued advances in the performance of graphics hardware will likely also make it attractive soon for games and other real-time applications.

References

BARAFF, D., ANDERSON, J., KASS, M., AND WITKIN, A. 2003. Physically based modelling. *ACM SIGGRAPH Course Notes* (July).

BLELLOCH, G. E. 1990. Prefix sums and their applications. Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov.

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖDER, P. 2003. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics 22*, 3 (July), 917–924.

BUHLER, J., AND WEXLER, D. 2002. A phenomenological model for bokeh rendering. In *ACM SIGGRAPH 2002 Conference Abstracts and Applications*. `http://www.flarg.com/Graphics/Bokeh.html`.

BURT, P. J., AND ADELSON, E. H. 1983. A multiresolution spline with application to image mosaics. *ACM Transactions on Graphics 2*, 4 (Oct.), 217–236.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, vol. 18, 137–145.

DEMERS, J. 2004. Depth of field: A survey of techniques. In *GPU Gems*, R. Fernando, Ed. Addison Wesley, Mar., ch. 23, 375–390.

GALOPPO, N., GOVINDARAJU, N. K., HENSON, M., AND MANOCHA, D. 2005. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 3.

GOODNIGHT, N., WOOLLEY, C., LEWIN, G., LUEBKE, D., AND HUMPHREYS, G. 2003. A multigrid solver for boundary value problems using programmable graphics hardware. In *Graphics Hardware 2003*, 102–111.

HAEBERLI, P. E., AND AKELEY, K. 1990. The accumulation buffer: Hardware support for high-quality rendering. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, vol. 24, 309–318.

HARRIS, M. 2005. Mapping computational concepts to GPUs. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, Mar., ch. 31, 493–508.

Figure 6: The image at the top left is an in-focus view of a scene that we render with our 3-layer DOF algorithm. Clockwise from top right, we show the results of our algorithm with cameras with a narrow aperture and a mid-distance focal plane; with a wide aperture and a distant focal plane; and with a wide aperture and a near focal plane.

HILLESLAND, K. E., MOLINOV, S., AND GRZESZCZUK, R. 2003. Nonlinear optimization framework for image-based modeling on programmable graphics hardware. *ACM Transactions on Graphics 22*, 3 (July), 925–934.

HOCKNEY, R. W. 1965. A fast direct solution of Poisson's equation using Fourier analysis. *Journal of the ACM 12*, 1 (Jan.), 95–113.

HORN, D. 2005. Stream reduction operations for GPGPU applications. In *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, Mar., ch. 36, 573–589.

KARNIADAKIS, G. E., AND KIRBY II, R. M. 2003. *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implementation*. Cambridge University Press, ch. 9, 455–537.

KASS, M., AND MILLER, G. 1990. Rapid, stable fluid dynamics for computer graphics. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, vol. 24, 49–57.

KŘIVÁNEK, J., ŽÁRA, J., AND BOUATOUCH, K. 2003. Fast depth of field rendering with surface splatting. In *Computer Graphics International*, 196–201.

KRÜGER, J., AND WESTERMANN, R. 2003. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics 22*, 3 (July), 908–916.

POTMESIL, M., AND CHAKRAVARTY, I. 1981. A lens and aperture camera model for synthetic image generation. In *Computer Graphics (Proceedings of SIGGRAPH 81)*, vol. 15, 297–305.

PRESS, W. H., FLANNERY, B. P., TEUKOLSKY, S. A., AND VETTERLING, W. T. 1992. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press.

RIGUER, G., TATARCHUK, N., AND ISIDORO, J. 2003. Real-time depth of field simulation. In *ShaderX^2: Shader Programming Tips and Tricks with DirectX 9*, W. F. Engel, Ed. Wordware, ch. 4.7, 529–556.

SCHEUERMANN, T., AND TATARCHUK, N. 2004. Improved depth-of-field rendering. In *ShaderX^3: Advanced Rendering with DirectX and OpenGL*, W. Engel, Ed. Charles River Media, ch. 4.4, 363–377.

WILLIAMS, L. 1983. Pyramidal parametrics. In *Computer Graphics (Proceedings of SIGGRAPH 83)*, vol. 17, 1–11.

YU, T.-T. 2004. Depth of field implementation with OpenGL. *Journal of Computing Sciences in Colleges 20*, 1 (Oct.), 136–146.