ROCKY MOUNTAIN
JOURNAL OF MATHEMATICS
Volume 41, Number 2, 2011

GPU-BASED METHODS FOR EXPLORING PARABOLIC PARTIAL DIFFERENTIAL EQUATIONS

DAVID EBERLY AND JERROLD BEBERNES

ABSTRACT. The mathematical investigation of solutions to a parabolic partial differential equation (PDE) can be complemented by numerically computing the solutions in order to obtain insight about their qualitative structure. The numerical methods are relatively slow on central processing units (CPUs), making it difficult to obtain rapid feedback about the solution over time. Graphics processing units (GPUs) are efficient hardware for massively parallel computations and provide rapid feedback. The numerical methods for parabolic PDEs map naturally onto the GPU but differ from the same methods implemented on a CPU. We describe the ideas for GPU-based solvers and illustrate them using parabolic PDEs that arise in combustion models.

1. Introduction. The concept of using numerical solutions to PDEs in order to gain insight about the qualitative structure of the theoretical solutions is not new, of course. In fact, several of the results mentioned in [**2**] were motivated by numerical experiments; in particular, they motivated the study of solution profiles and the relationship of their shape to bifurcation diagrams [**3**] and for developing generalized maximum principles [**1, 7**].

At that time (25 years ago), the experiments were performed on hardware that included an Intel 80486 CPU with an 80487 floating-point coprocessor and an Enhanced Graphics Adapter card. By today's standards, such hardware might as well be displayed in a museum about ancient computing devices. Back then it was enough to give some idea about solutions to the PDEs but was not suitable for visualization in interactive time let alone real-time.

Current generation hardware is much different from that of 25 years ago. CPUs now have multiple units of execution called cores. GPUs were invented to support the demands of consumers wanting more realistic graphics in computer video games. Although the evolution of CPUs and GPUs has been driven by consumer entertainment,

Received by the editors on August 21, 2010.
DOI:10.1216/RMJ-2011-41-2-457

an important consequence is that the processors provide significant power for scientific computation. The GPUs support massively parallel computations and are naturally suited for the numerical solution of parabolic PDEs. This paper shows how to map the mathematical algorithms onto the GPU and mentions the main differences between how you solve the PDEs on a GPU compared to on a CPU.

The ideas are illustrated using a nonlinear parabolic PDE that arises in combustion models and has solutions that blow up in finite time,

$$\begin{aligned} u_t &= \Delta u + \lambda e^u \left(\int_\Omega e^u \, dx \right)^{-p}, \quad (x,t) \in \Omega \times (0,\infty), \\ u(x,0) &= I(x), \quad x \in \Omega, \\ u(x,t) &= 0, \quad (x,t) \in \partial\Omega \times (0,\infty) \end{aligned} \tag{1}$$

where Ω is a compact set and $I(x) \in L^\infty(\Omega) \cap C(\overline{\Omega})$. The scalar λ is positive and chosen to be 1 for the numerical experiments. The problem is *local* when $p = 0$ and *nonlocal* when $p > 0$. The numerical experiments involved p-values of 0, 0.5, 0.99 and 1.

2. CPU-based method for dimension 1. The domain is chosen to be $\Omega = [-1, 1]$, and initially we look at the local problem ($p = 0$). The Crank-Nicholson method is used for the numerical solver, where a forward finite difference is used to estimate u_t. A central finite difference is used to estimate u_{xx} but involves the to-be computed time rather than the previous time. This leads to an implicit equation that has good numerical stability. Choose n spatial samples on $[-1, 1]$; that is, $\Delta_x = 2/(n-1)$ and $x_i = -1 + 2i/(n-1)$ for $0 \le i \le n-1$. The time step is Δ_t with $t_j = j\Delta_t$. The estimates are $u_i^j = u(x_i, t_j)$ and the numerical method to compute them is

$$u_i^{j+1} = a u_i^j + b_0 \left(u_{i+1}^{j+1} + u_{i-1}^{j+1} \right) + c \exp\left(u_i^j \right) \tag{2}$$

for $0 < i < n-1$, $j \ge 0$, $r = \Delta_t/\Delta_x^2$, $a = 1/(1+2r)$, $b = ar$, $c = a\Delta_t$, and with specified initial values u_i^0. The boundary values are always $u_0^j = u_{n-1}^j = 0$.

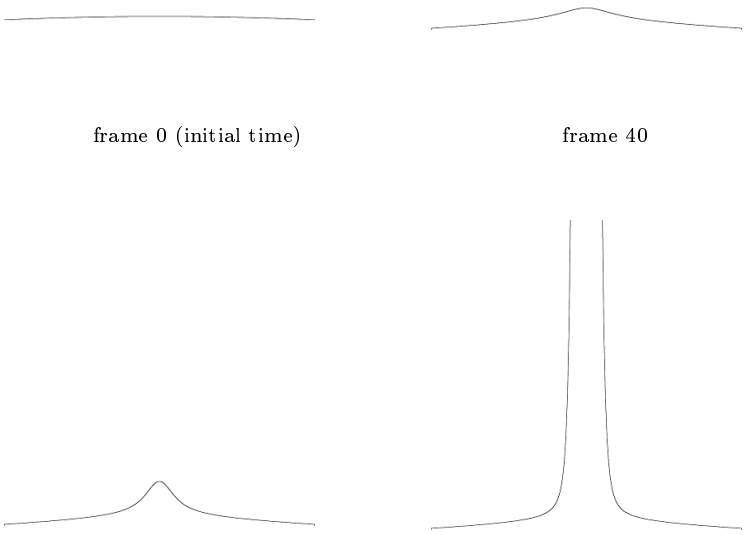

FIGURE 1. Solution profiles for local equation (1) in dimension 1.

For each time frame j, equation (2) must be iterated a specified number of times until convergence criteria are met. This is referred to as *Gauss-Seidel iteration* and is the method of choice for real-time physics simulations in modern video games. Flip-flop buffers are used to avoid expensive memory allocations and deallocations. Pseudocode for the numerical solver is listed next. The termination condition is that at least one solution value is infinite relative to the floating-point system.

```
void SolveLocalBlowup1 ()
{
int n = 1024, numGaussSeidel = 8;
float dx = 1, dt = 0.01, r = dt/(dx*dx), a = 1/(1+2*r), b = a*r, c = a*dt;
Buffer1⟨float⟩ u0(n), u1(n);
```

```
SetInitialBoundaryValues(u0, u1);
do_forever {
for (int j = 0; j < numGaussSeidel; ++j) {
for (int i = 1; i < n−1; ++i) {
u1(i) = a*u0(i) + b*(u1(i+1) + u1(i−1)) + c*exp(u0(i)));
}
}
if (some value of u1() is not finite) { return; }
SwapBuffers(u0,u1);
}
```

The initial function in the experiments was $u(x,0) = 1 - x^2$. The loop executed for 43 frames before exiting. The solution profiles for several frames are shown in Figure 1. The horizontal axis corresponds to $x \in [-1,1]$ and the vertical axis corresponds to $u \in [0,100]$.

The numerical implementation for the local problem is straightforward with no surprises. On the other hand, the implementation for the nonlocal problem requires some attention, whether implemented on a CPU or GPU. The finite difference equation in the pseudocode may be replaced by

```
u1(i) = a*u0(i) + b*(u1(i+1) + u1(i-1)) + c*exp(u0(i))/Integral(p,u0);
```

where `Integral(p,u0)` computes a numerical approximation to the pth power of the integral in equation (1). Because of the Gauss-Seidel iterations, the evaluation of `Integral` should be moved outside the Gauss-Seidel loop to avoid redundant calculations. The classical approach is to use the trapezoidal rule for integration, which involves a summation of terms $\exp(u_i)$. There are two problems, one obvious and one not so obvious.

The obvious problem is that, near the blow-up time, the values u_i become large and the exponentials of these even larger. Floating-point overflow is a certainty when summing the terms, potentially making it appear that there is blow up when theoretically there is not. It is better to transform the problem slightly by searching for the maximum m of all u_i and estimating the integral of $\exp(u(x,\cdot)-m)$. The terms in the summation of the trapezoidal rule are now $\exp(u_i - m)$, which are guaranteed to be no larger than 1. In worst case, u_i is small compared to m so that $\exp(u_i - m)$ exhibits gradual underflow or is flushed to zero, depending on how the floating-point unit is configured; this is

preferable to the overflow. The function `Integral` returns the pth power of the estimate of the modified integral *and* returns the value m. The latter value is used in the finite difference equation.

```
float u0max; // computed by Integral
float integral = Integral(p,u0,u0max);
float d0 = c/integral, d1 = p*u0max;
for (int j = 0; j < numGaussSeidel; ++j) {
for (int i = 1; i < n-1; ++i) {
u1(i) = a*u0(i) + b*(u1(i+1) + u1(i-1)) + d0*exp(u0(i)−d1);
}
}
```

The nonobvious problem might occur when the number of spatial samples is large. In worst case, the summation of a large number of large floating-point numbers can overflow. More frequent is that the partial sums are floating-point numbers in a sparsely populated region of the floating-point system, in which case numerical round-off errors accumulate significantly. Although usually not of consequence for the one-dimensional problem, it is an issue for higher dimensions when dense spatial sampling is used. An implementation of the trapezoidal rule may formulate the result in terms of the average value of the function. For the problem at hand, the trapezoidal rule to estimate $I = \int_{-1}^{1} \exp\left(u(x,\cdot) - m\right) dx$ is

$$\begin{aligned} I &\doteq \frac{2n}{n-1}\left(\frac{1}{n}\sum_{i=0}^{n} \exp\left(u_i - m\right)\right) - \frac{2}{n-1}\exp\left(-m\right) \\ &= \frac{2n}{n-1}A - \frac{2}{n-1}\exp\left(-m\right) \end{aligned} \tag{3}$$

where $\Delta_x = 2/(n-1)$ and A is an average of the n function samples.

The average A may be computed in a manner related to *mipmapping* that is used in computer graphics for producing a multiresolution pyramid of texture images [**15**]. Assuming n is a power of two, pairs of consecutive terms in the summation are averaged, leading to an array of numbers half the length of the original array. The process is repeated on this array. The repetition ends when the final array has one element that must be A. Assuming the original terms are finite floating-point numbers, the average of pairs are guaranteed to be finite floating-point numbers, albeit with potential round-off errors. The final result, A,

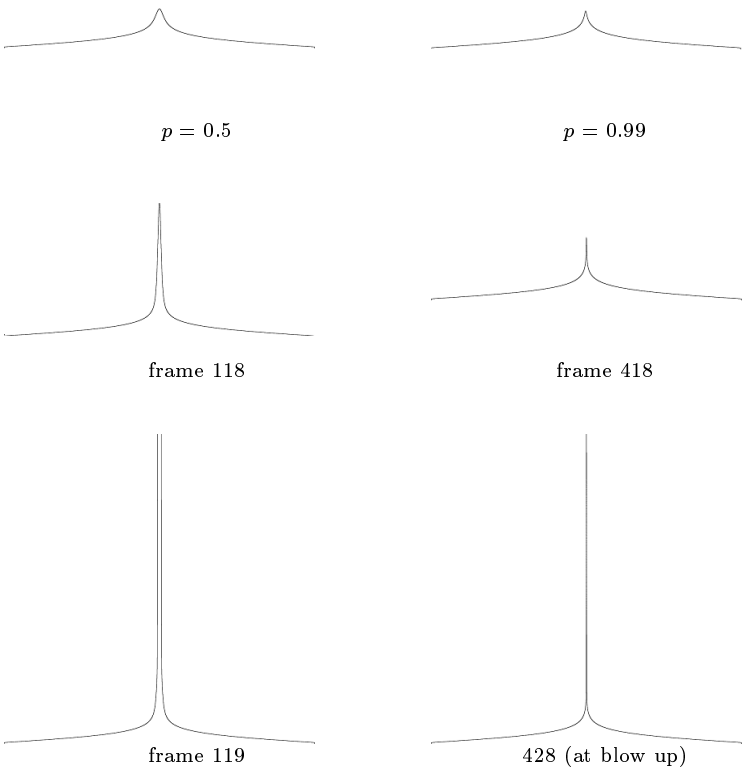

FIGURE 2. Solution profiles for nonlocal equation (1) in dimension 1.

is guaranteed to be a finite floating-point number. Moreover, given that the terms are consecutive samples from a continuous function, the numbers in each pair are relatively close, which avoids the typical floating-point problem of summation of a large and a small number whereby the small number is effectively treated as zero due to bit-shifting to match exponents.

With the modifications for robustly computing the integral and Gauss-Seidel iterates, and with the same initial function $u(x, 0) = 1 - x^2$, the nonlocal problem was run with values $p = 1$, $p = 0.99$

and $p = 0.5$. For $p = 1$, the solution gradually increased through large values at $x = 0$, but blow up did not occur. For $p = 0.99$, the loop executed for 470 frames to reach blow up. For $p = 0.5$, the loop executed for 121 frames to reach blow up. The solution profiles for several frames are shown in Figure 2.

3. Methods for Dimension 2. Numerical experiments were performed for dimension 2. The domain was $\Omega = [-1, 1]^2$ and the initial values were $I(x, y) = (1-x^2)(1-y^2)$. The same approach to computing the integral in the nonlocal problem was used, the integrals estimated by the following where n_d is the number of samples in dimension d: $I \doteq k_0 A - k_1 \exp(-m)$ where $k_0 = 4n_0n_1/\delta$ and $k_1 = 4(n_0 + n_1 - 1)/\delta$ with $\delta = (n_0-1)(n_1-1)$ The number A is the average of the exponential values computed using the mipmapping approach for robustness. The Gauss-Seidel iterates are

```
// 2D local problem, per Gauss-Seidel iterate
for (i1 = 1; i1 < n1−1; ++i1)
{
for (i0 = 1; i0 < n0−1; ++i1)
{
u1(i0,i1) = a*u0(i0,i1) + b0*(u1(i0+1,i1) + u1(i0−1,i1)) +
b1*(u1(i0,i1+1) + u1(i0,i1−1)) + c*exp(u0(i0,i1)));
}
}
// 2D nonlocal problem, per Gauss-Seidel iterate
for (i1 = 1; i1 < n1−1; ++i1)
{
for (i0 = 1; i0 < n0−1; ++i1)
{
u1(i0,i1) = a*u0(i0,i1) + b0*(u1(i0+1,i1) + u1(i0−1,i1)) +
b1*(u1(i0,i1+1) + u1(i0,i1−1)) + d0*exp(u0(i0,i1)−d1));
}
}
```

for appropriately chosen constants a, b_0, b_1, b_2, c, d_0 and d_1.

3.1. CPU method for dimension 2. The loops may be implemented as-is on the CPU, so the code is very similar to what was shown

for dimension 1. The solution profiles are qualitatively similar to those shown in Figures 1 and 2. Observe that the array values $u_1(\cdot,\cdot)$ are both read from memory (on the right-hand side of the equation) and written to memory (on the left-hand side of the equation). Read-write access of this form is common in programs written for a CPU.

3.2. GPU method for dimension 2. On the GPU, each calculation per (i_0, i_1) is performed by a *pixel shader*, which is a program written in a shading language similar to the C language (we used OpenGL and it associated GLSL language). The arrays on the right-hand side are accessed as *textures* provided to the pixel shader. The left-hand side is the return value of the pixel shader and is written to a texture managed by a *framebuffer object* that is associated with the pixel shader. Within a shader, the textures are read-only and the framebuffer texture is write-only, so the finite difference equation inside the double loop cannot be realized in a pixel shader. Therefore, the algorithm must be modified to allow GPU computations. The new approach requires three buffers but the numerical solution is nearly indistinguishable from that computed on the CPU, because the Gauss-Seidel iterations converge theoretically to the same values. For example, the local problem on the GPU is effectively

```
Texture u[3]; // all n0–by–n1 buffers
SetInitialBoundaryValues(u[0],u[1],u[2]);
TextureReference T[3];
int input0 = 0;
T[0] = u[input0];
for (j = 0; j < numGaussSeidel; ++j)
{
// Toggle inputs between (T[0],T[1]) and (T[1],T[0]).
int input1 = 1 − input0;
T[1] = u[input1];
int output = 2;
T[2] = u[2];
for (i1 = 1; i1 < n1−1; ++i1)
{
for (i0 = 1; i0 < n0−1; ++i1)
{
T2(i0,i1) = a*T0(i0,i1) + b0*(T1(i0+1,i1) + T1(i0−1,i1)) +
```

```
b1*(T1(i0,i1+1) + u1(T1,i1−1)) + c*exp(T0(i0,i1)));
}
}
// Toggle outputs between (T[1],T[2]) and (T[2],T[1]) or between
// (T[0],T[2]) and (T[2],T[0]).
int save = input1; input1 = output; output = save;
}
```

In this pseudocode, the inner loops compute values on the interior of the domain. In the real application, the pixel shader is executed for all values, interior and boundary. The shader can use branching to test if the current (i_0, i_1) values are on the boundary, setting the output to zero in this case. However, branching is expensive and not even supported on early generation GPUs. The branching can be avoided by having an additional *mask* texture whose values are 1 at interior points and 0 at boundary points. The output values are computed whether the incoming (i_0, i_1) is interior or boundary, but then the values are multiplied by the mask-texture values, thus causing all boundary values to be zero.

The details of setting up the graphics system are important but are more about software engineering than mathematics. They are not presented here. The source code is available online [**6**] and works on Microsoft Windows (XP/Vista/7), Macintosh OS X, and Linux as long as the graphics card supports OpenGL 2 or later. The distribution includes installation and release notes.

To compare execution times, we chose $n_0 = n_1 = 1024$ and measured the execution time for the Gauss-Seidel iteration; that is, we excluded the time for initializing and terminating the graphics objects. The local problem required 16.911 seconds on a 3 GHz CPU but only 0.656 seconds on an NVIDIA 9800 GT GPU. The nonlocal problem required 56.862 seconds on the CPU but only 7.925 seconds on the GPU.

The implementation for the nonlocal problem reads back the frame-buffer texture and estimates the integral using computations on the CPU. The read-back is a transfer from video memory to system memory, a process that is a known bottleneck on GPUs. With more shader programming effort, the read-back can be avoided. The maximum of the current u-values can be computed on the GPU using a pyramidal approach, just as shown here for the average of the exponential of the

samples (minus the maximum). Once the maximum is known, the average may also be computed on the GPU using a pyramidal approach.

Similarly, the implementations for both the local and nonlocal problems read back the framebuffer texture in order to locate any buffer values that are infinite. When an infinite value is found, the numerical method terminates. The read-back here can also be avoided by searching for an infinite value using shader programming and a pyramidal approach.

Finally, the implementations used the read-back so that an x-slice of the solution could be used to generate bitmap images that show the graph of the solution for the chosen value of x. As an alternate visualization, the framebuffer texture does not have to be read back. Instead, it can be used as a *displacement map* for a *vertex shader* to generate a triangle mesh that approximates the graph of the solution and that can be displayed using a 3D graphics rendering system. Thus, all read-backs can be avoided for optimum performance and the results can be visualized and allow interaction such as display of the mesh as if it were embedded in a virtual trackball. The source code distribution [**6**] has implementations that do this.

4. Methods for dimension 3. Numerical experiments were also performed for dimension 3. The domain was $\Omega = [-1,1]^3$, and the initial values were $I(x,y,z) = (1-x^2)(1-y^2)(1-z^2)$. The integral estimate for the nonlocal problem is $I \doteq k_0 A - k_1 \exp(-m)$ where $k_0 = 8n_0n_1n_2/\delta$ and $k_1 = 8(n_0(n_1-1)+n_1(n_2-1)+n_2(n_0-1)+1)/\delta$ with $\delta = (n_0-1)(n_1-1)(n_2-1)$. The number A is the average of the exponential values computed using the mipmapping approach for robustness. Each Gauss-Seidel iterate encapsulates a triple loop whose inner-most expression is the finite difference equation.

The CPU implementation for dimension 3 is very similar to that for dimension 2. Both the local and nonlocal solutions have profiles similar to those of Figures 1 and 2.

The GPU implementation for dimension 3 requires more effort than for dimension 2. The GPU is designed so that the pixel shaders write to a two-dimensional framebuffer object that manages a two-dimensional texture. There is no concept of a three-dimensional framebuffer object. On the CPU, the triple loop is implemented to access a three-

dimensional array $u(x, y, z)$ for which memory is stored in lexicographical order. If i is the 1-dimensional index into the contiguous block of memory that stores the array, and if the array has dimensions n_0, n_1, and n_2, then the mapping between three-dimensional array and one-dimensional memory is

$$i = x + n_0(y + n_1 z), \tag{4}$$

where the x, y and z values are integers with $0 \leq x < n_0$, $0 \leq y < n_1$, and $0 \leq z < n_2$. You may think of this visually as a stack of n_2 tiles, each tile of size $n_0 \times n_1$. Within a tile, it is easy to compute the six immediate neighbors of (x, y, z), namely, $(x \pm 1, y, z)$, $(x, y \pm 1, z)$, and $(x, y, z \pm 1)$. These index neighbors are used in the finite difference equation to look up the corresponding u-values.

On the GPU, the three-dimensional output must be stored as a two-dimensional block of memory. This is accomplished by storing the xy-tiles as a two-dimensional array of tiles. They may be stored in row-major order so that the $z = 0$ tile is in the first row and first column, the $z = 1$ tile is in the first row and second column, and so on. Only some of the tiles fit on the first row, so the remaining tiles are stored in the second row, the third row, and so on.

If (x, y, z) is an interior point of a tile, its xy-neighbors are the obvious ones: $(x \pm 1, y \pm 1, z)$. The z-neighbors live in two other tiles in the array of tiles. This information is needed by the pixel shader to look up the neighboring u-texture values. The easiest way to handle the look up is to have a two-dimensional *offset texture* that is made available to the shader. Given an (x, y) index into the two-dimensional texture that represents the array of tiles, the (x, y) values of the z-neighbors are looked up in the offset texture. The u-texture values are then looked up using these offsets.

As in the two-dimensional GPU solver, the pixel shader must assign zero to the boundary points of the domain. When the three-dimensional output is stored as an array of tiles, the boundary points of the tiled array are boundary points of the three-dimensional output. However, some of the interior points of the tiled array are also boundary points of the three-dimensional output. Just as in the two-dimensional GPU solver, a *mask texture* is used whose values are 1 at tiled-array points corresponding to the interior points of the three-dimensional output and are 0 elsewhere.

Again, a large portion of the details of implementing the GPU-based solver are in the realm of software engineering, not mathematics, so the details are not presented here. The source code distribution [**6**] contains implementations for the three-dimensional blow-up problems.

5. Variations of the GPU-based solvers. The numerical experiments described here were for Ω which is an interval in dimension 1 and a Cartesian product of intervals in higher dimensions. Although a lot of research results have been developed for *convex* Ω, less is known about the qualitative behavior of solutions when the domains are not convex. The GPU-based solvers may be extended in a simple manner to allow experimentation with nonconvex domains. It is sufficient to embed Ω in a Cartesian product of intervals. A corresponding *domain mask texture* may be provided to the pixel shader. This mask is a binary image that is 1 for pixels in Ω and 0 for pixels not in Ω. The look ups of u-values for the finite differences can be clamped using an appropriately initialized offset texture. The outputs computed by the pixel shader are multiplied by the domain mask values.

The initial functions in the numerical experiments were even functions in their components. However, the textures corresponding to the initial data can be whatever you want. For example, if you want to explore blow up at a point not at the origin or at multiple points, you can vary the initial data to try to make this happen.

The main limitation of GPU-based methods is the size of the output textures. Graphics drivers impose limits on the size; for example, the NVIDIA 9800 GT GPU on which the experiments were run has a limitation of 4096 per dimension. The largest two-dimensional output supported for the experiments is 4096×4096 using a 1-channel texture whose component is 32-bit floating-point. To have more dense sampling, a significant amount of systems/software engineering must take place to decompose the domain and solve the PDE piecewise. This is tedious but tractable.

6. GPU-based solvers for the Navier-Stokes equation. For many years, the real-time video game industry has had an interest in simulation of fluids and gases; for example, [**8**–**14**].

Of particular interest related to this paper, [**14**] discusses equations for conservation of mass and conservation of momentum (Navier-Stokes equation). The focus is on producing a *believable* simulation, not on obtaining numerically accurate solutions, but there is nothing inherent in the presentation that prevents one from implementing simulations that produce accurate solutions. The conservation equations are stated in the paper without derivation and with few details about the assumptions made to produce them. The paper also discusses only a CPU-based numerical method. From a practical perspective, the important part of this paper is showing how to deal with diffusion, advection, and the nonlinearity when solving the equations numerically.

A somewhat elementary but detailed pedagogic derivation and discussion about various simplifying assumptions are presented in [**4**]. Moreover, the book has discussions about the mapping of the Navier-Stokes equation onto the GPU, both for the two-dimensional and three-dimensional problems. The system-engineering details are significant, but at the core are the concepts mentioned in this paper about use of shaders, Gauss-Seidel iterations, memory mapping, mask textures and offset textures. The source code is available from the Geometric Tools website (`http://www.geometrictools.com/`). Screen captures and a flow chart for a GPU-based two-dimensional simulation of the Navier-Stokes equation used in [**14**] is available online [**5**].

REFERENCES

1. J. Bebernes and D. Eberly, *A description of self-similar blowup for the solid fuel ignition model*, Indiana Math. J. **36** (1987), 295–305.

2. ———, *Mathematical problems from combustion theory*, Appl. Math. Sci. **83**, Springer-Verlag, New York, 1989.

3. J. Bebernes, D. Eberly and W. Fulks, *Solution profiles for some simple combustion models*, Nonlinear Analysisn–Theory, Methods and Applications **10** (1986), 165–177.

4. D. Eberly, *Game physics*, 2nd edition, Morgan Kaufmann (an imprint of Elsevier), Burlington, MA, 2010.

5. ———, GPU*-based numerical solution of the* 2D *Navier-Stokes equation*, `http://www.geometrictools.com/SamplePhysics/GpuFluids2D/GpuFluids2D.html`, 2010.

6. ———, *Source code for* GPU*-based numerical solution of parabolic* PDE*s*, `http://www.geometrictools.com/RMMJ/GpuPdeSolvers.zip`, 2010.

7. D. Eberly and W. Troy, *On the existence of logarithmic-type solutions to the Kassoy-Kapila problem in dimensions* $3 \leq n \leq 9$, J. Diff. Equations **80** (1987), 309–324.

8. R. Fedkiw, J. Stam and H.W. Jensen, *Visualization of smoke*, in Proceedings of SIGGRAPH 2001, 15–22, 2001.

9. N. Foster and R. Fedkiw, *Practical animation of liquids*, in Proceedings of SIGGRAPH 2001, 23–30, 2001.

10. Mark Harris, CUDA *fluid simulation in* NVIDIA PhysX, `http://sa08.idav.ucdavis.edu/CUDA_physx_fluids.Harris.pdf`, 2008.

11. D.Q. Nguyen, R. Fedkiw and H.W. Jensen, *Physically based modeling and animation of fire*, in Proceedings of SIGGRAPH 2002, 721–728, 2002.

12. Hagit Schechter and Robert Bridson, *Evolving sub-grid turbulence for smoke animation*, in Proceedings of the 2008 ACM/Eurographics Symposium on Computer Animation, 2008.

13. Andrew Selle, Nick Rasmussen and Ronald Fedkiw, *A vortex particle method for smoke, water and explosions*, in Proceedings of SIGGRAPH 2005, 910–914, 2005.

14. Jos Stam, *Real-time fluid dynamics for games*, in Proceedings of the Game Developer Conference, March 2003, 2003.

15. L. Williams, *Pyramidial parametrics*, Computer Graphics **7** (1983), 1–11.

Geometric Tools, LLC, 5911 East Spring Road, Scottsdale, AZ 85254-5548
Email address: deberly@geometrictools.com

Department of Applied Mathematics, 526 UCB, University of Colorado, Boulder, CO 80309-0526
Email address: Jerrold.Bebernes@Colorado.edu