Real-time Lattice Boltzmann Shallow Waters Method for Breaking Wave Simulations

Jesus Ojeda[1] and Antonio Susín[2]

[1] Dept. LSI, Universitat Politècnica de Catalunya
[2] Dept. MA1, Universitat Politècnica de Catalunya

Abstract. We present a new approach for the simulation of surface-based fluids based in a hybrid formulation of Lattice Boltzmann Method for Shallow Waters and particle systems. The modified LBM can handle arbitrary underlying terrain conditions and arbitrary fluid depth. It also introduces a novel method for tracking dry-wet regions and moving boundaries. Dynamic rigid bodies are also included in our simulations using a two-way coupling. Certain features of the simulation that the LBM can not handle because of its heightfield nature, as breaking waves, are detected and automatically turned into splash particles. Here we use a ballistic particle system, but our hybrid method can handle more complex systems as SPH. Both the LBM and particle systems are implemented in CUDA, although dynamic rigid bodies are simulated in CPU. We show the effectiveness of our method with various examples which achieve real-time on consumer-level hardware.

Keywords: Fluid Simulation, Natural Phenomena, Physically-based Animation

1 Introduction

In the last years, professionals from real-time rendering and interactive fields have become more aware of physically-based effects as new graphics hardware can be used for such purposes. Among the most common features in actual computer games we find particle systems, rigid bodies and fluid simulations, being the last one the most complex and difficult to achieve in real-time. Moreover, the possibility of coupling all these simulations opens a wide range for building rich scenes with more interactivity.

Regarding fluid simulations, the restrictions of the equations and the extension of the simulations make them difficult to solve. Eulerian fluid simulations compute the fluid properties at fixed points in space, distributed over a grid. On the other hand, Lagrangian approximations evaluate the fluid properties at points that are advected with the fluid itself. Whatever the chosen method, the visualization of the fluid is usually based on its surface, which, for great volumes of water, can be simplified to this boundary, so the 3D simulation could potentially be reduced to a 2D simulation of an evolving height field.

Solving the 2D wave equation is a common technique to simulate fluid surfaces as height fields, but it can not resolve effects based on horizontal velocity fields as whirlpools. To account for this, a shallow water framework is preferred. Derived from the more common Navier-Stokes equations, it is implemented based on a discretization on time and space over a grid. A less commonplace alternative derivation of these equations is based on the Lattice Boltzmann Method, which simplifies the implementation, restricting the maximal wave speed.

A 2D heightfield representation of a fluid can not account for many interesting phenomena that could happen in a full 3D simulation, like breaking waves. To improve this situation, we propose an implementation of an hybrid system that couples a Shallow Water Lattice Boltzmann with particle systems in CUDA for real-time fluid simulation with the following key features:

- Use of arbitrary underlying terrain.
- A method to maintain stability and to track dry-wet regions in the simulation.
- Two-way simplified coupling with rigid body simulations using a proxy system.
- Breaking wave detection conditions.
- Full particle generation, simulation and reintegration with the heightfield system.

Although we have used a ballistic particle system for the present work, it is easily interchangeable with other, more sophisticated methods, like SPH.

1.1 Related Work

A simple way to simulate water surfaces is based in procedural methods, as those based in the Fast Fourier Transform like [1] or [2]. These methods are well suited for the generation of high resolution and large scale animations, and have been used extensively in commercial products as movies or videogames; however, they are not easily coupled with solid objects and are unable to simulate eddies.

In computer graphics, among the first to use a shallow waters framework, [3] implemented a pipe model, which was later extended by [4], by using particles for the splashes generated from falling objects. More recently, [5] ported this model to GPUs for the simulation of hydraulic erosion. As an alternative, [6] presented a novel approach using wave trains on 2D particles to solve the wave equation. These methods, however, can not simulate the effects of horizontal flow.

On the other hand, the Shallow Water Equations (SWE) can simulate these phenomena. In addition to the heightfield description of the fluid surface, it also simulates a 2D horizontal velocity field. [7] were the first to introduce them to the graphics field. Among other works, [8] used them to simulate breaking waves and later were ported to CUDA by [9], coupling it with a particle system.

A Smooth Particle Hydrodynamics (SPH) system can also be used to solve these equations. [10] coupled an SPHSW with the wave equation to obtain higher detail fluid surfaces. [11] ported the SPHSW simulation to CUDA and has already been extended by [12].

Yet another formulation can be stated with the Lattice Boltzmann Method (LBM). The LBMSW derivation can be found in [13] and has been used in various scenarios. Among others, [14] coupled it with a full 3D LBM simulation and [15] used it to simulate the currents in the strait of Gibraltar. More recently, [16] simplified the force terms of the formulation.

As the LBM is quite similar to a cellular automaton, it can be implemented in a parallel setting without much effort with regard to other methods. There are already GPU implementations as [17], where it was adapted using textures; but more recently, with the advent of general programmability of GPUs we find CUDA implementations like [18] and [19]. [20] proposed an alternative kernel implementation to reduce memory usage and [21] targeted multiple different parallel architectures using higher-level libraries for solving the LBMSW model.

2 Methodology

The main steps our hybrid particle-LBM coupling executes for one time step can be summarized as:

1. LBMSW fluid simulation.
2. Rigid body simulation.
3. Two-way coupling of rigid bodies and LBMSW.
4. Particle generation and simulation.
5. Render

The first step is to advance the LBMSW simulation explained in Section 2.1. This takes into account external forces and the dry-wet region tracking from Section 2.2. Using any external package, as the Bullet Physics library in our case, rigid bodies are simulated. These are then coupled with the fluid as presented in Section 2.3. This two-way coupling affects the movement of the dynamic objects but also modifies the behaviour of the fluid. Next, particles are generated and simulated for all the fluid regions the LBMSW can not handle, as breaking waves. These particles subtract some volume from the LBMSW in their generation and restore it back when they fall to the surface again. Details about this process will be discussed in Section 2.4. Finally, the render of the scene should be done.

Further details about the CUDA implementation are given in Section 3.

2.1 Lattice Boltzmann Shallow Waters

In contrast to other methods where a set of partial differential equations is discretized and solved directly, the Lattice Boltzmann Method already provides a discrete model suitable for parallel computations using only arithmetic operations. The fluid is simulated by particle distributions over a regular grid (distribution functions *df*s). The particle's movement is restricted to certain directions $\mathbf{e}_i$ defined by the Boltzmann discretization used.

We use the D2Q9 model, pictured in Figure 1, and assuming an adimensional parametrization as in [14], the velocity vectors $\mathbf{e}_{0..8}$ take the values: $\mathbf{e}_0 = (0,0)^T$, $\mathbf{e}_{1,2} = (\pm 1, 0)^T$, $\mathbf{e}_{3,4} = (0, \pm 1)^T$ and $\mathbf{e}_{5..8} = (\pm 1, \pm 1)^T$.

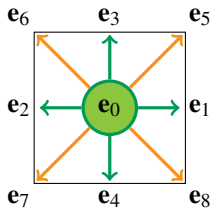

Fig. 1. D2Q9 model: nine velocity square lattice.

The Lattice Boltzmann Equation, then, defines the behaviour of the fluid by the chosen collision operator. We employ here the common BGK operator [22]

$$f_i(\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) = f_i(\mathbf{x}, t) - \omega(f_i - f_i^{eq}) + \mathcal{F}_i, \tag{1}$$

where f_i^{eq} is the *df* for the $\mathbf{e}_i$ direction, ω is the relaxation parameter, in close relation with the viscosity of the fluid, and f_i^{eq} is the local equilibrium distribution function, which defines the actual equations that are being solved. The original SWE can be recovered by applying Chapman-Enskog expansion if f^{eq} is defined like in, e.g., [13]

$$f_i^{eq}(h, \mathbf{u}) = \begin{cases} h\left(1 - \frac{5}{6}gh - \frac{2}{3}\mathbf{u}^2\right), & i = 0, \\ \lambda_i h\left(\frac{gh}{6} + \frac{\mathbf{e}_i \cdot \mathbf{u}}{3} + \frac{(\mathbf{e}_i \cdot \mathbf{u})^2}{2} - \frac{\mathbf{u}^2}{6}\right), & i \neq 0, \end{cases} \tag{2}$$

where $\lambda_i = 1$ for $i = 1..4$ and $\lambda_i = 1/4$ for $i = 5..8$. g is the gravity and h and $\mathbf{u}$ are the macroscopic fluid properties; height level from the underlying terrain and velocity, respectively, calculated as

$$h(\mathbf{x}, t) = \sum_i f_i, \tag{3}$$

$$\mathbf{u}(\mathbf{x}, t) = \frac{1}{h} \sum_i \mathbf{e}_i f_i. \tag{4}$$

From Equation 1, $\mathcal{F}_i$ are the external forces applied to the LBMSW. In contrast to how these forces are applied in, e.g., [15] or [21]; [16] stated them with simpler arithmetic operations as

$$\mathcal{F}_i = \mathcal{X}_i + \mathcal{Z}_i. \tag{5}$$

From a constant underlying terrain $z_b(\mathbf{x})$ defined as a heightfield, $\mathcal{X}_i$ is the force caused by its slope as

$$\mathcal{X}_i = \begin{cases} \frac{g[h(\mathbf{x}+\mathbf{e}_i\Delta t, t) + h(\mathbf{x}, t)]}{2}[z_b(\mathbf{x} + \mathbf{e}_i \Delta t) - z_b(\mathbf{x})], & i = 1..4, \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

$\mathcal{Z}_i$ internalises other forces F, as friction or the Coriolis effect, defined as

$$\mathcal{Z}_i = \begin{cases} 0, & i = 0, \\ \frac{F_\alpha}{6e_{i_\alpha}}, & i \neq 0, \end{cases} \tag{7}$$

where α is a Cartesian index and Einstein summation convention is used. The same can be applied to e_{i_α}. We only consider the friction with the underlying terrain, so F_α is defined as

$$F_\alpha = C_t u_\alpha \sqrt{u_\beta u_\beta}, \tag{8}$$

where β is the other Cartesian index and C_t is the terrain friction coefficient, defined as a constant. u_α and u_β are the components of the fluid velocity in the α and β directions, respectively.

As boundary conditions, we use a no-slip boundary which is implemented as a bounce-back rule: the *df*s that should be streamed from boundary cells are just inverted as

$$f_i(\mathbf{x}, t + \Delta t) = f_{\bar{i}}(\mathbf{x}, t), \tag{9}$$

where $f_{\bar{i}}$ is the *df* in the opposite direction of f_i, i.e., $\mathbf{e}_i = -\mathbf{e}_{\bar{i}}$.

Additionally, we use for the rest of the paper the value η defined as

$$\eta(\mathbf{x}, t) = h(\mathbf{x}, t) + z_b(\mathbf{x}). \tag{10}$$

2.2 Dry-Wet Region Tracking

In order to be able to track dry regions, i.e., cells that do not contain fluid, we modify the original algorithm. We define a threshold ϵ as the minimal height a cell must satisfy to be considered a Fluid cell. After an iteration of the LBM has been executed, we must look for cells whose level has dropped below the threshold. For all the found cells, we must tag them as Empty. In order to not lose fluid mass, we also distribute the remainder of the fluid between the Fluid neighbours favoring the direction of the underlying terrain gradient as follows:

$$f_i(\mathbf{x} + \mathbf{e}_i \Delta t) = f_i(\mathbf{x} + \mathbf{e}_i \Delta t) + h(\mathbf{x}) \cdot (\zeta_i / \zeta_{total}), \tag{11}$$

where ζ_{total} is the sum of all weights ζ_i, which are computed as

$$\zeta_i = \begin{cases} -(\nabla z_b \cdot \mathbf{e}_i) & \text{if } -(\nabla z_b \cdot \mathbf{e}_i) > 0 \text{ and cell at} \\ & (\mathbf{x} + \mathbf{e}_i) \text{ is a Fluid one,} \\ 0 & \text{otherwise.} \end{cases} \tag{12}$$

Seamlessly, we search also for Fluid cells whose fluid level is above the threshold and whose neighbours are Empty cells. We tag these Empty cells as Fluid, in order to allow the advance of the fluid from the Fluid tagged cell.

This addition enables for the tracking of dry-wet regions, but Equation 4 still poses a limitation: as the fluid level goes down, the velocities can grow very large and lead to inevitable instabilities. In contrast to [21], where they used a modified minmod flux limiter to solve this, we use the Froude number, which relates the characteristic velocity of the fluid to a gravitational wave velocity

$$Fr = \frac{\sqrt{\mathbf{u} \cdot \mathbf{u}}}{\sqrt{gh}}, \tag{13}$$

and is defined as $Fr < 1$ for subcritical flows, just the case of the LBMSW simulations [23]. We define an upper limit parameter φ for that ratio. When, due to low fluid height, the ratio does not hold, we compute a new suitable velocity $\mathbf{u}$ for the fluid and replace the dfs of the cell with new ones computed from Equation 2. Also, we can further use this condition to dampen high velocities through the full body of fluid and ensure a stable simulation.

Although not physically correct, this method ensures stability in a similar fashion to the Smagorinsky method [24]: it changes the local viscosity of the fluid and dampens high velocities, as can be seen in Figure 2.

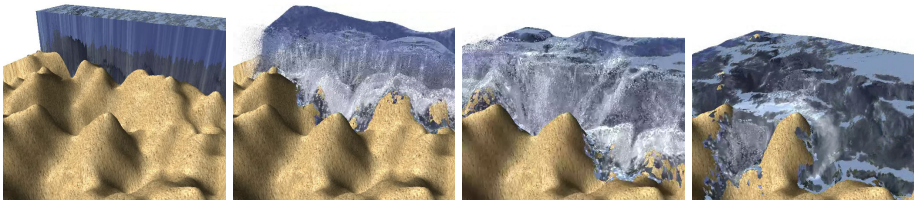

Fig. 2. Image stills from the breaking dam over noisy ground example. Using values of $\epsilon = 0.1$ and $\varphi = 0.95$.

2.3 Two-Way Coupling of Dynamic Rigid Bodies

For the introduction of rigid bodies to the LBMSW simulation we propose the use of a proxy model to decouple the complexity of the interaction of the fluid with the object mesh, in contrast to [9], where they use a tessellated mesh to the level of using triangles of areas similar to Δx^2, from the fluid simulation.

Our proxy model is composed of a set spheres and can be understood as a rough discretization of the object mesh. The properties defined for the spheres are the radius r, the position $\mathbf{p} = (p_x, p_y, p_z)^T$ and a normal $\mathbf{n} = (n_x, n_y, n_z)^T$. During the simulation, the spheres will also hold a velocity $\mathbf{v} = (v_x, v_y, v_z)^T$.

For our examples, we have used manually discretized models, as the boat in Figure 3. Depending on the discretization of the model in spheres, the simulation becomes more accurate but also more expensive. For a regular discretization with spheres of radius $r < \Delta x/2$ our results are visually similar to [9].

Fluid to Rigid Body For the implementation of the fluid to solid coupling we follow the path of [6] and [9]. There are three main forces a fluid induces to a solid body: buoyancy, drag and lift. We will compute them at the sphere positions of the proxy object. Assuming the simulation plane is xz, then $\hat{y} = (0, 1, 0)^T$.

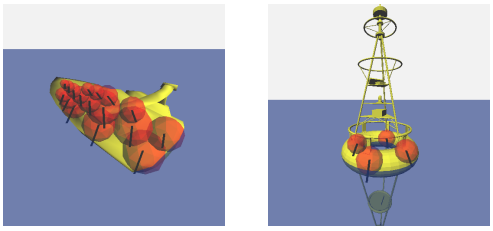

Fig. 3. Sphere discretization example for the boat and buoy models. The spheres are positioned and sized within the models, their normal vectors represented by the black short lines.

The buoyancy force points upward and is proportional to the weight of the displaced fluid, we can define it for sphere i as

$$\mathbf{f}_i^{buoy} = \begin{cases} 0 & \text{if } S_i^p - S_i^r > \eta_p, \\ g\rho V_{sub}\hat{y} & \text{otherwise,} \end{cases} \tag{14}$$

where, η_p is the water level at the sphere position, S_i^r is the sphere radius, S_i^p is the y coordinate of the location of the sphere, ρ is the density of the fluid and V_{sub} is the volume of the submerged part of the sphere calculated as

$$V_{sub} = \int_{-S_i^r}^{top} \pi(S_i^{r\,2} - x^2)dx, \tag{15}$$

with $top = (\eta_p - (S_i^p - S_i^r))$.

Drag force is a resistive force and is dependent on the actual velocity of the obstacle with regard to the fluid. Lift is a force perpendicular to the oncoming flow direction, but is also dependent on the actual fluid velocity. For sphere i, they are defined as

$$\mathbf{f}_i^{drag} = -\frac{1}{2}C_D A_{2D}\|\mathbf{u}_{rel}\|\mathbf{u}_{rel}, \tag{16}$$

$$\mathbf{f}_i^{lift} = -\frac{1}{2}C_L A_{2D}\|\mathbf{u}_{rel}\|\left(\mathbf{u}_{rel} \times \frac{S_i^n \times \mathbf{u}_{rel}}{\|S_i^n \times \mathbf{u}_{rel}\|}\right), \tag{17}$$

where C_D and C_L are the drag and lift coefficients, $\mathbf{u}_{rel}$ is the relative velocity of the sphere with respect to the fluid, S_i^n is the normal defined for the sphere and A_{2D} is the area of the circle that cuts the sphere at water level η_p.

The forces are finally added to the ith sphere. The rigid body simulator will take care of the evolution of the proxy model and will provide the corresponding transform which will be applied in the render phase.

Rigid Body to Fluid In this case, the rigid body will modify the behaviour of the fluid. As before, the computations are done per sphere. To change the fluid correctly, we get the velocity of the obstacle for the ith sphere as $\mathbf{v}$ and

the difference between the submerged height of the sphere and the fluid level as *depth*. We compute the following values

$$decay = \exp(-depth), \tag{18}$$

$$h_o = decay \cdot C_{dis} \cdot depth, \tag{19}$$

$$\mathbf{u}_o = decay \cdot C_{adp} \cdot \mathbf{v}, \tag{20}$$

and input these h_o and $\mathbf{u}_o$ into the LBM equilibrium distribution, Equation 2, updating the previous *df*s as

$$\begin{aligned} f_0 &= f_0 - h_o, \\ f_i &= f_i + f_i^{eq}(h_o, \mathbf{u}_o) + \frac{f_0^{eq}(h_o, \mathbf{u}_o)}{\mathsf{w}_o}, \text{ where } \mathsf{w}_o = \begin{cases} 5 & i = 1..4, \\ 20 & i = 5..8. \end{cases} \end{aligned} \tag{21}$$

The values of w_o are calculated from the contribution each $\mathbf{e}_i$ gives on the D2Q9 model [25]. With this computation we push the fluid the obstacle is displacing to the neighbour cells, taking into account in this process the obstacle velocity. Additionally, to avoid high differences between lattice neighbours, we distribute the h_o and $\mathbf{u}_o$ among the nearest cells using linear interpolation.

decay takes into account the depth the sphere is at and limits accordingly the effect it has over the fluid surface. C_{dis} and C_{adp} are parameters in the range $[0, 1]$ that allow to dampen the effect of the coupling as desired. We have used the values $C_{dis} = 0.8$ and $C_{adp} = 0.5$ for the examples of Figure 4.

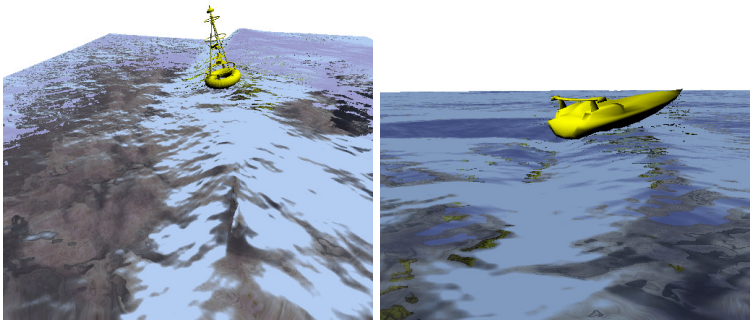

Fig. 4. A buoy is being dragged by the fluid (left). The boat introduces some new fluid waves at its tail as a result of the coupling (right).

2.4 Coupling of Particle Systems

The LBMSW model described so far is only capable of representing fluids as heightfields and certain phenomena is limited, e.g., breaking waves can not be

resolved. In order to deal with this restriction, we have coupled it with a ballistic particle system and adapted the detection of breaking waves and generation of the respective particles from [9]. They also proposed other detection conditions for when to generate particles for the interaction with obstacles and terrain discontinuities like waterfalls, which could also be adapted to our system. In this case, however, we restrict ourselves to the breaking wave example. In contrast, we will present an implementation that allows alternative particle systems like SPH with minor changes in Section 3.

Breaking Wave Detection Waves that would break in a full 3D simulation just produce singular waves due to numerical instability in a Shallow Waters simulation. The detection of this situation for a given cell (i, j) is done via three parametrized conditions:

$$\|\nabla \eta_{i,j}\| > \Phi g, \tag{22}$$

$$\eta_{i,j} - \eta_{i,j}^{prev} > \Psi, \tag{23}$$

$$\nabla^2 \eta_{i,j} < \Upsilon, \tag{24}$$

where $\eta_{i,j}^{prev}$ is the fluid height in the previous time step and Φ, Ψ and Υ are parameters, which should be tailored per scene, and more specifically by its scale. Equation 22 ensures the wave is steep enough to break. Equation 23 requires that the cell is part of the front of the wave and it is raising fast, introducing a comparison with the previous value of fluid height. Finally, Equation 24 makes sure particles are only generated near the top of the wave.

The computation of $\nabla \eta_{i,j}$ is done using the maximum among the one-sided derivatives and $\nabla^2 \eta_{i,j}$ is computed using central differences

$$\nabla \eta_{i,j} = \begin{bmatrix} \frac{max(|\eta_{i+1,j} - \eta_{i,j}|, |\eta_{i,j} - \eta_{i-1,j}|)}{\Delta x} \\ \frac{max(|\eta_{i,j+1} - \eta_{i,j}|, |\eta_{i,j} - \eta_{i,j-1}|)}{\Delta x} \end{bmatrix}, \tag{25}$$

$$\nabla^2 \eta_{i,j} = \frac{\eta_{i+1,j} + \eta_{i-1,j} + \eta_{i,j+1} + \eta_{i,j-1} - 4\eta_{i,j}}{\Delta x^2}. \tag{26}$$

If all three conditions are met, the next step will generate and initialize particles for the given cell. The total volume V_{total} the added particles will subtract from the LBMSW is proportional to $\|\nabla \eta_{i,j}\| - \Phi g$ and can be controlled introducing a new parameter θ, as

$$V_{total} = \theta(\|\nabla \eta_{i,j}\| - \Phi g). \tag{27}$$

Particle Generation For each cell detected in the previous step, a number of particles will be generated for the volume of Equation 27.

The particles are positioned within a cell-centered rectangle of width equal to the LBMSW cell width and height V_{total} as shown in Figure 5. This rectangle is oriented with the opposite direction of the gradient computed in Equation 25.

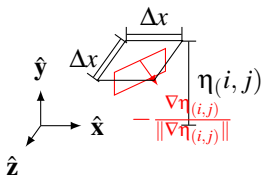

Fig. 5. For breaking wave detected particles, they are placed within the red rectangle in their generation step.

The particle velocities in the xz plane are defined by the wave speed as in [8] and the y component can be defined as a fraction of the height differences from Equation 23 as

$$\mathbf{v}_{xz} = \frac{-\nabla\eta_{i,j}\sqrt{gh}}{\|\nabla\eta_{i,j}\|}, \tag{28}$$

$$v_y = \lambda_y(\eta_{i,j} - \eta_{i,j}^{prev}), \tag{29}$$

where λ_y controls the fraction. We have used here a value of $\lambda_y = 0.1$.

Additionally, we lightly perturb the velocity of each particle and jitter their initial positions between $[\frac{-\Delta x}{2}, \frac{\Delta x}{2}]$ in the gradient direction, which helps to add variation and result in less uniform, more chaotic, particle movement.

The total volume the particles supply must be subtracted from the LBMSW, as well as the momentum they get. We do this by computing the equilibrium distribution function from Equation 2; using as input values V_{total} and the xz velocity components from the particle velocities, prior to the perturbations we apply. These newly computed equilibrium dfs will be subtracted from the cell's original df set as

$$f_i = f_i - f_i^{eq}\left(\frac{V_{total}}{\Delta x^2}, \mathbf{v}_{xz}\right). \tag{30}$$

As said previously, particles are not restricted to be generated only from the detected breaking waves of the previous step. We can generate and initialize particles with other requirements in mind, like a faucet pouring fluid into a basin as demonstrated by Figure 6.

Particle Reintegration Finally, the particles must be reintegrated to the LBMSW when they hit the surface of the fluid, i.e., $p_y \leq \eta_{i,j}$. The volume the particles carry, as well as their momentum, must be absorbed by the cell they fall on.

As the LBMSW has no explicit method to input vertical velocities, we introduce an interpolation for the absorption of the volume of the particle among the cell's dfs. This interpolation is based on the terminal speed the particle could achieve, defined as

$$v_T = \sqrt{\frac{8rg}{3C_D}}, \tag{31}$$

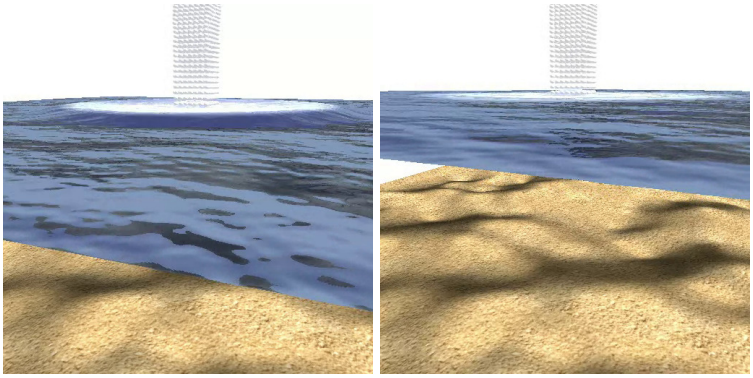

Fig. 6. Particles generated in middle air (like a heavy rain or some pipe open tab), integrated afterwards to the bulk of the fluid. After a few seconds, the level of the LBMSW is effectively raised.

where C_D is the drag coefficient. We normalize the particle's vertical speed with v_T and clamp the result to the range $[0, 1]$, as $\chi = clamp(v_y/v_T, 0, 1)$.

Taking into account the previous consideration, we can update the dfs of the cell using the following computations

$$f_0^{eq\chi} = f_0^{eq}\left(\frac{V_p}{\Delta x^2}, \mathbf{v}_{xz}\right), \tag{32}$$

$$f_0 = f_0 + (1-\chi)\cdot f_0^{eq\chi}, \tag{33}$$

$$f_i = f_i + f_i^{eq}\left(\frac{V_p}{\Delta x^2} + \chi f_0^{eq\chi}, \mathbf{v}_x z\right). \tag{34}$$

Similarly to the obstacle to fluid coupling from Section 2.3, using the interpolation with the terminal speed, the added volume is pushed from the cell's center to its neighbours with more energy, the faster the particle drops. Figure 2 shows how particles generated from a breaking dam wave are reintegrated even in dry sections and Figure 6 shows how the water level of a basin is effectively raised from the mid-air dropped particles.

3 Implementation Details

In this section we give some implementation details of the Particle-LBMSW coupling. As there is no simple way to maintain a dynamic data structure for the particles on the GPU, we have resorted to a fixed number of particles from the beginning of the simulation. In addition to the usual particle properties as position and velocity, we add two more: a TTL (time-to-live) value and an active (ACTIVE/INACTIVE) flag. We will explain their use in the particle-related functions.

In Algorithm 1 we show high-level pseudo-code for the full simulation. All CUDA functions are kernels, except the sort, remove and prefix_sum parallel

Algorithm 1 Full per frame Particle-LBMSW high-level algorithm.

`dt` = (Δt) frame time step (16ms)
`dt'` = ($\Delta t'$) LBM dimensional time step

```
foreach(frame) {
  //CPU
  ObstacleSimulation();
  ObstacleFluidCoupling();
  //CUDA
  ReintegrateParts_S1();
  sort_tuples();
  remove_nonValidTuples();
  prefix_sum_tuples();
  ReintegrateParts_S2();

  for(i=0; i<dt; i+=dt') {
    LBM_stream_collision();
    LBM_applyForce();
    upd_CellTags_pre();
    upd_CellTags_Fluid();
    upd_CellTags_Empty();
  }
  computeLBM_GradLaplacian();

  sort_particlesByTTL();
  stepParticles();
  detectBreakingWaveCells();
  prefix_sum_NeededPartsPerCell();
  initParticles();

  //Render
}
```

operations, which are provided by the Thrust library. The kernels that are only targeted to a limited group of cells or particles provide an early exit condition for the elements that are not to be changed. Below, we will explain the different kernels, starting from the LBM simulation to the Particle coupling at last.

The LBM core, executed in the LBM_stream_collision kernel is the same as previous LBM implementations in CUDA like, e.g., [21], [18] or [19]; using the BGK collision operator instead of the MRT one. This is done in an inner loop, as the $\Delta t'$ for the LBM can be smaller than the Δt of the frame and depends on the parametrization used. In contrast to [20], where they proposed an A-A memory access pattern to reduce memory requirements, we have used an A-B memory access pattern; there are two arrays for the dfs in memory and they are interchanged after each iteration. The reason for this choice is the additional

operations we are doing, they would have required to double the kernels, as the A-A memory access pattern needs two kernels just for the core simulation.

The LBM_applyForce kernel adds the force terms from Equation 5 for the underlying slope and friction.

The three kernels upd_CellTags_* are the ones responsible for the actual Dry-Wet region tracking. Fluid cells that have a height above the threshold ϵ must convert their Empty neighborhood to Fluid in order to allow the fluid to advance. Seamlessly, Fluid cells with a height below the threshold must be changed back to Empty. In CUDA, we could fall into race conditions in this change of type for the cells, so we have to serialize the reflaging operations, thus the three kernels. upd_CellTags_pre checks the height of the cells against the threshold and preflags them with an additional type if necessary: tobeFluid or tobeEmpty. Next, we change the type of the cells conservatively, first the Fluid-to-be ones and then the Empty ones, ensuring that no cells are changed prematurely if they should be needed in the next iteration.

Then, the computation of the gradient and laplacian of the fluid height is done for further use in the breaking wave detection kernel.

The particles are then sorted by their TTL in ascending order and the simulation is advanced in stepParticles, which depends on the chosen particle system. For a ballistic particle system, the interaction between particles is ignored. The particles' TTL are also updated, subtracting the current Δt and their status is set to ACTIVE. If a particle has died (TTL ≤ 0) before being reintegrated, we let them be ACTIVE but out of view. This ensures we don't lose mass because of dead particles in the particle generation step.

From the previously computed gradient and laplacian and Equations 22 to 24 we detect the cells that have a breaking wave. Each cell will output the needed particle count that it needs. Then, with a prefix sum operation we can obtain an accumulated sum of the needed particles and use this result as the index at the particle array from which each cell will take their needed particle count. As particles have been sorted by TTL, we ensure the particles first taken in this step are those who had a lower TTL. With a bad parametrization this can lead to artifacts, as disappearing particles from frame to frame as they are needed. initParticles will, then, initialize the particles needed for each cell as explained in Section 2.4, marking them as ACTIVE2 which ensures they are alive at least for a frame. Their TTL is also set up as the maximum allowed time to live for a particle, which is a user-defined parameter. For particles that were previously marked as ACTIVE, no fluid will be subtracted from the LBMSW, ensuring no mass loss; thus, only INACTIVE particles will take fluid from the LBMSW.

Finally, the particles are reintroduced. Only ACTIVE particles will be looked for. For these particles, the reintegration should be as easy as the explanation from Section 2.4 but we can fall in race conditions if multiple particles fall in the same cell. As our hardware, a GTX280, does not support atomic float operations, we had to solve it from another perspective: ReintegrateParts_S1 relates which particles have fallen in which cells and how many there are for each cell; from the cell point of view, ReintegrateParts_S2 will gather the fallen particles and update

Table 1. Timings per frame for various examples in milliseconds; the number in the name indicates the number of particles used, where $k = 2^{10}$. LBM includes the LBM simulation and the dry-wet region tracking. Solids accounts for the coupling of rigid bodies. PGen, PSim and PReint are the timings for the generation, simulation and reintegration of the particles, respectively. The timings for the sort operation from the Thrust library are noted in Psort and PReint_sort; they do not depend directly on the other steps but have a significant impact on the results. Psort is for the sorting of particles by their TTL. PReint_sort is the sort of the (cell_id, particle_id) tuples.

		Total	LBM	Solids	Psort	PGen	PSim	PReint	PReint_sort
CPU	boat	10.78	10.69	0.09	0.00	0.00	0.00	0.00	0.00
	buoy	10.91	10.85	0.06	0.00	0.00	0.00	0.00	0.00
	drop 32k	372.96	9.97	0.00	214.56	1.16	1.46	31.50	114.31
	drop 128k	1635.36	9.75	0.00	1001.35	2.09	2.79	114.92	504.46
	wave 32k	410.88	9.62	0.00	224.04	3.68	4.85	32.58	136.11
	wave 128k	1694.87	9.62	0.00	1007.30	3.37	10.71	117.51	546.36
GPU	boat	0.82	0.35	0.47	0.00	0.00	0.00	0.00	0.00
	buoy	0.87	0.35	0.52	0.00	0.00	0.00	0.00	0.00
	drop $32k$	14.30	0.35	0.00	7.03	0.45	0.10	1.24	5.13
	drop $128k$	23.50	0.34	0.00	10.71	0.41	0.14	1.75	10.15
	wave $32k$	15.28	0.36	0.00	7.26	0.99	0.12	1.32	5.23
	wave $128k$	25.71	0.36	0.00	11.21	1.42	0.32	2.01	10.39
	wavegr $64k$	18.79	0.39	0.00	9.48	1.18	0.18	1.64	5.92

the local *df*s. In order to do so, for each particle, S1 will write a tuple associating the cell id with the particle id, as well as the particle count for each cell. Particles not to be reintegrated are associated to a fake cell, in this case we use the cell 0 that we ensure is a Boundary cell for all examples. Sorting the tuples by the cell id, removing those with the fake cell id and doing a prefix sum on the particle-in-cell count will lead us to the cells having the index where their particle count starts in the tuple array. S2 will, for each cell, take their counted fallen particles and reintegrate them, marking them as INACTIVE with TTL = 0.

While the rigid body simulation is done in CPU, we update the values as in Section 2.3 using the CUDA memory arrays mapped to CPU memory space.

4 Results and Discussion

We have tested our implementation both on CPU with OpenMP and GPU with CUDA, timings shown in Table 1 for various examples. Our test system was an Intel Core2Duo E8400 with 4GB of RAM memory and a Nvidia GTX280. The size of the grid used throughout the examples is set to 128x128 and we fix the time step for each frame to $\Delta t = 16ms$. For the boat and buoy scenes, the particle coupling was deactivated to allow us a better timing and similarly, for the wave examples, the object coupling was deactivated. The wavegr example is basically the same as the others, a breaking wave generated from a breaking dam, but in this case the rest of the domain is totally empty as shown in Figure 2.

Although a direct comparison with [9] would not be totally fair because of the difference in the hardware and their lack of implementation details, at least for the particle simulation in CUDA, we think that our LBM-based hybrid system is a great alternative up to the challenge for real-time fluid simulations.

To ensure all the particles were reintroduced correctly without loss mass and because the GTX280 had no support for float atomic operations, we had to separate the reintroduction step in two kernels plus some other Thrust powered operations; the particles are not directly reintroduced but gathered by the cells. These additional operations add more time to the processing of the particles than what it should be needed with more modern hardware.

Nevertheless, we have shown that a coupling of LBMSW with a particle system is feasible for higher-detail fluid simulations. The particle system, however, is not limited to the ballistic version used in here. While the coupling should be the same, i.e., generation and reintegration, TTL of particles and active flag; the simulation and behaviour of the particles can be defined alternatively. A CUDA implementation of SPH like [26] could be easily adapted to our hybrid system.

One limitation our system has, however, is the sudden disappearance of particles due to high demanding simulations, i.e., more particles are needed per frame than what is available. It will be interesting to look at LOD techniques that relax this situation: if more particles than available are needed, the ones actually being active could be represented with simpler primitives, grouping nearby particles, etc. Alternatively, it also would be worth to try and prioritize the preservation of visible particles, i.e., those that fall in the actual view frustum.

Although we have not explained how the visualization is done, the render of the fluid is based in triangle meshes in OpenGL. This can provoke some visual artifacts in the dry-wet region boundaries, which should also be considered.

Other future work also includes the use of a rigid body simulation totally in the GPU, improve the detection conditions for breaking waves and add other particle generation conditions to further broaden the use of this method.

Acknowledgements

With the support of the Research Project TIN2010-20590-C02-01 of the Spanish Government.

References

1. Tessendorf, J.: Simulating ocean water. SIGGRAPH course notes (1999)
2. Hinsinger, D., Neyret, F., Cani, M.P.: Interactive animation of ocean waves. In: SCA. (2002) 161–166
3. Kass, M., Miller, G.: Rapid, stable fluid dynamics for computer graphics. In: SIGGRAPH. (1990) 49–57
4. O'Brien, J.F., Hodgins, J.K.: Dynamic simulation of splashing fluids. In: Proc. of the Computer Animation. CA '95 (1995) 198–
5. Št'ava, O., Beneš, B., Brisbin, M., Křivánek, J.: Interactive terrain modeling using hydraulic erosion. In: SCA. (2008) 201–210

6. Yuksel, C., House, D.H., Keyser, J.: Wave particles. ACM Trans. Graph. **26**(3) (July 2007)
7. Layton, A.T., van de Panne, M.: A numerically efficient and stable algorithm for animating water waves. The Visual Computer **18** (2002) 41–53
8. Thürey, N., Müller-Fischer, M., Schirm, S., Gross, M.: Real-time breaking waves for shallow water simulations. In: 15th Pacific Conference on Computer Graphics and Applications. (2007) 39 –46
9. Chentanez, N., Müller, M.: Real-time simulation of large bodies of water with small scale details. In: SCA. (2010) 197–206
10. Cords, H.: Mode-splitting for highly detailed, interactive liquid simulation. In: GRAPHITE. (2007) 265–272
11. Lee, H., Han, S.: Solving the shallow water equations using 2d sph particles for interactive applications. The Visual Computer **26** (2010) 865–872
12. Solenthaler, B., Bucher, P., Chentanez, N., Müller, M., Gross, M.: SPH Based Shallow Water Simulation. In: VRIPHYS. (2011) 39–46
13. Salmon, R.: The lattice boltzmann method as a basis for ocean circulation modeling. Journal of Marine Research **57**(3) (1999) 503–535
14. Thürey, N.: Physically based Animation of Free Surface Flows with the Lattice Boltzmann Method. PhD thesis, Dept. of Computer Science 10, University of Erlangen-Nuremberg (2007)
15. Thömmes, G., Seaïd, M., Banda, M.K.: Lattice boltzmann methods for shallow water flow applications. International Journal for Numerical Methods in Fluids **55**(7) (2007) 673–692
16. Zhou, J.G.: Enhancement of the labswe for shallow water flows. Journal of Computational Physics **230**(2) (2011) 394 – 401
17. Wei, X., Li, W., Mueller, K., Kaufman, A.: The lattice-boltzmann method for simulating gaseous phenomena. Visualization and Computer Graphics, IEEE Transactions on **10**(2) (march-april 2004) 164 –176
18. Tölke, J.: Implementation of a lattice boltzmann kernel using the compute unified device architecture developed by nvidia. Computing and Visualization in Science **13** (2010) 29–39
19. Obrecht, C., Kuznik, F., Tourancheau, B., Roux, J.J.: A new approach to the lattice boltzmann method for graphics processing units. Computers & Mathematics with Applications **61**(12) (2011) 3628 – 3638
20. Bailey, P., Myre, J., Walsh, S., Lilja, D., Saar, M.: Accelerating lattice boltzmann fluid flow simulations using graphics processors. In: Int. Conf. on Parallel Processing. (2009) 550 –557
21. Geveler, M., Ribbrock, D., Göddeke, D., Turek, S.: Lattice-boltzmann simulation of the shallow-water equations with fluid-structure interaction on multi- and manycore processors. In Keller, R., Kramer, D., Weiss, J.P., eds.: Facing the multicore-challenge. Springer-Verlag (2010) 92–104
22. Qian, Y.H., D'Humières, D., Lallemand, P.: Lattice BGK models for Navier-Stokes equation. EPL (Europhysics Letters) **17**(6) (1992) 479
23. Zhou, J.G.: Lattice Boltzmann Methods for Shallow Water Flows. Springer (2004)
24. Hou, S., Sterling, J., Chen, S., Doolen, G.D.: A lattice boltzmann subgrid model for high reynolds number flows. Fields Institute Communications **6** (1996) 151–166
25. He, X., Luo, L.S.: Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation. Phys. Rev. E **56** (Dec 1997) 6811–6817
26. Goswami, P., Schlegel, P., Solenthaler, B., Pajarola, R.: Interactive SPH simulation and rendering on the GPU. In: SCA. (2010) 55–64