IEEE/ EG Symposium on Volume and Point-Based Graphics (2008)
H.- C. Hege, D. Laidlaw, R. Pajarola, O. Staadt (Editors)

Adaptive Sampling and Rendering of Fluids on the GPU

Yanci Zhang[†] Barbara Solenthaler Renato Pajarola

Visualization and MultiMedia Lab, University of Zürich

Abstract

In this paper, we propose a novel GPU-friendly algorithm for the Smoothed Particle Hydrodynamics (SPH) simulation for weakly compressible fluids. The major goal of our algorithm is to implement a GPU-based SPH simulation that can simulate and render a large number of particles at interactive speed. Additionally, our algorithm exhibits the following three features. Firstly, our algorithm supports adaptive sampling of the fluids. Particles can be split into several sub-particles in geometrically complex regions to provide a more accurate simulation. At the same time, nearby particles deep inside the fluids are merged to a single particle to reduce the number of particles. Secondly, the fluids are visualized by directly computing the intersection between ray and an isosurface defined by the surface particles. A dynamic particle grouping algorithm and equation solver are employed to quickly find the ray-isosurface intersection. Thirdly, based on the observation that the SPH simulation is a naturally parallel algorithm, the whole SPH simulation, including the adaptive sampling of the fluids as well as surface particle rendering, is executed on the GPU to fully utilize the computational power and parallelism of modern graphics hardware. Our experimental data shows that we can simulate about 50K adaptively sampled particles, or up to 120K particles in the fixed sampling case at a rate of approximately 20 time steps per second.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computational Geometry and Object Modeling]: – Physically Based Modeling; I.3.3 [Picture/Image Generation]: – Bitmap and Framebuffer Operations; I.3.7 [Three-Dimensional Graphics and Realism]: – Color, shading, shadowing, and texture I.3.7 [Three-Dimensional Graphics and Realism]: – Animation

1. Introduction

Physically based fluid simulation is an increasingly important technique in engineering applications, movie and game industry. Creating and rendering realistic fluids at interactive frame rates remains a challenging and interesting problem. The SPH method [Mon92, MCG03] is a very successful approach used in computer graphics to simulate fluids. Compared to other physics-based fluid simulation methods such as mesh-based methods [CMT04, KFCO06] or height-field approximations [IGLF06], SPH may be the best choice to simulate splashes, spray, as well as very large or unbounded simulation domains.

With the increasing demand for more complex animations and therefore simulations with more particles, improving the simulation and rendering efficiency becomes a very important issue. There are several techniques that can be exploited to address this problem. The first is to adaptively sample the fluid. With this technique, the computational resources can be focused on the simulation regions with interesting fluid flow behavior without compromising the visual quality of the animation. The second is to utilize the computational power and parallelism of the GPU. Note that the SPH method itself is basically a data-parallel algorithm. There are only a few data dependencies in the standard SPH simulation which can be removed by appropriately replacing some of the formulas. This makes it possible to map the entire SPH simulation onto the GPU. The third is to render the fluid on the GPU, thus avoiding the slow data exchange between graphics and system memory.

In this paper, we present a new GPU-friendly technique for the SPH simulation. The most important feature of our algorithm is that the entire SPH simulation, including adaptive sampling as well as rendering of the fluid particles, is executed on the GPU. According to our knowledge, this is the first fully GPU-based system which can accomplish all

[†] email: [zhang,solenthaler]@ifi.uzh.ch, pajarola@acm.org

the above mentioned tasks. The main contributions of this paper can be summarized as follows:

1. **Full GPU implementation**: The data dependencies in the SPH simulation are removed so that it can efficiently be mapped onto the GPU. At the same time, the computational model of SPH is changed from *gathering* to *distributing* to make SPH more GPU-friendly.
2. **Adaptive sampling of fluid particles**: The particles in certain regions of interest can be split into multiple sub-particles such that the computation can be focused on these regions. At the same time, the number of particles can be reduced by merging suitable particles, e.g. those deep inside the fluid.
3. **Efficient visualization**: Each surface particle is visualized as a metaball [Bli82]. The intersections between rays and the isosurface defined by the particle metaballs are directly computed on the GPU by a dynamic particle grouping algorithm and efficient equation solver.

2. Related Works

The Lagrangian SPH method, originally designed for the simulation of stars [Mon92], is a powerful alternative to grid-based Eulerian methods. The basic idea of SPH [MCG03] is to use a set of particles to sample and represent the fluid and define a method to smoothly interpolate the sampled attribute fields by a blending kernel function.

SPH-based adaptive sampling is studied in [OVSM98] where particles are split to several sub-particles according to their physical attributes. Most recently, Adams *et al.* proposed a method [APKG07] to resample the fluid based on extended local feature size, which can significantly reduce the number of particles and simulation time. In the standard SPH simulation, the ideal gas equation is used to relate pressure and density. This results in high compressibility, which is not a desirable feature for the simulation of liquids. Some adaptions have been made to enforce incompressibility. Examples include [Mon92,BT07], where a new formula for the computation of pressure is adopted.

In order to fully utilize the computational power and parallelism of GPUs, several GPU implementations of the SPH simulation have been proposed recently. One of the most difficult challenges of a GPU implementation is to find an efficient way to completely perform the required neighborhood search on GPU. Amata *et al.* [AIY*04] proposed a semi-GPU implementation in which they executed the neighbor search on the main CPU and subsequently transferred the neighbor information to the GPU for each time step. A hierarchical dynamic quadtree structure is employed in [KH06] to accelerate the query for closest particles. Harada *et al.* [TH07] defined a 3D grid so that the neighbor particles searching can be implemented by executing texture lookups for neighboring voxels.

With respect to the rendering of fluids, most related work is the rendering of isosurfaces. One of the most often used methods is the Marching Cubes algorithm [LC87] which extracts a polygonal mesh of an isosurface from a 3D scalar field. There are several approaches focusing on the GPU implementation of the Marching Cubes algorithm. For example, Dyken *et al.* implemented a high-speed Marching Cubes algorithm on the GPU, based on the interpretation of Marching Cubes as a stream compaction and expansion process [DZTS07]. [MSD07] presented a screen-space meshes method which only generates triangle meshes for the front most layer of the fluid. For SPH simulation, the metaball approach proposed in [Bli82] potentially provides a better solution, because the concept of metaballs is closely related to the concept of SPH. Both of them employ a kernel function to represent and interpolate point attributes that are smoothed out over a small volume of space. Kooten *et al.* [KvK07] proposed a GPU algorithm to sample the metaballs' implicit surface by constraining free-moving particles to this surface.

3. SPH Fluid Model

In this section, we briefly describe the standard SPH model [Mon92] and subsequently show how these equations can be adapted to allow for a GPU implementation with adaptive particle resolution.

In SPH, the fluid is discretized by particles carrying field quantities A. These quantities can be evaluated at any position $\mathbf{r}$ by summing up the weighted contributions of the neighboring particles b: $A(\mathbf{r}) = \sum_b \frac{m_b}{\rho_b} A_b W(\mathbf{r}-\mathbf{r}_b, h)$, where m_b is the mass of particle b, ρ_b its density, and $W(\mathbf{r}-\mathbf{r_b}, h)$ the weighting kernel with smoothing length h. In the standard formulation, m as well as h are constant throughout the simulation. At each particle position, the density can be computed by

$$\rho_a = \sum_b m_b W(\mathbf{r}_{ab}, h), \tag{1}$$

where $\mathbf{r}_{ab} = \mathbf{r}_a - \mathbf{r}_b$. The pressure P of a particle is given by the modified gas state equation [DC96] $P_a = k(\rho_a - \rho_0)$ where ρ_0 is the rest density of the fluid and k its stiffness. The pressure and viscous forces acting on a particle are directly derived from the Navier-Stokes equations and can be written as [Mon92]

$$\mathbf{F}_a = -\sum_b m_a m_b \left(\frac{P_a}{\rho_a^2} + \frac{P_b}{\rho_b^2} + \Pi_{ab}\right) \nabla W(\mathbf{r}_{ab}, h). \tag{2}$$

For a more detailed description on how the viscosity Π_{ab} can be computed and about SPH in general we refer to [Mon92, Mon05].

Some changes have to be made to the standard SPH model such that it becomes more suitable for the simulation of adaptively sampled fluids and an efficient GPU implementation. The major adaptions include: 1) computing adaptive smoothing lengths to support particles with different masses;

2) modification of the formula for density computation to remove data dependency in SPH; 3) modification of the formula for pressure computation to make the fluid more incompressible; 4) mass correction to make the SPH simulation more stable.

- **Kernel Support**: A fixed kernel support radius is not suitable in our method due to the fact that the adaptive sampling will generate particles of different masses. Equation 3 from [Mon92] is adopted to compute the smoothing support h according to the particle's mass m and density ρ.

$$h = \sigma(\frac{m}{\rho})^{1/3}, \tag{3}$$

where σ is a constant ~ 1.3.

- **Density**: Equation 1 will introduce some problems in the simulation because the density will drop near the boundary of the fluids and the resulting pressure will be instable. In our method, Equation 4 is adopted to compute the density [Mon92], where $v_{ab} = v_a - v_b$:

$$\frac{d\rho_a}{dt} = \sum_b m_b v_{ab} \nabla_a W_{ab} \tag{4}$$

Equation 4 has two important advantages over the standard method. Firstly, it does not have the problem of computing densities for surface particles because it only computes the density change rate. Secondly, Equation 4 results in only one computation pass, whereas the standard SPH requires two because the densities in Equation 2 depend on the results from Equation 1. Equation 4 removes this data dependency by assigning an initial density to each particle and then the computation of density change rate can be combined with the computation of force. This feature is advantageous for a GPU implementation.

- **Pressure**: Equation 5 is employed to compute the pressure, which is quite sensitive to density changes [Mon92, BT07]. A relatively small density change can cause large pressures, and this is helpful to make the fluids incompressible.

$$P = B((\frac{\rho}{\rho_0})^7 - 1) \tag{5}$$

The pressure constant B can be defined as $B = \frac{\rho_0 c_s^2}{7}$ where c_s is the speed of sound in the fluid. Please refer to [Mon92, BT07] for more details.

- **Mass correction**: We may get particles with significant mass differences in the adaptive particle sampling. As shown in Figure 1, suppose a particle p_2 with large mass in the neighborhood region of particle p. Even their distance being fairly large, p_2 may still have a big influence on p because of its big mass value. In order to solve this problem, a mass correction is made before particles contribute to their neighbors. Notice that the neighborhood region of a particle p is a sphere whose radius r_p is defined by the smoothing kernel radius h_p: $r_p = 2h_p$. In our method, the corrected mass m_c of p_2 derived from Equation 6 is proportional to the volume V of the spheres' intersection region.

$$\frac{m_{p_2}}{m_c} = \frac{\frac{4}{3}\pi(2h_{p_2})^3}{V}, \tag{6}$$

where V is computed according to Equation 7.

$$V = \frac{\pi(r_{p_2} + r_p - d)^2(d^2 + 2d(r_p + r_{p_2}) - 3(r_p - r_{p_2})^2)}{12d}, \tag{7}$$

where d is the distance between p and p_2.

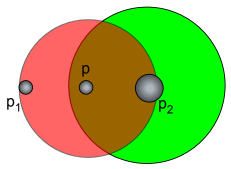

Figure 1: *Mass correction. The corrected mass is computed according to the volume of the spheres' intersection part.*

4. Adaptive Sampling

Adaptive sampling is a powerful method to improve the simulation effectiveness. In our algorithm, particles can be split into multiple sub-particles in geometrically complex regions to provide a more accurate simulation as well as a better visualization. At the same time, nearby particles deep inside the fluid can be merged to a single particle. This merging reduces the number of particles which in turn improves the efficiency of the computation.

Adaptive sampling of dynamic fluids is a very complicated problem. [APKG07] presented a good method to solve this problem. Unfortunately, this method cannot directly be adopted in our algorithm because it contains many expensive operations which are difficult to map onto the GPU. Our goal is to achieve an algorithm to simulate and render fluids on the GPU interactively, so we have to make a good trade-off between physical accuracy and performance.

4.1. Splitting

In our algorithm, the geometrically complex regions are defined as the surface regions of the fluids. The particles near the fluid surface are potential candidates for *splittable particles*. In order to detect splittable particles, *surface particles* have to be identified first. Particle p is considered a part of the surface if its distance to the center of mass χ_p of its neighborhood exceeds a certain threshold. The mass center χ_p can be defined by Equation 8, where x_i is the position of particle i in p's neighborhood, and m_{c_i} is the corrected mass computed from Equation 6.

$$\chi_p = \frac{\sum m_{c_i} x_i}{\sum m_{c_i}} \tag{8}$$

Equation 8 alone, however, is not enough to define the surface particles in some extreme cases. Considering a point p inside a splash, it may be close to its mass center χ_p because the neighbors of p distribute quite symmetrically in space. This problem can be solved by adding the criterion $\frac{\sum m_{c_i}}{m_p} < \delta$ to mark the surface particle, where δ is a user-defined parameter.

A surface particle is not always a splittable particle because we do not want to keep subdividing the surface into too small sub-particles. The splitting operation is stopped once the mass of a surface particle is smaller than some threshold ε_m. We will discuss how to define ε_m later in Section 5 based on our GPU-friendly data structure. Hence from the discussion above, a splittable particle is defined as a surface particle whose mass is bigger than ε_m.

One splittable particle p is split into four sub-particles located at the corners of a tetrahedron centered at p. The physical attributes of the newly generated sub-particles p_i $(i = 1,2,3,4)$ are derived from their parent particle p in the following way:

- The positions of the new sub-particles can be derived from a local coordinate system $\Re$ centered at p. The Z-axis of $\Re$ is defined as $normalize(x_p - \chi_p)$, and X, Y-axis can be derived from $X = normalize(-Z.z, 0, Z.x)$ and $Y = cross(X,Z)$. The distance d from p to p_i is defined as αh_p where α is a user-defined constant;
- The mass of p_i is one fourth of the mass of p;
- The density and velocity of p_i is set to the same corresponding values of p;
- The smoothing kernel length of p_i is computed from Equation 3;

4.2. Merging

Similar to the splitting operation, a special type of particles called *inner particles* are marked to define the merging domain. Based on the notion that inner particles are deep inside the fluid, they can be defined similarly to the definition of the surface particles. Particle p is considered to be an inner particle if it is a non-surface particle and its distance to the center of mass χ_p of its neighborhood is below a certain threshold. For a set of nearby inner particles p_i, they will be merged to a single aggregate particle p_{merged} whose attributes are derived in the following way:

- The mass of p_{merged} is $\sum m_i$;
- The position, density and velocity of p_{merged} is set to the weighted average of the corresponding attributes of p_i, where their masses are used as the weight factor;
- The smoothing kernel length of p_{merged} is again computed from Equation 3;

Note that merged particles have a bigger mass, resulting in increased pressure on other particles preventing them from coming too close, and hence the merging process is stopped.

5. GPU Implementation

The SPH simulation can be parallelized well because there are almost no data dependencies. Since a modern GPU has significant parallel computing power, it makes sense to implement the entire SPH algorithm on the GPU to achieve great improvements in simulation performance. In our algorithm, all SPH simulation stages, including the adaptive sampling and rendering of particles, are executed on the GPU.

In order to implement the SPH simulation on the GPU, some modifications have to be made to the computation. As shown on the left of Figure 2, the computational model of the standard SPH method is a *gathering process*, as the contributions from neighbors are summed up as discussed in Section 3. The basic idea of the GPU implementation, which is a *distribution process*, is illustrated on the right of Figure 2. This process can actually be viewed as a 3D splatting operation. Each particle distributes its contribution to its neighbors, and the contribution is accumulated at the neighbor. After all particles are processed, the attributes of particles can be computed from the accumulated values. The advantages of distribution over gathering will be discussed in more detail later in Section 7.

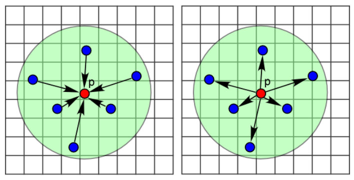

Figure 2: *Gathering vs. distribution.*

5.1. Data Structure

The main challenges of our GPU implementation are: 1) GPU does not directly support 3D splatting so that it has to be converted to some other GPU-supported operations. 2) a dynamic data structure is required to store the particles because the adaptive sampling keeps producing new and removing old particles.

The basic idea to address the first issue is to convert the spherical 3D splat into several 2D splat slices. The 2D splatting operation is well studied in point-based graphics [GP07]. In order to fulfill this conversion, a 3D grid called *bucket space* is defined to cover the simulation space. Each particle is mapped to the closest voxel, and its physical attributes such as position, velocity, density and mass, are stored in the closest voxel. Notice that the physical attributes recorded in the voxel are the original ones instead of the voxelized values, so there are no discretization errors in the mapping process. The 3D grid can further be interpreted as a stack of 2D slices along the Y-axis, and these slices are put together to form a big 2D texture called *slice space texture*. Based on this data structure, a 3D splat can easily be decomposed into multiple 2D splat slices, as shown in Figure 3. For instance, a 3D splat centered at p has an influence region that should be a sphere whose radius is defined by p's smoothing length. The decomposition of a 3D splat is naturally done by the intersections between this sphere and the 2D slices of the bucket space, forming multiple 2D splats with different radii.

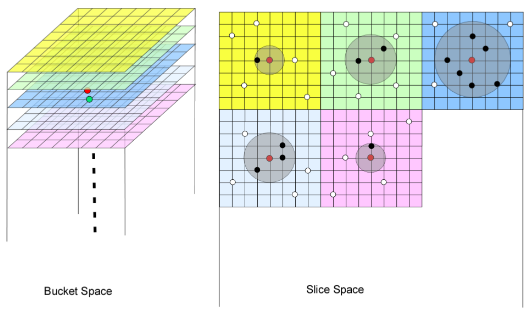

Figure 3: *The decomposition of one 3D splat into several 2D splats. Red dots in the right image represent the 2D splat centers in slices of particle p. Black and white dots represent the particles that are inside and outside the influence region of p respectively.*

A hardware feature called *stream-output stage* introduced in Direct3D 10 is exploited to implement the dynamic data structure to store particles. The purpose of the stream-out stage is to write vertex data streamed out of a geometry shader stage to another vertex buffer. Based on this, two vertex buffers V_1 and V_2 are employed in our method. Suppose the original particles are stored in V_1, and after the adaptive sampling performed in a geometry shader, the new set of particles is written to V_2. V_1 and V_2 are swapped for the next time step. More details will be discussed in Section 5.3

5.2. Adaptive Sampling on the GPU

As mentioned earlier, the splitting operation should be stopped if the particle mass is smaller than ε_m. ε_m can be derived based on our GPU data structure as follows. Because each particle is assigned to the closest voxel cell in the bucket space, the basic requirement of the splitting operation is to prevent that newly generated sub-particles are mapped to the same voxel. This in turn means that the subdivision exceeds the resolution limitation of the bucket space data structure. Based on this observation, the distance between the new sub-particles $d_s = \sqrt{\frac{8}{3}}\alpha h$ should be larger than $\sqrt{3}s$ where s is the grid size of the bucket space. Combining this with Equation 3 we get:

$$\sqrt{\frac{8}{3}}\alpha\sigma\left(\frac{\varepsilon_m}{\rho}\right)^{1/3} \geq \sqrt{3}s \Rightarrow \varepsilon_m \geq \rho\left(\frac{\sqrt{9/8}s}{\alpha\sigma}\right)^3 \tag{9}$$

With this, the implementation of the splitting operation on the GPU is straightforward. It is performed in a geometry shader at the end of every time step, after the particle attributes have been updated. Then, the output of the splitting operation is directed to another vertex buffer by *stream-output stage*.

The merging operation itself is simpler than splitting, but its GPU implementation is more tricky because of the following issues. The first problem is how to efficiently find all nearby inner particles. The straightforward solution to this would be to search the neighborhood for each particle after it has been mapped to the bucket space. This, however, is very inefficient since it requires computing distances between particles. Our strategy is to change the rasterization rule for inner particles. Instead of mapping inner particles to the closest voxel in the bucket space, they are mapped to the closest voxel with even coordinates. As shown in Figure 4, inner particles p_i $(i = 1,2,3,4,5)$ in red color are not mapped to their corresponding closest voxel marked by the dark particle with a cross, but instead are mapped to the closest voxel with even coordinates $(4,4)$ so they can be merged into a single particle. This strategy has the same effect as a coarser grid definition would have for the mapping of inner particles.

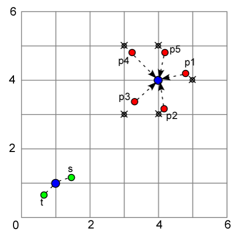

Figure 4: *Merging of particles on the GPU in the bucket space texture data structure.*

The second issue is how to merge particles. A two-pass algorithm with two set of textures T_{B1} and T_{B2} is employed to address this issue. In the first pass, all particles are mapped to their closest voxel by using T_{B1} as an accumulation buffer. If more than one particle is mapped to the same voxel, their attributes will be accumulated in T_{B1}. For example, the mass of particles will be accumulated to $\sum m_i$, and other attributes A, such as density and velocity, will be mass weighted and accumulated $\sum m_i A_i$. The subsequent second pass is an image pass to normalize the density and velocity by the accumulated mass as $\frac{\sum m_i A_i}{\sum m_i}$ and the result will be written to T_{B2}.

The third challenge of the GPU implementation is how to remove the obsolete particles. Suppose four particles p_i $(i = 1,2,3,4)$ are being merged to a single particle. Only one particle representing the merged particle can be kept and the other three have to removed. An extra texture T_{ID} called *PrimitiveID texture* is introduced to solve this problem. A unique primitive ID for each particle is employed and stored in T_{ID}. This unique ID can directly be adopted from SV_PrimitiveID or SV_VertexID, which are two systemsupplied values in Direct3D 10. When T_{ID} is updated, the depth test function is set to pass all fragments so that only the last written fragment is kept for a single texel in T_{ID}. In the subsequent stages, the primitive ID for each particle will be compared to the ID recorded in T_{ID}. The particles will be discarded as obsolete if their ID does not match.

While our splitting strategy can guarantee that no newly generated sub-particles from the same parent particle will be mapped to the same voxel, sub-particles from different parent particles or some non-inner particles may still be mapped to the same voxel. This is shown in Figure 4, where two green particles s and t are mapped to the same bucket cell. This situation suggests that the particle positions exceed the resolution limit of our data structure. Simply increasing the grid resolution cannot completely solve this problem. Our strategy to address this issue is to merge them to a single particle. Fortunately, this case does not happen frequently at all, because the pressure force Equation 5 is very sensitive to density changes. A relatively small density change can cause large pressures which prevent two particles coming too close.

5.3. Simulation Stages

Based on the above discussions, we eventually arrive at a four-pass rendering algorithm for GPU-accelerated SPH simulation as shown in Figure 5.

1. **Distributing particles**: Each particle is rasterized as one texel in the bucket slice space texture T_{B1}. If multiple particles are mapped to the same texel, their attributes are accumulated, using their mass as the accumulation weight. At the same time, the unique primitive ID of each particle is also computed and recorded in texture T_{ID}.
2. **Merging particles**: This is an image processing pass to normalize the attributes of merged particles and output results to another texture T_{B2}.
3. **3D splatting**: The 3D splat is first decomposed into a set of 2D splats in the geometry shader. Then the contribution from the particle to its neighbors covered by a set of 2D splats is computed and accumulated at the corresponding texels of its neighbors.
4. **Updating and splitting particles**: This rendering pass solves two tasks: First, the attributes of each particle are computed in the vertex shader according to the values stored in the textures. Note that obsolete particles are discarded by comparing their primitive ID to the value recorded in T_{ID}. Additionally, a splitting operation is performed in the geometry shader. All particles that satisfy the splitting criteria are split into four sub-particles. The remaining set of original particles and the newly split sub-particles are streamed out into a vertex buffer.

5.4. Optimization: Early Z-Culling

Note that a fine 3D grid is preferred to represent the simulation domain because it results in a smaller threshold ε_m, according to Equation 9, that allows the adaptive sampling to produce smaller particles. The side effect of a fine 3D grid is that it introduces more empty voxels with no corresponding particles, especially near particles with larger mass. These empty voxels potentially waste memory and computation time.In the ideal case, the fragments corresponding to the empty voxels should be prevented from being processed. The early Z-culling technique can help us to achieve this, by using the following steps:

1. The Z-buffer is initialized to value 0;
2. In the *distribution* pass, the depth function is set to allow all fragments to pass the Z-test. Each particle is assigned a constant depth value $d_1 (0 < d_1 < 1)$. After this pass, voxels containing real particles have a depth value of d_1, while empty voxels still hold a value of 0.
3. In the *3D splatting* pass, the depth function is set to only allow fragments with smaller depth values to pass the Z-test. Fragments generated during this 3D splatting are all assigned another constant depth value $d_2 (1 > d_1 > d_2 > 0)$. Hence the early Z-culling will discard all fragments corresponding to empty voxels.

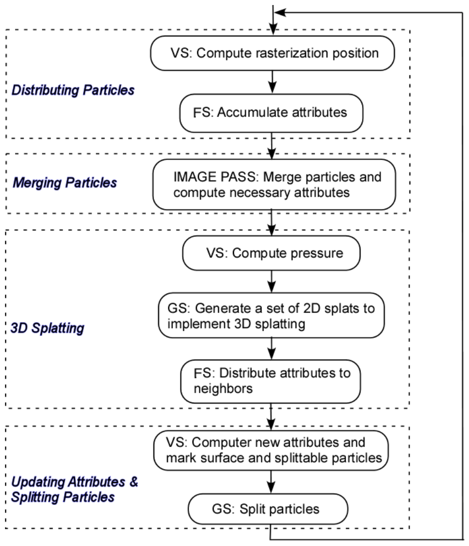
Distributing Particles
VS: Compute rasterization position
FS: Accumulate attributes
Merging Particles
IMAGE PASS: Merge particles and compute necessary attributes
3D Splatting
VS: Compute pressure
GS: Generate a set of 2D splats to implement 3D splatting
FS: Distribute attributes to neighbors
Updating Attributes & Splitting Particles
VS: Computer new attributes and mark surface and splittable particles
GS: Split particles

Figure 5: *Overview of the GPU implementation of the SPH simulation. VS, GS, FS stand for vertex, geometry and fragment shader respectively.*

6. Rendering Surface Particles

In our algorithm, each surface particle is represented as a metaball [Bli82], and the fluid is rendered by directly computing the intersection between a ray and the isosurface determined by the surface metaball particles. In order to improve the rendering efficiency, an extra rendering pass is executed before computing the ray-isosurface intersection. In that extra pass, surface particles are identified and stored in a second vertex buffer. Rendering is only performed on the second vertex buffer, and thus, non-surface particles do not enter the rendering pipeline. This is also accomplished by exploiting the stream-output stage feature.

6.1. Metaball and Isosurface Function

The metaball function Equation 10 is defined on the normalized distance $x = \frac{d}{r_s}$, where d is the distance to the center of the metaball and r_s is the support radius. Note that r_s should be defined as a function of the smoothing kernel length h of the corresponding particle. However, we simply use $r_s = c_1 h$, where c_1 is a user-defined constant.

$$f(x) = He^{-\frac{x^2}{2\sigma^2}} \tag{10}$$

Based on the metaball function, the isosurface function F is defined as Equation 11, with C as user-defined threshold.

$$F = \sum He^{-\frac{x_i^2}{2\sigma^2}} = C \tag{11}$$

In our algorithm, C is defined as the value of the metaball function at $c_2 r_s$, where c_2 is also a user-defined constant ($c_2 = 0.85$ in our implementation). In order to achieve good rendering results, the two parameters H and σ in Equation 10 have to be chosen carefully. H and σ are designed to meet the following requirements:

1. The value of the metaball function in regions outside of the support radius should be fairly small with respect to C, so that those regions can be safely ignored for the computation of ray-isosurface intersections. This requirement can be satisfied by setting σ according to Equation 12, with $c_3 = 10$, as this means that the metaball function value at the support radius is only one tenth of the isosurface value C.
$$\frac{f(c_2)}{f(1)} = c_3 \Rightarrow \sigma = \sqrt{\frac{1 - c_2 * c_2}{2 \ln c_3}} \tag{12}$$
2. In order to reduce the errors in the computation of ray-isosurface intersections, the derivative value near $c_2 r_s$ should be set to a high enough value c_4. This requirement is accomplished by setting H according to Equation 13
$$f'(c_2) = c_4 \Rightarrow H = \frac{c_4 \sigma^2}{c_2 e^{-\frac{c_2}{2\sigma^2}}} \tag{13}$$

6.2. Ray-Isosurface Intersection

The computation of the intersection between the ray $E + Rt$ and the metaball isosurface can be formulated to solve Equation 14, where E is eye point and R is the normalized ray direction.

$$F(t) = \sum He^{-\frac{(E+Rt-p_i)^2}{2\sigma^2 r_{s_i}^2}} = C \tag{14}$$

The basic idea to solve this equation is to search along the ray using a binary search algorithm. Given t_{low} and t_{high} which satisfy $F(t_{low}) <= C$ and $F(t_{high}) >= C$, it can be guaranteed that the solution t for Equation 14 lies between t_{low} and t_{high}. If the isosurface function value at $t_{mid} = (t_{low} + t_{high})/2$ is below C, t_{low} is replaced, otherwise t_{high}. This iteration continues until the solution for Equation 14 is found.

Similar to the situation in point-based rendering where one pixel is always covered by multiple splats, there are always multiple metaballs M_i that have contributions to the isosurface function at point x. The most difficult problem in the evaluation of the isosurface function F at point x is that we do not know which metaballs are involved. This can be solved by a dynamic point grouping algorithm as proposed in [ZP07]. The basic idea is to separate the overlaps between points/metaballs by partitioning them into multiple non-overlapping groups and render each group to a different texture as shown in Figure 6. The dynamic grouping algorithm is GPU-based and can handle dynamic particles, so costly particle-readbacks to CPU can be avoided. Please refer to [ZP07] for more details.

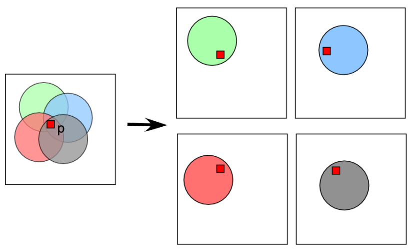

Figure 6: *Metaballs are divided to multiple non-overlapping groups and each group is rendered to a different texture. Based on this technique, all metaball sets M_i that have contributions to point p on the isosurface can be recorded.*

The basic fluid visualization algorithm contains two rendering passes. At first a geometry pass is executed to partition particles into K non-overlapping groups. Each group then is rendered to a different texture. At the same time, the intersections between ray and spheres defined by each particle are computed and will be used to create the initial t_{low} and t_{high} for the intersection between a ray and isosurface. Then, an image pass is executed to compute the final intersections in the fragment shader.

In the first pass, two spheres S_p and S'_p are defined for each surface particle p, and the intersections between a ray and these spheres are computed. Both S_p and S'_p are centered at p, but with different radii, r_s and $c_2 r_s$ respectively. r_s is the support radius for particle p and c_2 is the same constant defined in Section 6.1. Surface particle p is rendered as a quad Q covering the sphere S_p. Rays with form $E + Rt$ are casted from the eye point for all fragments covered by Q, where E and R are the eye point and the normalized ray direction respectively. Note that because of Equation 12, a fragment can be safely discarded if its corresponding ray has no intersection with S_p, and according to the definition of S'_p, it can be guaranteed that the intersection t'_p between the ray and S'_p satisfies $F(t'_p) >= C$. This can be used to define the initial t_{high} in the next stage. A two metaball example is as shown in Figure 7, where the red curve represents the isosurface defined by p_1 and p_2, and the blue and green circles stand for S_{p_i} and S'_{p_i} respectively. Note that (t_{p_1}, t'_{p_1}) and (t_{p_2}, t'_{p_2}) will be written to different textures because of the dynamic particle grouping.

In the second pass, the set of intersections $(t_{p_k}, t'_{p_k})(k = 1,2..,K)$ generated in the first pass can be read from textures for pixel a. The initial t_{low} and t_{high} can be derived from $t_{high} = \min t'_{p_k}$ and $t_{low} = \min(t_{p_k})$. Only the set of metaballs M_i satisfying $t_i \leq t_{high}$ is involved in solving Equation 14. For the example shown in Figure 7, $t_{high} = t'_{p_1}, t_{low} = t_{p_1}$, and both p_1 and p_2 will be involved in the ray-isosurface intersection computation because of $t_{p_1} < t_{high}, t_{p_2} < t_{high}$.

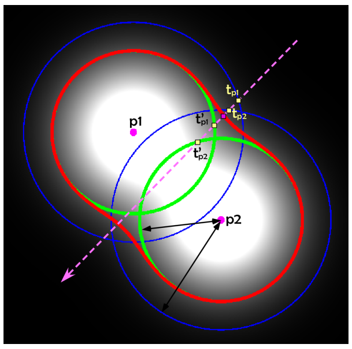

Figure 7: *Computation of ray-isosurface intersection.*

In most cases, the intersections between a ray and the two spheres S and S' can be found so that the initial t_{low} and t_{high} can be defined. If there is no intersection between the ray and S/S', one of the following two situations applies:

- If the ray has no intersection with sphere S, it is guaranteed that there is no intersection between a ray and the isosurface.
- If the ray intersects with S but not with S', only the initial t_{low} can be defined. In order to try to find the initial value for t_{high}, we search along the ray, starting with t_{low}. Iteratively, t_{low} is increased by a small amount $t_{low} = t_{low} + \Delta$ and the isosurface function F is evaluated at the new t_{low}. If $F(t_{low}) > C$, the initial t_{high} is set to current t_{low} and the division search can be started. If no value t that satisfies $F(t) > C$ has been found after several steps, we assume that there is no intersection between a ray and the isosurface.

7. Results and Discussions

The method proposed in this paper has been implemented on a PC with a NVIDIA Geforce 8800GTX GPU.

We tested our algorithm with three differently-sized particle sets. The simulation results are as shown in Table 1. The first column shows the initial size of the particle set. During the simulation, the number of particles varies because of the adaptive sampling of the fluids. The average number of particles is shown in the second column. The performance of our algorithm is measured by the number of time steps that can be executed in one second. The third and fourth columns show the number of time steps that can be executed in one second without and with rendering of the fluid respectively. Note that for smooth rendering it is not necessary to render the particles every time step. In our tests, we only render the particles every 20 time steps.

ini-size	avg-size	sim	sim+visualize
3,003	7,463	129.71	49.01
6,992	14,002	75.61	37.28
19,964	52,746	19.34	8.21

Table 1: *Simulation data for three differently-sized particle sets.*

More detailed experimental results are shown in Figure 8. The top image shows how the number of particles varies in the simulation period. It can be seen that the number of particles increases from 20K to 75K in the first few time steps because of the adaptive sampling. The middle image shows the number of time steps that can be executed in one second. It depends on the number of particles, and the performance of early Z-culling also has some impact on it. The bottom image shows the average, maximum and minimum density during the simulation. It can be seen that the average density is quite stable in the whole simulation period. Even the maximum and minimum densities are quite close to the rest density due to the density computation method described in Equation 4. Two screenshots are as shown in Figure 9.

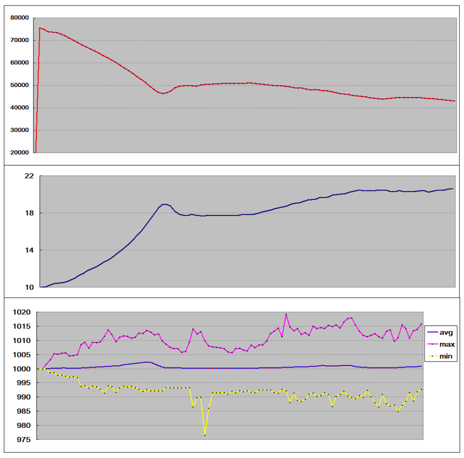

Figure 8: *Detailed experimental data. Top: Number of particles. Middle: The number of time steps that can be executed in one second; Bottom: Density.*

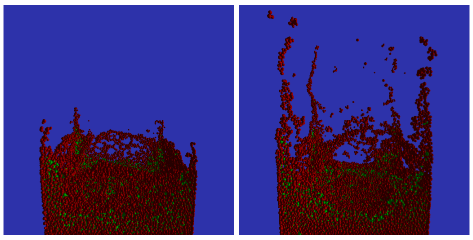

Figure 9: *Screenshots for adaptive sampling. Red and green particles represent surface and inner particles respectively.*

7.1. Simulation Performance

With respect the the simulation performance, we compare our method to the most recent results from [TH07], which can simulate 60K particles at about 17 time steps per second. Table 1 shows that our method is close to theirs. Notice that their method does not support adaptive sampling which is quite expensive on the GPU. For example, a dynamic data structure obviously has some performance penalty if compared to a static data structure. Additionally, all attributes in the SPH simulation have to be computed on the fly because of the adaptive sampling, whereas many of them are constants in a fixed sampling method.

In order to have a fair comparison to [TH07], we disabled the adaptive sampling and related computations. Four differently-sized particle sets are employed to test our fixed sampling version algorithm. The results are as shown in Table 2. It can be seen that our GPU implementation is about 2.5 times faster than [TH07]. As shown in the last row of Table 2, the performance can be further improved by running the simulation at half-floating-point precision (FP16) at the cost of physical accuracy. The differences of running the simulation at the precision of 32-bit floating point (FP32) and FP16 are as shown in Figure 10. It can be seen that most of the particles behave similarly.

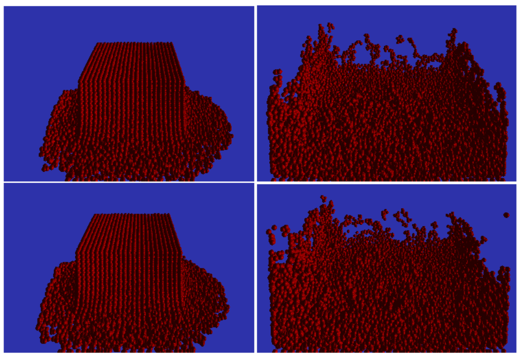

Figure 10: *The differences between running simulation at FP32 and FP16 precision. Most of the particles behave quite similarly. Top row: FP32 mode. Bottom row: FP16 mode*

The reasons for the improved performance compared to [TH07] are:

1. Our distribution model has a big advantage over the gathering model used in [TH07]. In the gathering mode, neighborhood searching is necessary, which is accomplished by executing a large number of texture lookups over the nearby texels. In our distribution mode, an early z-culling technique can be employed to cull the empty texels so that no computations are wasted.
2. Because of Equation 4, data dependencies are removed so that the density computation in our method can be executed in the same pass as the computation of the force.

size	6,992	19,964	61,336	127,452
FP32	388.00	126.69	43.16	21.18
FP16	513.01	172.59	57.76	27.12

Table 2: *The simulation performance of our fixed sampling method. The numbers in the last two rows represent the number of time steps that can be simulated in one second.*

The biggest drawback of our GPU implementation is that our method consumes lots of memory. Uniform grids are employed to store all the physical attributes as well as the intermediate results. Actually most of the grid cells are empty, which introduces a huge waste of graphics memory. Using low resolution grids of course reduces the memory cost, but also put more limits on particle splitting and makes more particles to be merged.

7.2. Rendering

The rendering results are shown in Figure 11. In the two images in the top row, some artifacts are visible due to the low number of particles. To reduce the bumpy appearance, a fairly big metaball radius has to be used, at the cost of very thick surfaces. With more particles, our rendering algorithm can produce more realistic results, as shown in the two images of the bottom row.

Compared to the Marching Cubes algorithm whose rendering quality relates heavily to the subdivision grid size, our method does not require space subdivision. Additionally, our method has a performance advantage over the Marching Cubes algorithm because its complexity only depends on the fluid surface area, and not on the fluid volume. Compared to the point-based rendering algorithm that treats each particle as a 2D splat, our method produces better rendering results because the point-based method requires estimating the normal vector for each splat, which is quite instable. Unfortunately, there are also some drawbacks in rendering particles as metaballs. The support radius of the metaballs has to be chosen carefully, otherwise some bumpy artifacts will appear. Employing a big support radius will alleviate the problem, but may result in too thick fluid volume.

8. Conclusions

In this paper, we have presented a novel algorithm to simulate and visualize particles on the GPU. The efficiency of

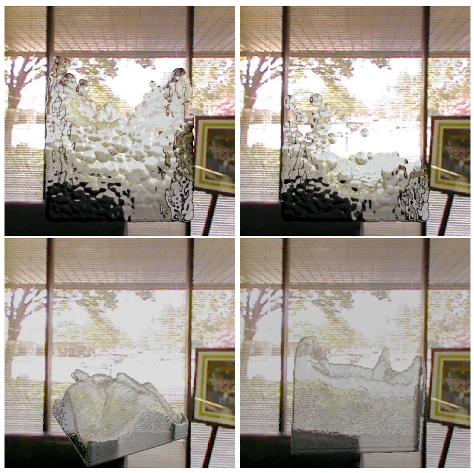

Figure 11: *Rendering results. Top row: Rendering with only a few surface particles. Bottom row: Rendering with more surface particles.*

the SPH simulation is improved by combining adaptive sampling of the fluids and ray-isosurface intersection computation in a GPU-based algorithm.

Unfortunately, some problems remain when using our method and they will be addressed in the future work. Using a 3D uniform grid to represent the simulation domain is costly in terms of texture memory. A possible solution to this problem would be the use of an adaptive data structure. Additionally, improvements to our rendering algorithm might reduce bumpy appearance and thick fluid volume for low number of particles.

9. Acknowledgements

This work was partially supported by the Swiss National Science Foundation Grant 200021-111746/1.

References

[AIY*04] Amada T., Imura M., Yasumoto Y., Yamabe Y., Chihara K.: Particle-based fluid simulation on gpu. In *ACM Workshop on General-Purpose Computing on Graphics Processors* (2004).

[APKG07] Adams B., Pauly M., Keiser R., Guibas L. J.: Adaptively sampled particle fluids. In *Proceedings ACM SIGGRAPH* (New York, NY, USA, 2007), ACM Press, pp. 48–54.

[Bli82] Blinn J. F.: A generalization of algebraic surface drawing. *ACM Trans. Graph. 1*, 3 (1982), 235–256.

[BT07] Becker M., Teschner M.: Weakly compressible sph for free surface flows. In *Symposium on Computer Animation* (2007), pp. 209–217.

[CMT04] Carlson M., Mucha P. J., Turk G.: Rigid fluid: animating the interplay between rigid bodies and fluid. In *Proceedings ACM SIGGRAPH* (2004), pp. 377–384.

[DC96] Desbrun M., Cani M.-P.: Smoothed particles: A new paradigm for animating highly deformable bodies. In *Eurographics Workshop on Computer Animation and Simulation* (1996), pp. 61–76.

[DZTS07] Dyken C., Ziegler G., Theobalt C., Seidel H.-P.: *GPU Marching Cubes on Shader Model 3.0 and 4.0*. Research Report MPI-I-2007-4-006, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, August 2007.

[GP07] Gross M. H., Pfister H. (Eds.): *Point-Based Graphics*. Series in Computer Graphics. Morgan Kaufmann Publishers, 2007.

[IGLF06] Irving G., Guendelman E., Losasso F., Fedkiw R.: Efficient simulation of large bodies of water by coupling two and three dimensional techniques. In *Proceedings ACM SIGGRAPH* (New York, NY, USA, 2006), ACM Press, pp. 805–811.

[KFCO06] Klingner B. M., Feldman B. E., Chentanez N., O'Brien J. F.: Fluid animation with dynamic meshes. In *Proceedings ACM SIGGRAPH* (New York, NY, USA, 2006), ACM Press, pp. 820–825.

[KH06] Kyle Hegeman Nathan A. Carr G. S. P. M.: Particle-based fluid simulation on the gpu. In *International Conference on Computational Science (4)* (2006), pp. 228–235.

[KvK07] Kees van Kooten Gino van den Bergen A. T.: Point-based visualization of metaballs on a gpu. *GPU Gems III* (2007).

[LC87] Lorensen W. E., Cline H. E.: Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings ACM SIGGRAPH* (New York, NY, USA, July 1987), vol. 21, ACM Press, pp. 163–169.

[MCG03] Müller M., Charypar D., Gross M.: Particle-based fluid simulation for interactive applications. In *Symposium on Computer Animation* (2003), pp. 154–159.

[Mon92] Monaghan J.: Smoothed particle hydrodynamics. *Annu. Rev. Astron. Physics 30* (1992), 543.

[Mon05] Monaghan J.: Smoothed particle hydrodynamics. *Rep. Prog. Phys. 68* (2005), 1703–1759.

[MSD07] Müller M., Schirm S., Duthaler S.: Screen space meshes. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation* (Aire-la-Ville, Switzerland, Switzerland, 2007), Eurographics Association, pp. 9–15.

[OVSM98] Owen J., Villumsen J., Shapiro P., Martel H.: Adaptive smoothed particle hydrodynamics: Methodology ii. *Astrophys. J. Suppl. Ser. 116* (1998), 155–209.

[TH07] Takahiro Harada Seiichi Koshizuka Y. K.: Smoothed particle hydrodynamics on gpus. In *Computer Graphics International* (2007).

[ZP07] Zhang Y., Pajarola R.: Deferred blending: Image composition for single-pass point rendering. *Comput. Graph. 31*, 2 (2007), 175–189.