Modeling Water Waves in the Surf Zone with GPU-SPHysics

Alexis Herault, Anna Vicari, Ciro del Negro
INGV - Sezione di Catania
Piazza Roma, 2
95123 Catania, Italy
herault@ct.ingv.it, vicari@ct.ingv.it, delnegro@ct.ingv.it

Robert A. Dalrymple
Department of Civil Engineering
John Hopkins University
3400 NO. Charles St., Baltimore, MD 21204
rad@jhu.edu

***Abstract*—Water waves impinging on a beach provides a good example of the ability of SPH to model complex free surface flows. The formation of a plunging breaker was shown early with the pioneering work of Monaghan [1]. Dalrymple and Rogers [2] examined breaking waves on a beach, noting that the initially two-dimensional wave field becomes three dimensional when the waves break. Recently Herault et al. [3] have developed a version of the SPHysics open source model (http://wiki.manchester.ac.uk/sphysics) for Nvidia graphics cards using the CUDA extended C++ programming language. Here we show 2D and 3D results using several Nvidia cards and demonstrate the speed-ups achievable with GPU programming versus CPU programming. Our speed up results are for three generations of Nvidia cards, including the new Tesla card, which has 240 streaming processors available. This paper will discuss the Nvidia CUDA code to point out the difference between using the GPU versus the CPU, the various methods used in the GPU model for such tasks as neighbor lists, the memory (shared, global, texture models), and the uses of CUDA kernels. We then present applications of the model to different water waves at beaches. The waves are created within 2D or 3D wave basins, with sloping beaches. The wavemaker is currently a sinusoidally forced flap wavemaker, but that is easily changed. As the waves shoal and break on the slope, the motion clearly becomes three-dimensional. The surf zone vorticity will be discussed and the coherent turbulent eddies will be identified using longshore and cross-shore sections and the q-criterion of Dubief and Delcayre [4] (2000).**

I. Introduction

The introduction of the massively parallel graphic processors (GPU) and particularly of the first compiler (NVidia CUDA) allowing the programming outside of a strictly graphic context of those processors has changed the game in terms of numerical simulation.

Indeed a graphic card of the last generation has a computational power of about 1TFlops and costs only 300 €. It is therefore quite natural to use the GPU for scientific computation and numerical simulation.

Regarding the hardware, one could consider the GPU as a massively parallel computer accessing a global *device memory*, holding hundreds of computing units employing a new architecture called Single Instruction Multiple Threads (SIMT). This architecture is akin to the Single Instruction Multiple Data (SIMD) vector organisation. The NVidia GPU of the last generation (G200) has 240 processors, and a device to device memory bandwidth of about 140 Go/s for a computational power of about 1 TFlop.

At this time, only NVidia distributes a C compiler dedicated to those processors, which has a certain number of restrictions, the most important ones being the impossibility of writing recursive functions and the lack of dynamical allocation in device memory. Also, we can only hope to obtain optimum performance by conforming closely to the SIMT scheme. Indeed, for compute bound operations if the scheme is respected, except for some few exceptions, we are certain to have a code whose performances scales linearly with the number of processors. In addition, one must also acknowledge that the memory bandwidth between the graphic card and the host by the PCI bus is one-to-two order of magnitude lower than the one for the device memory.

Those material constraints are making impossible the direct transformation of a numeric code, even parallelized in a GPU version. The use of GPU as a mathematical coprocessor is also quite limited by the low bandwidth of the PCI bus.

Because of their structure, SPH are particularly well adapted to the GPU. In fact, except in a few rare cases, notably for the treatment of solid boundaries, the calculation of forces, speed and positions is done rather in the same manner for all particles, in a SIMT fashion.

In this context we have developed a GPU version of the SPhysics code. To take the most advantage of the GPU performances, the ensemble of the algorithms is executed on the GPU, the transfers between host and GPU happening only to record the results of the calculation.

II. Previous work

The first implementation of the SPH method totally on GPU was realized by Kolb and Kuntz [5], and Harada et al. [6] (both occurring before the introduction of CUDA). Earlier work by Amada et al. [7] used the GPU only for the force computation, while administrative tasks such as neighbor search was done on the CPU, requiring no less than two host-card memory transfers at each iteration.

Both Kolb and Kuntz and Harada used OpenGL and Cg (C for graphics) to program the GPU, requiring knowledge of computer graphics and also some tricks to work around the restrictions imposed by these languages. In particular, since OpenGL doesn't offer the possibility to write to three-dimensional textures, the position and velocity textures have to be flattened to a two-dimensional structure.

III. SPH FORMULATION

We use almost the same formulation as in SPhysics that we shall briefly recall. In all formulas below, h is the smoothing length, W the smoothing kernel, $\mathbf{r}_{ab} = \mathbf{r}_a - \mathbf{r}_b$, $\mathbf{v}_{ab} = \mathbf{v}_a - \mathbf{v}_b$, $r = |\mathbf{r}_{ab}|$, $W_{ab} = W(r,h)$ and $\sum_b$ denotes the sum over the neighbors of the particle a.

A. Smoothing kernels

We implemented four smoothing kernels: cubic spline, quintic spline, quadratic and Wendland.

B. Continuity equation

$$\frac{d\rho_a}{dt} = \sum_b m_b \mathbf{v}_{ab} \nabla_a W_{ab} \tag{1}$$

C. Momentum equation with artificial viscosity

$$\frac{d\mathbf{v}_a}{dt} = -\sum_b m_b \left(\frac{P_a}{\rho_a^2} + \frac{P_b}{\rho_b^2} + \prod_{ab} \right) \nabla_a W_{ab} + \mathbf{g} \tag{2}$$

where $\prod_{ab}$ is the artificial viscosity term [8] defined by :

$$\prod_{ab} = \begin{cases} \dfrac{-\alpha c_{ab}\mu_{ab}}{\rho_{ab}}, \text{ if } \mathbf{v}_{ab}.\mathbf{r}_{ab} < 0 \\ 0, \text{otherwise} \end{cases}$$

with $\mu_{ab} = \dfrac{h\mathbf{v}_{ab}.\mathbf{r}_{ab}}{r^2 + 0.01h^2}$, $\rho_{ab} = \dfrac{1}{2}(\rho_a + \rho_b)$ and $c_{ab} = \dfrac{1}{2}(c_a + c_b)$

D. Momentum equation with laminar viscosity

For the laminar viscosity term we use the formulation given by Morris et al. [9]:

$$\frac{d\mathbf{v}_a}{dt} = -\sum_b m_b \left(\frac{P_a}{\rho_a^2} + \frac{P_b}{\rho_b^2} \right) \nabla_a W_{ab} + \mathbf{g} + \sum_b m_b \frac{4\nu_0 \mathbf{r}_{ab}.\nabla_a W_{ab}}{(\rho_a + \rho_b) r^2} \mathbf{v}_{ab} \tag{3}$$

where ν_0 is the kinetic viscosity.

E. Equation of state

Following Monaghan [1] and Batchelor [10] the relationship between pressure and density is assumed to follow the expression:

$$P = B\left[\left(\frac{\rho}{\rho_0} \right)^\gamma - 1 \right] \tag{4}$$

where $\gamma = 7$ and B is chosen such that the minimum sound speed is ten times greater than the maximum expected flow speed [1].

F. Kernel corrections

We implemented a zeroth order and a first order Moving Least Square (MLS) kernel correction used to periodically reinitialize the density field.

Among that we also implemented the velocity gradient correction proposed by Chen and Braun [11].

The MLS and Chen and Braun kernel corrections require a matrix inversion. In order to avoid any attempt to invert a singular matrix we use the following criterion: the matrix M is inverted only if $|M|^k > \varepsilon$, k being the dimension of M and ε a safety factor typically set to 0.01.

1) Zeroth order: Shepard Filter

The 'Shepard Filter' [12] is a straightforward correction where the density field is periodically reinitialized using the following equation:

$$\tilde{\rho}_a = \sum_b m_b \tilde{W}_{ab} \tag{5}$$

where $\tilde{W}_{ab} = \dfrac{W_{ab}}{\sum_b \dfrac{m_b}{\rho_b} W_{ab}}$

2) First order: MLS

The MLS approach developed by Dilts [13] is a first order correction designed to exactly reproduce linear variations of the density field. As for the Shepard filter, the density field is periodically reinitialized using the following equation:

$$\tilde{\rho}_a = \sum_b m_b W_{ab}^{MLS} \tag{6}$$

where W_{ab}^{MLS} is the corrected kernel evaluated as follow:

$$W_{ab}^{MLS} = \mathbf{b}(\mathbf{r}_a).\mathbf{r}_{ab} W_{ab}$$

In 2D the correction vector **b** is given by:

$$\mathbf{b}(\mathbf{r}_a) = \mathbf{A}^{-1} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

where:

$$\mathbf{A} = \sum_b \frac{m_b}{\rho_b} W_{ab} \tilde{\mathbf{A}}$$

$$\tilde{\mathbf{A}} = \begin{pmatrix} 1 & x_a - x_b & y_a - y_b \\ x_a - x_b & (x_a - x_b)^2 & (x_a - x_b)(y_a - y_b) \\ y_a - y_b & (x_a - x_b)(y_a - y_b) & (y_a - y_b)^2 \end{pmatrix}$$

3) Chen and Braun gradient correction

The Chen and Braun gradient correction is a generalization of the Bonnet and Lok [14] correction for the gradient of the velocity. It consist in replacing the term $\nabla_a W_{ab}$ by $\mathbf{A}_a^{-1}\nabla_a W_{ab}$ where $\mathbf{A}$ is the matrix defined by:

$$A_a = \sum_b \frac{m_b}{\rho_b}\nabla_a W_{ab} \otimes \mathbf{r}_{ba} \qquad (7)$$

G. Boundary conditions

We implemented repulsive boundary conditions proposed by Monaghan and Kos [15] using special particles generating a short-range repulsive force according to:

$$\mathbf{f}(\mathbf{r}) = \begin{cases} D\left(\left(\frac{r_0}{r}\right)^{p_1} - \left(\frac{r_0}{r}\right)^{p_2}\right)\frac{\mathbf{r}}{r}, \text{ if } r \le r_0 \\ 0, \text{ if } r > r_0 \end{cases} \qquad (8)$$

where r_0 is the range of action, $p_1 = 12$ and $p_2 = 6$ and D is a problem depending constant.

We also implemented open periodic boundary conditions.

H. Time integration

We use the Predictor-Corrector scheme. As an example, for the velocity, the predictor step predict the midpoint:

$$\mathbf{v}_a^{n+\frac{1}{2}} = \mathbf{v}_a^n + \frac{\Delta t}{2}\frac{d\mathbf{v}_a^n}{dt} \qquad (9)$$

while the corrector step is:

$$\tilde{\mathbf{v}}_a^{n+\frac{1}{2}} = \mathbf{v}_a^n + \frac{\Delta t}{2}\frac{d\mathbf{v}_a^{n+\frac{1}{2}}}{dt} \qquad (10)$$

the value of $\mathbf{v}$ at the end of the step is then:

$$\mathbf{v}_a^{n+1} = 2\tilde{\mathbf{v}}_a^{n+\frac{1}{2}} - \mathbf{v}_a^n = \mathbf{v}_a^n + \Delta t\frac{d\mathbf{v}_a^{n+\frac{1}{2}}}{dt} \qquad (11)$$

The evolution of density and position are similar.

When using variable time-stepping, the time step is calculated according to (Monaghan and Kos [15]):

$$\Delta t = 0.3\min\left(\Delta t_f, \Delta t_{cv}\right)$$

$$\Delta t_f = \min_a \sqrt{\frac{h}{|\mathbf{f}_a|}};\ \Delta t_{cv} = \min_a \frac{h}{c + \max_b\left|\frac{h\mathbf{v}_{ab}.\mathbf{r}_{ab}}{r_{ab}^2}\right|} \qquad (12)$$

We also implemented the XSPH correction given by Monaghan [16].

IV. CUDA PROGAMMING MODEL

A. Hardware considerations

The CUDA enabled NVidia processors (modèles G80, G92 et G200) share the same hardware architecture (figure 1): each processor is made up by a set of multiprocessors each one containing eight processors.

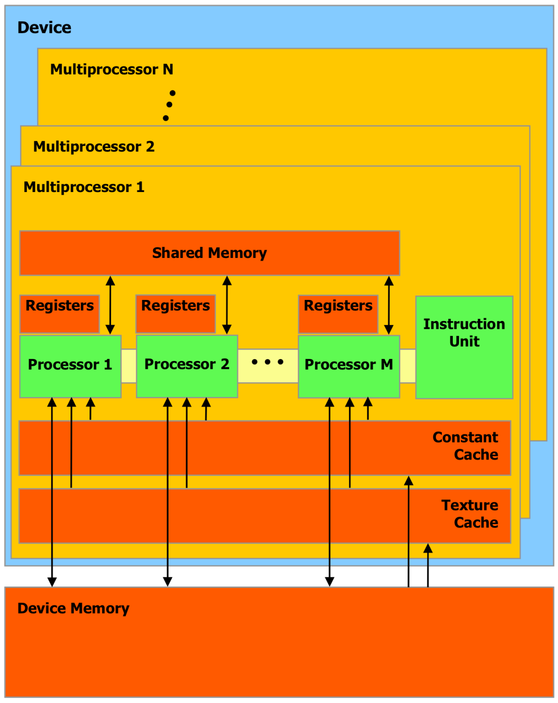

Figure 1. Hardware organization

Each multiprocessor has four kinds of on-chip memories:

- a set of 32 bits registers per processor,
- a *shared memory* shared by all processors of the same multiprocessor,
- a read only *constant cache* designed to speed-up reads from constant memory,
- a read only *texture cache* designed to speed up texture read. Textures are accessed via a texture unit implementing various addressing and filtering modes (like texture interpolation).

The device itself accesses a global memory space, located on the board, the *device memory*.

B. Software considerations

CUDA is an extension of C programming language that allows the programmer to write functions executed directly on the graphic processor. Unlike other classic C functions, those functions called kernels (not to be confused with SPH kernels), are executed N times, in parallel by N different CUDA threads. At this time, CUDA does not support recursive functions.

Those threads are organized in a grid containing an ensemble of blocks. As a rule, the size of the grid is directly linked to the size of the data to be treated. Those blocks are processed by a multiprocessor. The number of threads per block is limited by the number of registers and by the quantity of shared memory used by the kernel. Each multiprocessor has a special unit, the SIMT unit, in charge of spreading the threads between the various processors. This architecture is related to the SIMD architecture of the vectorial computers while being less restrictive: two threads executed over two different processors can execute different instructions. However, this comes at a cost, so it is recommended to restrict to the bare minimum the conditional instructions leading to those situations.

The access time to the memory depends as well of the type of memory involved:

- the first access to the *device memory* has a latency of 400 - 600 clock cycles,
- access to *shared memory* and *registers* is generally a zero extra clock cycle operation,
- *constant cache* and *texture cache* are cached memory, so the access cost in the event of a cache miss is the same that for the *device memory*, otherwise it just costs one read from the cache.

The *device memory* bandwidth can be improved using some special access patterns, called coalesced accesses (for details see the CUDA Programming Guide). The thread scheduler can hide the memory access latency if the kernel contains sufficient independent arithmetic instructions that can be issued while waiting for other memory access to complete. Such kind of kernel is called *compute bound kernel* and guaranties a linear performance scaling in respect of the number of processors. In opposition a kernel involving mostly memory accesses is called *memory bound kernel* and will not scale with the number of processors.

The *texture cache* is optimized for spatial locality, so threads of the same block reading texture addresses close together will achieve a better performance.

The *shared memory* is shared by the threads of a block, and cannot be used for inter-block communication.

There is no way to avoid read-after-write, write-after-read or write-after-write hazards in *device memory*, so there are no global variables safely accessible by all the threads of a grid. CUDA provides Atomic Functions to perform safely read-modify-write operations, but they are restricted to integers and have a significant cost.

Summarizing, in order to obtain the best performance, we can list three main issues:

- choose the most suitable kind of memory with the best access pattern,
- restrict to absolute minimum conditional instructions,
- try to have a maximum of *compute bound kernels*.

V. SPH IMPLEMENTATION WITH CUDA

A. General overview

The code for our CUDA implementation of SPH exploits C++ structures to clearly isolate the components. The CPU part is composed of a *Problem* virtual class from which the actual problems (dam break, piston, paddle, …) are derived. A *ParticleSystem* class acts as an abstraction layer for the particle system, taking care of CPU and GPU data structure initialization and CUDA kernels calling. It exposes methods to initialize data, run each simulation time step, and retrieve simulation results for visualization (OpenGL image displayed on screen and refreshed periodically as the model is running) or on-disk storage (targa image format and/or VTU data file written at a given interval).

Following the classical SPH approach, the algorithm for the simulation time step can be decomposed into the following four steps: neighbor search and list update, force computation, velocity and position update. In addition, Shepard or MLS filtering can be periodically done.

B. The Problem class

The *Problem* class is the user-exposed part of the code. In order to run a new type of problem, the user must write a child class of *Problem* in which it defines all the simulation parameters (domain size, smoothing length, kernel type, density at rest, coefficients for the equation of state,…) and options (periodic boundary, variable time step, kernel correction to apply,…). In addition it must fill an array with the initial particle position, velocity and density. This is done by implementing the virtual functions defined in the *Problem* class.

Furthermore the user is provided with a set of geometrical utility classes (for rectangles, polygons, cubes, …) with a function allowing the user to fill the corresponding geometry with particles.

C. Inside the ParticlSystem class

1) CUDA kernels development guidelines

The methods of *ParticleSystem* class handle the whole SPH simulation by calling different CUDA kernels. All CUDA kernels are written using the following guidelines:

- the arrays stored in the *device memory* are allocated once, before the beginning of the computation,
- the particles data are stored in arrays with elements of one, two or four floats in order to ensure in most of the cases coalesced memory access. When it's not possible the array are bound to a one dimensional texture and accessed via a texture fetch,
- except for radix sort used by neighbor search and parallel reduction used for the adaptive time-stepping, **one thread processes one particle**,
- the constants used in SPH formulation are stored in the *constant memory*,
- we avoided any unnecessary runtime conditional expressions arising from the simulation options by using the templatization capabilities of the CUDA C compiler (selection of the kernel type, viscosity type, periodic boundary) or by writing multiples slightly different versions of CUDA kernels (variable time step, use of gradient correction, XSPH).

2) Data structures

All the data are stored in arrays in the *device memory*. The more relevant are listed below:

- the *pos* arrays contain positions and masses,
- the *vel* arrays contain velocities and densities,

- the *forces* array contains the accelerations and density derivatives
- the *neiblist* array contains the indexes of all neighbors of all particles.

Actually we have *two* pos and *vel* arrays for handling the extra storage needed by the integration scheme.

3) *Neighbor list construction*

For the neighbor search, we use the algorithm described in the Particles example of the CUDA toolkit (Green, 2008). The computational domain is divided into a grid of linearly indexed cells; each particle is assigned the cell index it lies in as a hash value, and the particles are reordered by their hash value. The neighbor list for each particle is then built by looking for neighbors only in nearby cells. Four CUDA kernels are involved in the neighbor list construction: hash calculation, hash table sorting with a radix sort, particle reordering and the neighbor list builder. We actually fix the maximum number of neighbors per particle at compile time for simplicity and efficiency reasons, as it allows the compiler to unroll any loop involved, resulting in faster code and allowing us to avoid any complex index tricks.

Rather than computing the neighbor list on the fly at each iteration, we store the neighbor list in the *neiblist* array that gets rebuilt periodically (typically every 10 iterations). This choice requires more memory but results in faster execution, since the neighbor search is a memory-bound operation and therefore doesn't scale up with the computational power of the GPU.

4) *Force computation*

The *Forces* CUDA kernel is devoted to the density derivative and acceleration computation. It walks through the neighbor list of the particle, accumulating the density derivative and acceleration according to equations (1), (2) or (3) and (8).

When using the Chen and Braun gradient correction, the *Forces* kernel firstly computes the correction matrix (eq. 7), inverts it, and proceeds to the density derivative and acceleration computation with the corrected gradient. The kernel has thus, at this stage, gone twice through the particles' neighbor list.

The resulting acceleration and density derivative are stored in the *forces* array.

For adaptive time-stepping we perform a first parallel reduction directly in the *Forces* kernel taking advantage of shared memory. This allows us to find the value of the time step (eq. 12) for the particles processed by the same thread block. The maximum time step value is then found by calling a parallel reduction kernel on the remaining data.

5) *Time integration and positon update*

The two integration steps in the Predictor-Corrector method (eq. 9 and 10) are structurally very similar to a simple Euler integration step. So we implemented a basic *Euler* kernel doing such operation.

Then, a full integration step consist in:

- a call to the *Forces* kernel to compute density derivative and acceleration at time step n,
- a call to the *Euler* kernel with an half-fed time step to compute the positions and velocities at time step $n + 1/2$,
- a call to the *Forces* kernel to compute the corrected density derivative and acceleration at time step $n + 1/2$,
- a last call to *Euler* kernel with a full time step to complete the integration.

The position and velocities of boundary particles are updated according to their type (fixed boundary particles or moving boundary particles belonging to a gate, piston or paddle).

In case of adaptive time-stepping each call to the Forces kernel is followed by a call to a parallel reduction kernel. The time step used at the next iteration is then set at the value of the lower of the two time steps given by the two parallel reductions.

6) *Shepard and MLS kernel corrections*

The Shepard and MLS kernel corrections are implemented in two CUDA kernels periodically called during the computation at a user-defined rate (typically every 20 – 40 time step).

As for the *Forces* kernel those kernels perform a walk through (one for the Shepard correction and two for the MLS correction) the neighbor list computing the corrected density according to equations (5) or (6).

VI. BENCHMARKS

Table 1 shows the speedup of our implementation against its execution on the CPU only. The table shows that the neighbor list kernels do not scale linearly with the number of processors in the GPU; this indicates that they are memory-bound, which is expected given that their main feature is to reorder memory structures based on the particle hash and that this only requires a small number of mathematical operations. For the force calculation procedure the speedup is considerably closer to the optimal speedup, indicating that the corresponding kernels are compute-bound, as expected.

The Euler step scales roughly linearly with the GPU compute capability, indicating a strong computational bound, but its advantage over the reference CPU is closer to the memory-bound procedures. Reasons for this can be found in the high memory access/operation ratio (5 memory accesses for 4 operations) and in the highly vectorization structure of the Euler computation that is exploited by the vectorized instructions available on the CPU.

Fig. 2 shows the repartition of calculation time between the different parts of the algorithm.

TABLE I. SPEED-UPS BY GPU TYPE AND CODE PROCEDURE FOR GPU-SPHYSICS CODE FOR 677,360 PARTICLES

Procedure	**GPU model (#number of processors)**		
	8600m (32)	***8800GT (110)***	***GTX 280(240)***
Neighbor list	4.4	11.8	15.1
Force calculation	28	120	207
Euler step	3.2	13.7	23.8

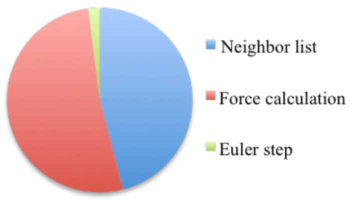

Figure 2. Repartition of calculation time.

Note that the speedups in Table 1 are calculated against the performance of the GPU code when run on a single CPU, an Intel Core2 Duo at 2.5GHz. It must be noted that this code is not optimal for CPU execution: for example, it does not consider the symmetry in particle interactions, evaluating them twice (once per particle instead of once per particle pair): although the independent, repeated evaluation is almost a necessity on the GPU, it is an unnecessary burden on the CPU; optimizing it away would improve the CPU speed by a factor of two, halving the speed-ups in Table 1.

VII. APPLICATIONS

A. Breaking waves

PaddleTest3D and a *PaddleTest2D* class were developed to examine waves in a wave basin or tank, respectively. The model problem involves generating boundaries for the tank/basin, including the sidewalls and the bottom, using a *Rect* (for rectangle) object, that creates a wall filled with boundary particles. In 3D, the bottom consists of two rectangles—a flat section, upon which the wavemaker is placed, and a sloping beach section. The wavemaker is another *Rect* object that stretches across the width of the tank and that moves with a prescribed sinusoidal motion (in time) about a line on the bottom. The paddle motion increases in amplitude linearly with elevation from the bottom to create a flap wavemaker. Finally, fluid particles are placed between the wavemaker and the beach in layers spaced Δp apart in all three dimensions. Δp is the initial particle spacing, which is related to the smoothing length of the weighting kernels by $h=1.3\Delta p$.

It is straightforward to vary the tank geometry, including the beach slope, and the wave properties, such as the wave period and the amplitude of the paddle motion, which are related [16]. This leads to different kinds of breaking waves on the slope: spilling, plunging, or surging. Dalrymple and Rogers [2] show that the SPH methodology is fully capable at examining plunging breaking waves with splash-up.

The GPU-SPHysics code, once initiated, produces three types of output. The first is real-time display of the waver tank or basin, with the current wave field, using OpenGL. The OpenGL image can be rotated about three axes, zoomed, and the user has the option of displaying particle positions, or pressure, or velocity mapped onto the particles. This provides the user with confidence that the model is running correctly. User-chosen periodic time-interval screen-capturing of the OpenGL window results in saved images (.tga format). Finally, also at a user-chosen periodic interval, data files are written that capture particle positions, velocities, pressures and densities at that instant. Currently these data files are in VTU format.

Fig. 3 shows a simulation using the PaddleTest3D model, but for waves in a narrow tank. This simulation was carried out on an Nvidia Tesla card, using 4 million particles—near the top limit that we can use, based on our current GPU-SPHysics code. The straight lines in the figure denote the various rectangles and polygons used to generate the wave tank boundaries. The wavemaker is at the left and it creates sinusoidal 2D waves—the model begins with the entire tank filled with still water and then the wavemaker begins moving. The water is shown in blue, while paler colors (and white) denote water moving at higher velocities. Interestingly, as the waves shoal on the sloping bottom and eventually break, the streakiness of the water in the middle of the tank and shoreward indicates that some three dimensionality develops in the flow.

Another positive result of the test is that the intersection between the dry beach and the water changes with time, moving on average shoreward. This is a manifestation of wave-setup, which is caused by the conservation of momentum of the incoming waves being balanced during breaking by the hydrostatic response of the surf zone.

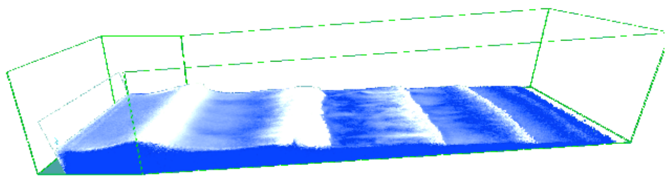

Figure 3. Wave tank test of the 3D GPU-SPHysics code, showing wave breaking and three-dimensional features developing in the breaking region. Wavemaker to the left generates 2D motion and the beach is located on the right.

Using a wider wave tank shows that GPU-SPHysics has some likelihood of modelling nearshore hydrodynamics, such as longshore currents and rip currents. In Fig. 4 the tank that is as wide as it is long is shown. In this wider domain, the wave crests are no longer straight but show some variation along the beach (on the right).

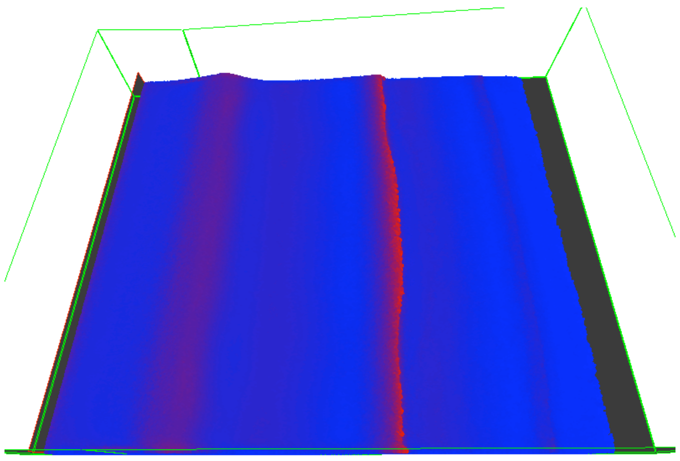

Figure 4. The wave basin is set up similarly to the wave tank in Figure 2. Here red denotes the high velocity regions.

B. Dam break on a real topography

We developed a special version of the *ParticleSystem* object dedicated to the simulation of flows on a real topography. The mains differences with the previously presented version are:

- for the neighbor list construction we divided the domain into a two-dimensional grid,
- the Digital Elevation Model (DEM) describing the topography is stored in a two-dimensional texture,
- we did not use any boundary particles to represent topography, but instead generated the repulsive force according to equation (8), i.e. in consideration of (i) the distance between the particles and the terrain and (ii) the associated normal direction.

We take benefit of the built-in linear texture interpolation in order to have smooth elevations and normals for close particles regardless of the resolution of the DEM (Fig. 5).

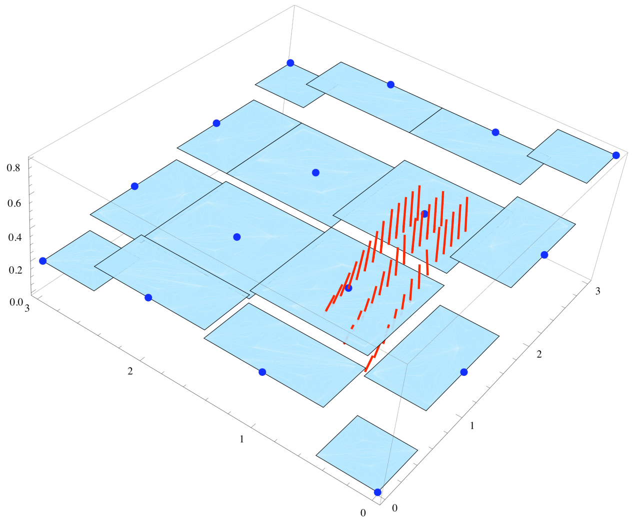

Figure 5. In blue the DEM data, in red terrain normals generated at a resolution ten times less than that used for the DEM.

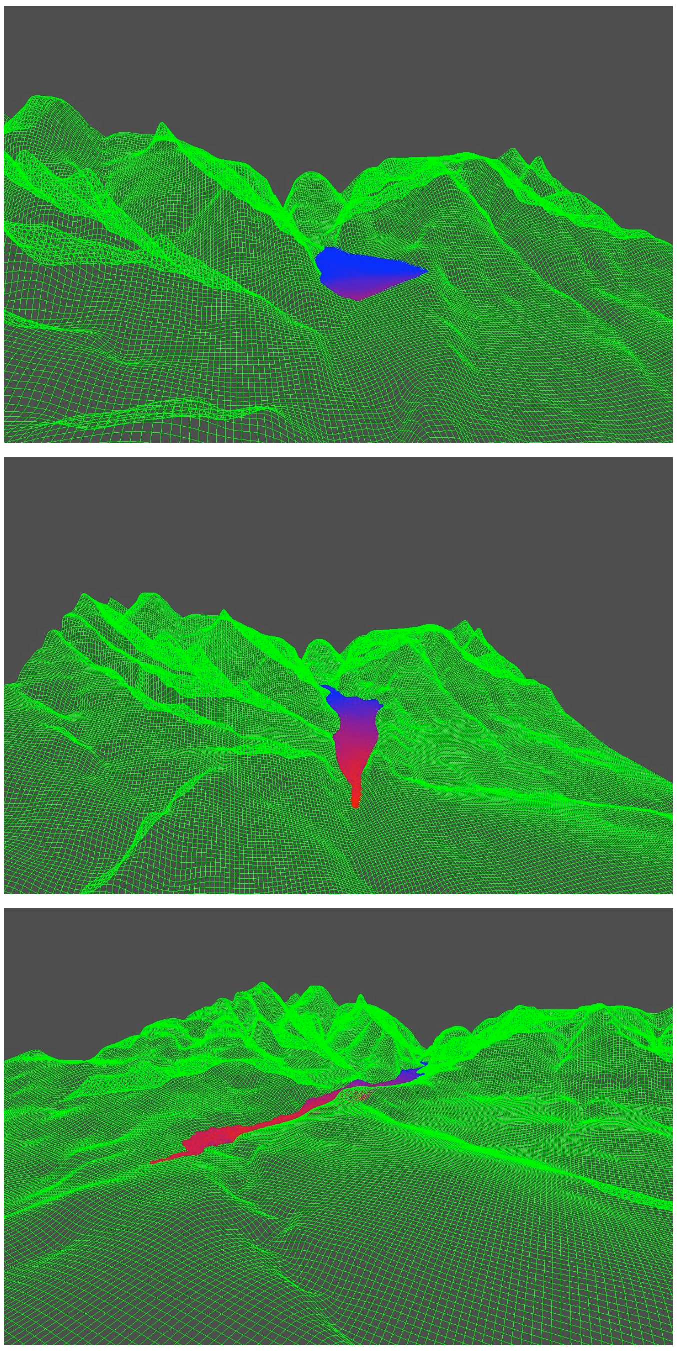

Figure 6. Evolution of the dam break at 0.5s, 5.03s and 9.16s.

Fig. 6 shows the evolution of the flow caused by a dam break on a real topography. For that simulation we used a 1m resolution DEM, an initial particle spacing of 0.2m and around 170000 particles.

Define abbreviations and acronyms the first time they are used in the text, even after they have been defined in the abstract. Abbreviations such as IEEE, SI, MKS, CGS, sc, dc,

[1] Monaghan, J.J. Simulating free surface flows with SPH. J. Comp. Phys., 110, pp. 399- 406, 1994

[2] Dalrymple, R. A. and Rogers, B. D. Numerical Modeling of Water Waves with SPH Method. Castal Engineering. 53(2-3), pp 141–147, 2006.

[3] Hérault, A., Dalrymple, R.A. and Bilotta, G. SPH on GPU with CUDA. J. Hydraulic Research, 2009, in press.

[4] TODO.

[5] Kolb, A. and Cuntz, N. Dynamic particle coupling for GPU-based fluid simulation, Proc. 18th Symposium on Simulation Technique, pp 722-727, 2005.

[6] Harada, T., Koshizuka S., and Kawaguchi Y. Smoothed Particle Hydrodynamics on GPUs. Proc. of Computer Graphics International, pp 63–70, 2007.

[7] TODO.

[8] Monaghan, J. J. Smoothed Particle Hydrodynamics. Annual Rev. Astron. Appl., 30, pp543–574, 1992.

[9] Morris, J., Fox, P., and Zhu, Y. Modeling low Reynolds number incompressible flows using SPH. Journal of Computational Physics, 136, pp 214–226, 1997.

[10] Batchelor, G. K. Introduction to fluid dynamics. Cambridge University Press, 1974.

[11] Chen, J. K., Braun, J. E., and Carney, T. C. A corrective Smoothed Particle Method for boundary value problems in heat con- duction. Computer Methods in Applied Mechanics and Engineering, 46, pp 231–252, 1999.

[12] Shepard, D. A two dimensional function for irregularly spaced data. In ACM National Conference, 1968.

[13] Dilts, G. A. Moving-Least–Squares-Particle Hydrodynamics – I. Consistency and stability, Int. J. Numer. Meth. Engng, 44, pp 1115-1155, 1999.

[14] Bonet, J. and Lok, T.-S. L. (1999). Variational and momentum preservation aspects of Smoothed Particle Hydrodynamics formulations. Computat. Methods Appl. Mech. Engineering., 180, pp 97–115, 1999.

[15] Monaghan, J. J. and Kos, A. Solitary Waves on a Cretan Beach. J. Waterway, Port, Coastal and Ocean Engineering., 125, pp 145–154, 1999.

[16] Dean, R.G. and Dalrymple, R.A. Water Wave Mechanics for Engineers and Scientists, World Scientific Press, 199