Visual Simulation of Shallow-Water Waves

T.R. Hagen, J.M. Hjelmervik, K.–A. Lie, J.R. Natvig, M. Ofstad Henriksen

SINTEF ICT, Applied Math., P.O. Box 124 Blindern, NO-0314 Oslo, Norway

Abstract

A commodity-type graphics card (GPU) is used to simulate nonlinear water waves described by a system of balance laws called the shallow-water system. To solve this hyperbolic system we use explicit high-resolution central-upwind schemes, which are particularly well suited for exploiting the parallel processing power of the GPU. In fact, simulations on the GPU are found to run 7–15 times faster than on a CPU. The simulated cases involve dry-bed zones and non-trivial bottom topographies, which are real challenges to the robustness and accuracy of the discretisation.

Key words: Water waves, high-resolution schemes, graphics processing unit, parallel processing, physics-based visualization
PACS: 02.60.Cb, 02.70.Bf

1 Introduction

Free-surface flow over a variable bottom topography under the influence of gravity can be modelled by the shallow-water (or Saint–Venant) equations

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ -gh\frac{\partial B}{\partial x} \\ -gh\frac{\partial B}{\partial y} \end{bmatrix}, \tag{1}$$

which we write on short form as

$$Q_t + F(Q)_x + G(Q)_y = H(Q, \nabla B).$$

Here $B(x, y)$ is the bottom topography, $h(x, y, t)$ is the distance from the bottom to the (wavy) surface, $[\,u, v\,]$ is the depth-averaged velocity, and g is the gravitational acceleration. The shallow-water equations are derived from the depth-averaged incompressible Navier–Stokes equations for the case where the surface perturbation is much smaller than the typical horizontal length scale.

To compute solutions of (1) it is common to use high-resolution schemes [1] with explicit temporal discretisation. Such schemes have an obvious and natural parallelism in the sense that each grid cell can be processed independently of its neighbours and are therefore ideal candidates for implementation using data-based stream processing on a graphics processing unit (GPU). In this paper we shall compute and compare approximate solutions to (1) using the GPU and the CPU. We shall demonstrate that the GPU gives a speedup of one order of magnitude, while retaining sufficient accuracy to make the GPU interesting for qualitative and quantitative simulations. Similarly, since the GPU simulations are so fast, solution of the shallow-water equations can be used to create semi-realistic nonlinear wave effects in visual applications.

2 Numerical Methods

The system (1) has two key features that makes it difficult to solve numerically. First of all, the solution may contain discontinuities that correspond to breaking waves. Classical schemes will therefore typically either smear the discontinuous parts or introduce spurious oscillations that pollute computed solutions. A common approach is therefore to use high-resolution schemes [1]. Here we will use a semi-discrete finite-volume scheme [2] to ensure high order of approximation for smooth waves and sharp resolution of discontinuities without the creation of spurious oscillations.

The second difficulty with (1) is that this system of balance laws admits steady-state solutions in which the source terms are exactly balanced by nonzero flux gradients. Capturing such balances is a challenging task for any numerical scheme. Here we will use a well-balanced treatment of the source terms [3] to accurately resolve surfaces that are steady-state solutions or small perturbations thereof.

The finite-volume scheme is defined over a regular Cartesian grid with grid cells Ω_{ij} and seeks approximations in the form of cell-averages, $Q_{ij} = \frac{1}{|\Omega_{ij}|}\int_{\Omega_{ij}} Q$. The simplest possible scheme is the first-order Lax–Friedrichs scheme

$$\begin{aligned} Q_{ij}^{n+1} = {} & \frac{1}{4}\Big(Q_{i+1,j}^n + Q_{i-1,j}^n + Q_{i,j+1}^n + Q_{i,j-1}^n\Big) + \Delta t S_{ij}^n \\ & - \frac{\Delta t}{2\Delta x}\Big[F\big(Q_{i+1,j}^n\big) - F\big(Q_{i-1,j}^n\big)\Big] - \frac{\Delta t}{2\Delta y}\Big[G\big(Q_{i,j+1}^n\big) - G\big(Q_{i,j-1}^n\big)\Big]. \end{aligned} \tag{2}$$

This is a very robust scheme, which unfortunately gives excessive smearing of nonsmooth parts of the solution.

To obtain better accuracy for nonsmooth solutions, we introduce a high-resolution scheme based upon a semi-discrete formulation where the cell averages are evolved in time according to

$$\frac{dQ_{ij}}{dt} = -\Big(F_{i+1/2,j} - F_{i-1/2,j}\Big) - \Big(G_{i,j+1/2} - G_{i,j-1/2}\Big) + S_{ij}, \tag{3}$$

where the flux over the edges is approximated using a fourth-order Gaussian quadrature

$$\begin{aligned} F_{i+1/2,j}(t) &= \frac{1}{|\Omega_{ij}|}\int_{y_{j-1/2}}^{y_{j+1/2}} F\Big(Q(x_{i+1/2},y,t)\Big)\,dy \\ &\approx \frac{1}{2\Delta x}\Big[F\Big(Q\Big(x_{i+1/2}, y_j + \frac{\Delta y}{2\sqrt{3}}, t\Big)\Big) + F\Big(Q\Big(x_{i+1/2}, y_j - \frac{\Delta y}{2\sqrt{3}}, t\Big)\Big)\Big]. \end{aligned} \tag{4}$$

To evaluate the integrand, we start with the cell-averages Q_{ij} and *reconstruct* a function that is piecewise polynomial inside each grid cell, see [2,4]. Here we use a component-wise piecewise linear function:

$$\begin{aligned} Q_{ij}(x,y) = Q_{ij} &+ L\Big(Q_{i+1,j} - Q_{i,j}, Q_{i,j} - Q_{i-1,j}\Big)\frac{x - x_i}{\Delta x} \\ &+ L\Big(Q_{i,j+1} - Q_{i,j}, Q_{i,j} - Q_{i,j-1}\Big)\frac{y - y_j}{\Delta y}. \end{aligned} \tag{5}$$

where the limiter function L is given as $L(a,b) = \mathrm{MM}(a,b)$ or $L(a,b) = \mathrm{MM}(\theta a, \frac{1}{2}(a+b), \theta b)$ with

$$\mathrm{MM}(z_1,\ldots,z_n) = \begin{cases} \max_i z_i, & z_i < 0 \forall i, \\ \min_i z_i, & z_i > 0 \forall i, \\ 0, & \text{otherwise.} \end{cases}$$

In each integration point $(x_{i+1/2}, y_{j\pm\alpha})$, we thus have a left-sided and a right-sided point value, Q^L and Q^R. To compute the flux we could use the average of the flux evaluated at the two one-sided point values. However, to get a high-resolution scheme we use the central-upwind flux-function [2]

$$\begin{aligned} \mathcal{F}(Q^L, Q^R) &= \frac{a^+ F(Q^L) - a^- F(Q^R)}{a^+ - a^-} + \frac{a^+ a^-}{a^+ - a^-}(Q^R - Q^L), \\ a^+ &= \max\Big(0, \lambda^+(Q^L), \lambda^+(Q^R)\Big), \quad a^- = \min\Big(0, \lambda^-(Q^L), \lambda^-(Q^R)\Big), \end{aligned} \tag{6}$$

where $\lambda^{\pm}(Q) = u \pm \sqrt{gh}$ denotes the eigenvalues of dF/dQ.

The ordinary differential equations (3) are integrated using a second-order TVD Runge–Kutta method [5],

$$\begin{aligned} Q_{ij}^{(1)} &= Q_{ij}^n + \Delta t R_{ij}(Q^n), \\ Q_{ij}^{n+1} &= \frac{1}{2}Q_{ij}^n + \frac{1}{2}\Big[Q_{ij}^{(1)} + \Delta t R_{ij}(Q^{(1)})\Big], \end{aligned} \tag{7}$$

where R_{ij} denotes the right-hand side of (3). The time step is restricted by a CFL-condition, which states that disturbances can travel at most one half grid cell each time step, i.e., $\max(a^+, -a^-)\Delta t \leq \Delta x/2$ and similarly in y.

As noted above, (1) admits steady-state solutions where non-zero flux gradients are balanced by the topographical source terms, i.e.,

$$[hu^2 + \frac{1}{2}gh^2]_x + [huv]_y = -ghB_x,$$

and similarly in the y-direction. Many physically interesting phenomena are perturbations of steady states. To accurately compute the time evolution of such perturbations, we must ensure that our scheme does not produce errors of the same magnitude as the waves we want to resolve. We would therefore like the spatial truncation error to be zero at steady states. This is an important and challenging problem.

Lake-at-rest ($hu = hv = 0$ and $h + B = \text{Const}$) is one important family of steady states for (1). By reconstructing the surface elevation $w = h + B$ rather than the water depth h, and by a special choice of quadrature for the cell-averaged source term S_{ij}, the spatial part of the truncation error for the scheme vanishes at these states and the scheme preserves steady states numerically, see [3] for further details. The quadrature rule for the second component of the source term is given by

$$\begin{aligned} S_{ij}^{(2)} &= -\frac{1}{|\Omega_{ij}|}\int_{y_{j-1/2}}^{y_{j+1/2}}\int_{x_{i-1/2}}^{x_{i+1/2}} g(w-B)B_x\,dxdy \\ &\approx -\frac{g}{2\Delta x}\Big(h^L_{i+1/2,j-\alpha} + h^R_{i-1/2,j-\alpha}\Big)\Big(B_{i+1/2,j-\alpha} - B_{i-1/2,j-\alpha}\Big) \\ &\quad -\frac{g}{2\Delta x}\Big(h^L_{i+1/2,j+\alpha} + h^R_{i-1/2,j+\alpha}\Big)\Big(B_{i+1/2,j+\alpha} - B_{i-1/2,j+\alpha}\Big), \end{aligned} \tag{8}$$

A similar construction is used for the third component of S_{ij}.

When water depths approach zero, reconstructing w will not guarantee nonnegative point-values for h. This is very undesirable in the numerics and physically wrong. On the other hand, by reconstructing in h, the scheme is guaranteed to yield nonnegative water depths (under a more restrictive time step). Therefore, as a reasonable compromise, one can choose a threshold K, and use the following strategy: reconstruct point values from the physical variables $[h, u, v]$ if $h < K$ and from $[w, hu, hv]$ otherwise. A version of this scheme is presented in [3]. Notice that this compromise is not well-balanced in the first component, i.e., it produces nonzero flux terms for lake-at-rest.

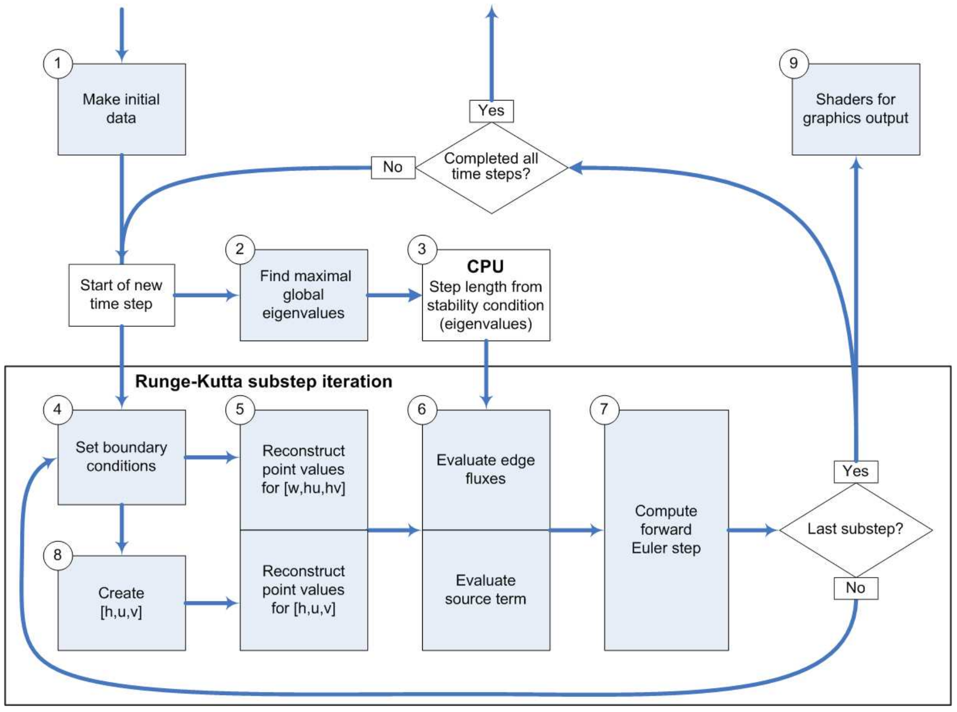
1
Make initial data
Yes
Completed all time steps?
No
9
Shaders for graphics output
2
Find maximal global eigenvalues
3
CPU
Step length from stability condition (eigenvalues)
Start of new time step
Runge-Kutta substep iteration
4
Set boundary conditions
5
Reconstruct point values for [w,hu,hv]
6
Evaluate edge fluxes
7
Compute forward Euler step
Yes
Last substep?
No
8
Create [h,u,v]
Reconstruct point values for [h,u,v]
Evaluate source term

Fig. 1. Flow chart for the GPU implementation of the semi-discrete finite-volume scheme. Gray boxes are executed on the GPU and white boxes on the CPU.

3 Implementation on the GPU

We have implemented the method presented above using OpenGL and Cg to utilise the GPUs capabilities for floating-point processing. The general setup is to render a triangle that covers the entire viewport, and use a fragment shader, that is, a program that the GPU executes for each pixel, as the computational kernel. This kind of usage is generally known as GPGPU (General-Purpose Computation using Graphics Hardware), and an introduction to techniques commonly used in this field can be found in [6]. Algorithms to be used in such a setting must be divided into steps, in such a way that each data cell can be calculated independently within a step in the algorithm. Each step can then be executed by rendering a triangle covering the viewport, which causes the fragment processor to execute the fragment shader, thereby calculating the new cell values. Figure 1 illustrates the different steps and the data flow in the algorithm we implemented.

Initial data <1> are created either by using height maps of the bottom topography and water level or by procedural textures. The CFL stability condition is determined by the global maximum of eigenvalues <2> in the grid cells. To find this maximum we use an 'all-reduce' operation utilising the depth buffer combined with read-back to the CPU for the final calculations <3>. Here we divide the domain into smaller sub-domains and render the eigenvalues of each sub-domain into the depth buffer. The depth buffer used has the same dimension as the sub-domains. The effect of this is that one of the values of the depth buffer must be the largest eigenvalue. This depth buffer is then read back to the CPU, which picks the global maximum. It is important to minimise both the number of read-backs and the amount of data transferred from the GPU to the CPU, because the GPU and the driver are optimised for data being sent from the application (system memory) to the graphics memory and not in the other direction. By utilising the depth buffer we have reduced the data transfer costs to a minimum.

Each step in the Runge–Kutta method begins with a fragment shader that sets the boundary conditions <4>. The next step is to reconstruct point values <5> from the cell averages. For cases with positive water depth, it is only necessary to run the fragment shader for the variables $[w, hu, hv]$. For computations with dry states, we need to reconstruct point values of both $[w, hu, hv]$ and $[h, u, v]$. In this case we precompute $[h, u, v]$ <8>, and then apply the reconstruction shader to both sets of variables.

The most computationally intensive step is the evaluation of edge fluxes and source terms <6>. The time step is provided to this shader after the stability calculations on the CPU are done. For the simplest case where B is constant, the source terms are zero. Then the shader only computes the flux integrals (4). For the more complex case of variable bottom, the source-term integrals (8) are also needed. When the simulation involves dry states, branching is needed at this step to determine if we should use the reconstruction of $[w, hu, hv]$ or $[h, u, v]$ for the calculations. The design of the graphics hardware is not optimal for handling branching efficiently so we should aim at using algorithms with as few branches as possible. This is hard to archive for the scheme used in this paper. Other schemes exist that may perform better in this case, e.g., [7].

The forward Euler step <7> corresponds to equation (7). By repeating the execution of the shaders inside the forward Euler box, one arrives at the second order Runge–Kutta scheme. A number of different shaders are used for output <9> to the screen as illustrated in Figure 2.

4 CPU vs. GPU

In this section we compare CPU and GPU implementations of the schemes in Section 2. We measure runtimes and explore the factors that affect the relative speedup. The GPU is a nVidia Geforce 6800 Ultra and the CPU is a 2.8 GHz Intel Xeon (EM64T).

Case 1. The first test case is presented in [8] and we give the results here for

Table 1
Runtime per time step and speedup factor ν for the CPU versus the GPU implementation of Lax–Friedrichs and the high-resolution scheme without source terms and dry states for the circular dambreak problem run on a grid with $N \times N$ grid cells.

	Lax–Friedrichs			2.order central-upwind		
N	CPU	GPU	ν	CPU	GPU	ν
128	2.22e-3	7.68e-4	2.9	3.06e-2	3.78e-3	8.1
256	9.09e-3	1.24e-3	7.3	1.22e-1	8.43e-2	14.5
512	3.71e-2	3.82e-3	9.7	4.86e-1	3.18e-2	15.3
1024	1.48e-1	1.55e-2	9.5	2.05e-0	1.43e-1	14.3

the convenience of the reader. We consider a simple circular dambreak problem over the domain $[-1.0, 1.0] \times [-1.0, 1.0]$ with absorbing boundary conditions. Th water surface is initially at rest with height $h = 1.0$ inside a circle of radius 0.3 and height $h = 0.1$ outside.

The solution has been computed using both the first-order Lax-Friedrichs scheme (2) and the second-order central-upwind scheme described in Section 2. Runtimes for CPU and GPU implementations are reported in Table 1. The Lax-Friedrichs scheme needs fewer arithmetic operations per grid cell than the central-upwind scheme. The timings reported here show that the speedup is better for schemes with more arithmetic operations per time step.

Case 2. We consider lake-at-rest defined by a variable bottom topography $B(x, y) = \max(0, 1 - x^2 - y^2)$ and a stationary flat water surface $h(x, y, 0) = \max(w_0 - B(x, y), 0)$. For $w_0 = 1.01$, the water depth is strictly positive and the solution is stationary and exactly preserved by the central-upwind scheme. Runtimes per time step for the CPU and the GPU are reported in Table 2 along with corresponding speedup factors ν. As we can see, there is a slight increase in runtimes, but not a significant reduction in speedup, since the added complexity of the scheme has more or less the same effect on the CPU and the GPU.

For $w_0 = 0.9$ we have zero water depth in parts of the domain. In this case, the branching described in Section 2 is needed to ensure nonnegative water depth. As pointed out in Section 3, branching is not as natural on the GPU and may increase the runtime per time step. To avoid data-dependent branching in the reconstruction of point values, both sets of variables are reconstructed in the GPU implementation. The CPU implementation, on the other hand, only reconstructs the needed variables. Though computations are relatively inexpensive, this contributes to the reduced speedup. The measured runtimes and speedup factors are reported in Table 2.

Table 2
Runtime per time step and speedup factor for the CPU versus the GPU implementation of the second-order central-upwind scheme. Initial data is the lake-at-rest with two different surface levels $w_0 = 1.01$ and $w_0 = 0.9$, respectively, run on a grid with $N \times N$ grid cells. For $w_0 = 1.01$ we have used the simpler scheme without the switch for dry-regions.

	without dry states ($w_0 = 1.01$)			with dry states ($w_0 = 0.9$)		
N	CPU	GPU	ν	CPU	GPU	ν
128	3.27e-2	3.83e-3	8.56	3.52e-2	6.18e-3	5.70
256	1.30e-1	9.68e-3	13.43	1.43e-1	1.90e-2	7.51
512	5.18e-1	3.66e-2	14.60	5.99e-1	7.16e-2	8.36
1024	2.14e-0	1.61e-1	13.32	3.27e-0	3.52e-1	9.28

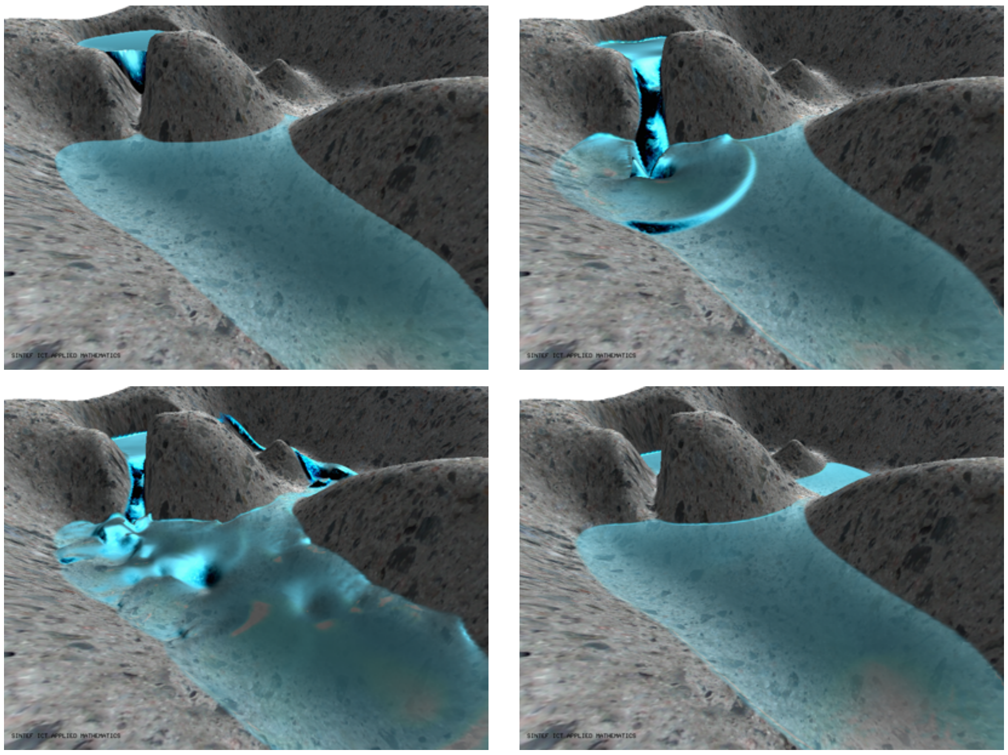

Fig. 2. Snapshots of a dambreak simulation in artificially constructed terrain.

Flood waves caused by a dambreak. In Figure 2, we have included four snapshots of a realistic-looking dambreak simulation on the GPU. Initially, the water in the upper and lower lake is at rest. When the water in the upper lake is released through the narrow canyon, it generates flood waves in the lower lake. The wave motion and the increased water level in the lower lake makes the water flow into the valley below before it again comes to rest. The simulation is interactive in the sense that it has a fly-through mode that allows the user to inspect the solution while it is being computed.

5 Final Comments

In this paper we have demonstrated the applicability of the GPU as a computational resource in PDE-based simulations of gravity-driven surface waves in shallow waters. Modern numerical schemes for such models are inherently parallel in the sense that very little global communication is needed in the computational domain to advance the solution forward in time. Therefore, this application can readily exploit the parallel architecture of modern GPUs. We have seen in practical computations that moving from a serial CPU-based implementation to an implementation on a modern GPU decreases the runtime by an order of magnitude. (From two hours to ten minutes!) To achieve the same speedup using CPUs, one would have to resort to a cluster of ten or more computers.

In our experience, computations on the GPU are as reliable as on the CPU. The single-precision arithmetic of the GPU does not negatively affect the computations, and indeed, for stable methods, it should not. As we have seen, the ratio of simulation speeds on the GPU and the CPU varies. It is interesting to identify what causes these variations. In this paper two effects seem clear: First, the advantage of the GPU is the speed at which arithmetic operations can be performed. Therefore, we can expect that the GPU will perform well for algorithms where the speed of arithmetic computations limits the overall speed. If other factors determine the run time, such as I/O, memory access, etc., the speed of a GPU implementation may be modest. This is what we observe in our time comparisons: The speedup is better for the high-resolution scheme than for the simple Lax-Friedrichs scheme. See [8] for a more extensive comparison.

Second, data dependent branching is expensive. There exist several known methods to conquer this challenge, but the result is highly dependent of the input data and the graphics hardware. Therefore we did not invest too much time to find the best possible solution for the hardware we tested on, but aimed for a reasonable compromise. It seems that much of the performance loss when data-dependent branching is involved happens because the compiler looses some of its freedom to reorder instructions. This is a limitation it is important to be aware of.

It seems obvious that the computing resources on the GPU have many applications outside of computer graphics. First of all, the computing power of desktop computers can be vastly increased with modest expenses. This can have a great impact in end-user software since one may be able to remove bottlenecks in computationally expensive applications. Computations that today are done in batch mode may with the use of the GPU become interactive. With interactivity and fast visualisation, new applications of PDE-based physics are possible, ranging from process steering and control, to computer games and educational simulators.

References

[1] R. J. LeVeque, Finite volume methods for hyperbolic problems, Cambridge Texts in Applied Mathematics, Cambridge University Press, Cambridge, 2002.

[2] A. Kurganov, S. Noelle, G. Petrova, Semidiscrete central-upwind schemes for hyperbolic conservation laws and Hamilton–Jacobi equations, SIAM J. Sci. Comput. 23 (3) (2001) 707–740 (electronic).

[3] A. Kurganov, D. Levy, Central-upwind schemes for the Saint-Venant system, M2AN Math. Model. Numer. Anal. 36 (3) (2002) 397–425.

[4] D. Levy, G. Puppo, G. Russo, Compact central WENO schemes for multidimensional conservation laws, SIAM J. Sci. Comput. 22 (2) (2000) 656–672 (electronic).

[5] C.-W. Shu, Total-variation-diminishing time discretisations, SIAM J. Sci. Stat. Comput. 9 (1988) 1073–1084.

[6] R. Fernando, GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley Professional, 2005.

[7] E. Audusse, F. Bouchut, M.-O. Bristeau, R. Klein, B. Perthame, A fast and stable well-balanced scheme with hydrostatic reconstruction for shallow water flows, SIAM J. Sci. Comp. 25 (2004) 2050–2065.

[8] T. Hagen, M. Henriksen, J. M. Hjelmervik, K.-A. Lie, How to solve systems of conservation laws numerically using the graphics processor as a high-performance computational engine, in: G. Hasle, K.-A. Lie, E. Quak (Eds.), Geometric Modelling, Numerical Simulation, and Optimization: Industrial Mathematics at SINTEF, Springer-Verlag, 2005, to appear.