Simple Rasterization-based Liquids

Martin Guay

Rasterization pipelines are ubiquitous today. They are found in most of our personal computers as well as in smaller, hand-held devices—like smart phones—with lower-end hardware. While Compute Shaders allow much more flexibility, they are only available on higher-end material, and therefore still have limited scope. Simulating particle-based liquids requires sorting the particles which is cumbersome when using a rasterization pipeline.

In this chapter, we describe a method to simulate liquids *without* having to sort the particles. Our method was specifically designed for these architectures and low shader model specifications (starting from shader model 3 for 3D liquids). Instead of sorting the particles, we splat them onto a grid (*i.e.* a 3D or 2D texture) and solve the inter-particle dynamics *directly* on the grid. Splatting is simple to perform in a rasterization pipeline, but can also be costly. Thanks to the simplified pass on the grid, we only need to splat the particles once.

The grid also provides additional benefits: we can easily add artificial obstacles for the particles to interact with, we can ray cast the grid directly to render the liquid surface, and we can even gain a speed up over sort-based liquid solvers—such as the optimized solver found in the DirectX 11 SDK.

1.1 Introduction

Simulating liquids requires dealing with two phases of *fluid*—the liquid and the air—which can be tricky to model as special care may be required for the interface between the two phases depending on the fluid model. In computer graphics, there are mainly two popular formulations for *fluids*: strongly incompressible and weakly incompressible.

The strong formulation is usually more complex as it requires a hard constraint (*e.g.* solving a Poisson eq.), but is more accurate and therefore more visually pleasing. Because it is more complex, it is often used along simple, regular *grid* discretizations [Stam 99]. For liquids, several intermediate steps are required for the surface to behave adequately [Enright et al. 02]. Implementing these steps using rasterization APIs is challenging. For instance, [Crane et al. 07] only partially implements them and the fluid behaves more like a single phase fluid. Furthermore, the strong formulation requires a surface representation like a *level-set* density field which requires its own set of specificities (re-initialisation). Again, in [Crane et al. 07] the level-set method is only partially implemented and had to be hacked into staying at a certain height; preventing them from generating such scenarios as the water jet shown in Fig. 1.3.

The weak formulation on the other hand, requires only a simple soft constraint to keep the fluid from compressing. It is much simpler, but also less accurate. It is often used along particle discretizations and *mesh-free* numerical schemes like *Smooth Particles Hydrodynamics* (SPH) [Desbrun and Cani 96]. The advantage of the weak formulation along particles is really for liquids. This combination allowed reproducing the behavior of liquids without computing any surface boundary conditions similar to [Enright et al. 02]. Additionally, the particles can be used directly to render the liquid surface and there is no need for a level-set. The drawback however, is that particles require finding their neighbors, in order to compute forces ensuring they keep at a minimal distance. Typically buckets or other spacial sorting algorithms are used to cluster the particles into groups [Amada et al. 04, Rozen et al. 08, Bayraktar et al. 08], which can be cumbersome to implement using rasterization APIs.

When using rasterization APIs, we find it more natural to *rasterize*, or *splat* the particles onto a grid instead of spatially sorting the particles. Then, we use the simplicity of the *grid* and finite difference to compute all the forces—including the *soft* incompressibility constraint—in a *single* pass. Some have considered splatting before [Kolb and Cuntz 05], but had to splat for each force (*Pressure and Viscosity*), while we only need to splat once—for all the forces. Finally, the particles are corrected and moved by sampling the grid—which in turn can also be used *directly* to render the liquid surface by ray casting the splatted *density.* Overall, our method allows treating each particle independently while making sure they automatically repulse one-another and avoid rigid obstacles in the domain.

1.2 Simple Liquid Model

Our liquid model is best described as follows: particles are allowed to move freely in the domain while their mass and kinetic energy remains conserved. For instance, if we add gravity it should translate into velocity $\mathbf{u}$. To keep the particles from interpenetrating, we add an incompressibility constraint P derived from the density ρ of particles and the resulting force is the negative gradient of P:

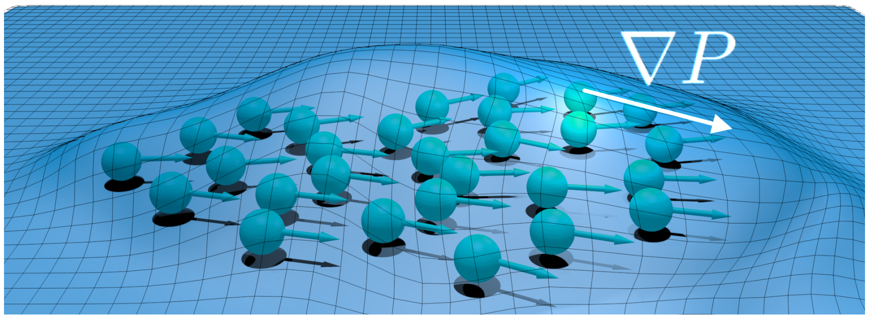


Figure 1.1. Illustrating the scalar density field used to defined the pressure constraint. The pressure force is proportional to the density gradient pushing the particles towards minimum density. For simplicity, we show the idea in 2D with density shown as a height field—which, can also be used as the liquid surface in *Shallow Water* simulations (see Fig. 1.4).

$$\frac{D\rho}{Dt} = 0, \tag{1.1}$$

$$\frac{D\mathbf{u}}{Dt} = -\nabla P + \mathbf{g} + \mathbf{f}_{ext}, \tag{1.2}$$

where $\mathbf{g}$ gravity and $\mathbf{f}_{ext}$ accounts for external forces such as user interactions. The terms $\frac{D\rho}{Dt}$ and $\frac{D\mathbf{u}}{Dt}$ account for the transport of density and velocity. There is never any density added nor removed, it is only transported and held by the particles leading to $\frac{D\rho}{Dt} = 0$. Energy added to the velocity—like gravity—needs to be conserved as well, resulting in equation (1.2). Next we define the incompressibility constraint P, and how to compute the density of particles ρ.

1.2.1 Pressure Constraint

To keep the fluid from compressing and the particles from inter-penetrating, we penalize high density: $P = k\rho$ [Desbrun and Cani 96], where k is a stiffness parameter that makes the particles repulse one another more strongly (but can also make the simulation unstable if too large). Hence, by minimizing the density, the particles will move in the direction that reduces density the most and thereby avoiding each other. At the same time, gravity and boundary conditions like the walls, will act as to keep the particles from simply flying away.

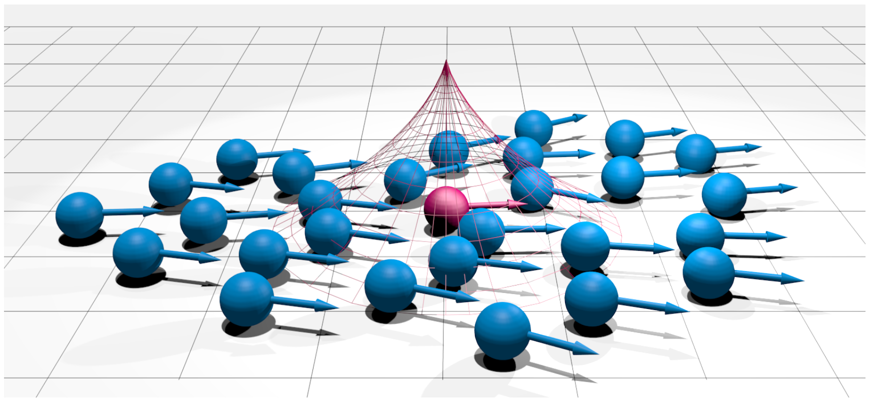


Figure 1.2. Splatting consists in *rasterizing* the smooth kernel function, the 2D case shown here in red. In Fig. 1.1, we see the sum of all the kernel functions; a scalar field representing the density of particles.

Keeping the particles from interpenetrating is crucial in particle-based liquid simulations. To give strict response to the particles that are close to colliding, we can make the density nonlinear leading to the pressure constraint [Becker and Teschner 07]: $P = k\left(\frac{\rho}{\rho_0}\right)^{\gamma}$, where γ is an integer (5 in our case). Dividing by the initial density ρ_0 comes in handy for stability; it becomes easier to control the magnitude of the force through the parameter k and thereby keeping the simulation stable.

Setting the initial ρ_0 can be tricky. It should be proportional an evaluation of ρ at an initial rest configuration; *i.e.* with a uniform distribution of particles. In practice however, we can set it manually. We approximate ρ by performing a convolution, which we pre-compute on a texture by rasterizing, or splatting the kernels of each particle.

1.3 Splatting

To evaluate the density of particles smoothly, we perform a convolution: the density is the weighted average of the surrounding discrete samples; in this case the particles. The weight is given by a kernel function which falls off exponentially with distance, as shown in Fig. 1.2. Instead of sampling the nearby particles, we rasterize the kernel function centered at each particle. The final result is a smooth density grid (texture)—like the one shown in Fig. 1.1—that is equivalent to a convolution evaluation at each point on the texture. We could say also that we now have a virtual particle on each grid cell.

1.3.1 Rasterizing Kernel Functions

To update the velocity on the grid, we need to transfer both the density of the particles—to compute pressure—and their velocity. Hence, the first step in our algorithm is to rasterize the smooth kernel function (red in Fig. 1.2) and the weighted velocity of each particle. We render the particles as points and create quad slices—spanning a cube—in the geometry shader. For each corner vertex $\mathbf{x}_i$, we write the distance $d = \|\mathbf{x}_i - \mathbf{x}_p\|$ to the center of the particle $\mathbf{x}_p$, and let the rasterizer perform the interpolation between vertices. Then, in a pixel shader, we render the smooth kernel value $w(d, r)$ to the alpha channel, and the weighted velocity $w(d, r)\mathbf{u}_p$ to the other 3 channels—in an additive fashion. Finally, the density on the grid can be sampled by multiplying the sum of kernel weights by the mass, and the velocity, by dividing the sum of weighted velocities by the sum of weights:

$$\rho(\mathbf{x}_i) = m_i \sum_p w\left(\|\mathbf{x}_i - \mathbf{x}_p\|, r\right) \qquad \mathbf{u}(\mathbf{x}_i) = \frac{\sum_p w\left(\|\mathbf{x}_i - \mathbf{x}_p\|, r\right)\mathbf{u}_p}{\sum_p w\left(\|\mathbf{x}_i - \mathbf{x}_p\|, r\right)},$$

where i denotes texture indices, p particle indices, and r the kernel radius. We used the following convolution kernel:

$$w(d, r) = \left(1 - \frac{d^2}{r^2}\right)^3.$$

Next we update the velocity field on the grid as to make the particles move in a direction that keeps them from compressing; by computing a pressure force from the density of particles and adding it to the velocity.

1.4 Grid Pass

In the grid pass, we update the splatted velocity field with the pressure force, gravity, and artificial pressure for obstacles (see Section 1.6). We compute the pressure gradient using a finite difference approximation and add forces to the velocity field using forward Euler integration:

$$\mathbf{u}^{n+1} = \mathbf{u}^n - \Delta t \left(\frac{P(x_{i+1}) - P(x_{i-1})}{\Delta x}, \frac{P(x_{j+1}) - P(x_{j-1})}{\Delta y}\right), \qquad (1.3)$$

where Δt is the time step, Δx the spatial resolution of grid, and n the temporal state of the simulation.

While we update the velocity, we set the velocity on the boundary cells of the grid to a no-slip boundary condition by setting the component of the velocity which is normal to the boundary to 0. This is a simple boundary test; before writing the final velocity value, we check if the neighbor is a boundary and set the component of the velocity in that direction to 0.

1.5 Particle Update

We update the position and velocity of particles following the Particle-In-Cell (PIC) and Fluid-In-Particle (FLIP) approaches that mix particles and grids [Zhu and Bridson 05]. The main idea with these numerical schemes is that instead of sampling the grid to assign new values (*e.g.* velocities) to the particles, we can recover only the differences to their original values.

1.5.1 Particle Velocity

In PIC, the particle's velocity is taken directly from the grid, which tends to be very dissipative, viscous and leads to damped flow. For more lively features and better energy conservation, FLIP assigns only the *difference* in velocities; the difference between the splatted velocity and the updated splatted velocity discussed in Section 1.4.

By combining both PIC and FLIP, the liquid can be made very viscous like melting wax, as it can also be made very energetic like water. A parameter r lets the user control the amount of each:

$$\begin{aligned}\mathbf{u}_p = r\mathbf{u}^{n+1}(\mathbf{x}_p) + (1-r)(\mathbf{u}_p - \Delta\mathbf{u}),\\ \text{with } \Delta\mathbf{u} = \mathbf{u}^{n}(\mathbf{x}_p) - \mathbf{u}^{n+1}(\mathbf{x}_p),\end{aligned}$$

where $\mathbf{u}^n$ and $\mathbf{u}^{n+1}$ are grid velocities before and after the velocity update in Section 1.4, and $\mathbf{x}_p, \mathbf{u}_p$ are the particle's position and velocity. Using a bit of PIC ($r = 0.05$) is useful in stabilizing the simulation performed with explicit Euler integration, which can become unstable if the time step is too large.

1.5.2 Particle Position

While we update the particle velocity, we also update the particle positions. We integrate the particle position using two intermediate steps of Runge-Kutta 2 (RK2); each time sampling the velocity on the grid. The second order scheme is only approximate in our case because the velocity field on the grid is kept constant during integration. At each intermediate step, we keep the particles from leaving the domain by clamping their positions near the box boundaries:

$$\mathbf{x}_p^{n+1} = \Delta t \mathbf{u}^{n+1}(\mathbf{x}_p^{n+\frac{1}{2}}) \quad \text{with} \quad \mathbf{x}_p^{n+\frac{1}{2}} = 0.5\Delta t \mathbf{u}^{n+1}(\mathbf{x}_p^n).$$

Note that we never modify the density value of the particles.

1.6 Rigid Obstacles

We can prevent the particles from penetrating rigid obstacles in the domain by adding an artificial pressure constraint where the objects are. This follows the same logic as with the particle density; we define a smooth distance field to the surface of the object which can also be viewed as a density field. We can use analytical functions for primitives like spheres to approximate the shape of the objects. This avoids rasterizing volumes or voxelizing meshes on the GPU. Alternatively, one could approximate shapes using *Metaballs* [Blinn 82], and implicit surfaces which naturally provide a distance field. No matter which approach we choose, the gradient of these distance fields ρ_{Obstacle} can be computed analytically or numerically and plugged into the velocity update formula covered in Section 1.4.

When looking at the Shallow Water Equations (SWE), we find similarities with the equations outlined in this chapter. In fact, by looking at the density field as the height field of a liquid surface, we can imagine using our method directly for height field simulations shown in Fig. 1.4. On the other hand, there is usually a ground height term in the SWE. This in turn can be interpreted in 3D as a distance field for rigid objects as we mentioned above.

1.7 Examples

We show a few simulation examples implemented with HLSL, compiled as level 4 shaders. They include a 3D liquid with rigid objects in the domain, a 2D shallow water height field simulation, and a 2D simulation comparing with the optimized Direct Compute implementation of SPH available in the DirectX 11 SDK. Measures include both simulation and rendering. We splat particles with a radius of 3 cells on 16-bit floating point textures without any significant loss in visual quality. In general we used a grid size close to $\sqrt[d]{N_{\text{P}}}$ texels per axis, where d is the domain dimension (2 or 3) and N_P is the total number of particles.

In Fig. 1.3, we rendered the fluid surface by raycasting the density directly. We perform a fixed number of steps and finish with an FXAA anti-aliasing pass (frame buffer size 1024×768). We used 125k particles on a 96^3 texture and the simulation performs at 35 frames per second (FPS) on a Quadro 2000M graphics card.

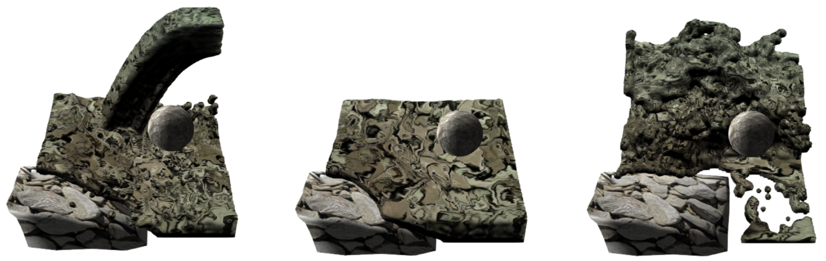


Figure 1.3. A 3D liquid simulation with obstacles in the domain implemented using the rasterization pipeline. The simulation runs at 35 FPS using 125k particles on a Quadro 2000M graphics card.

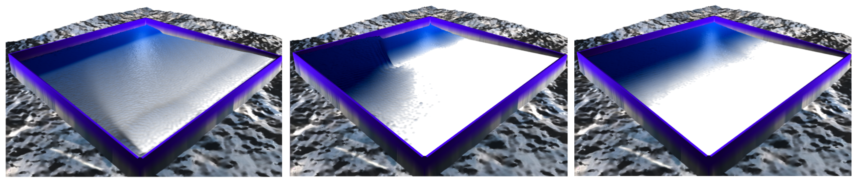
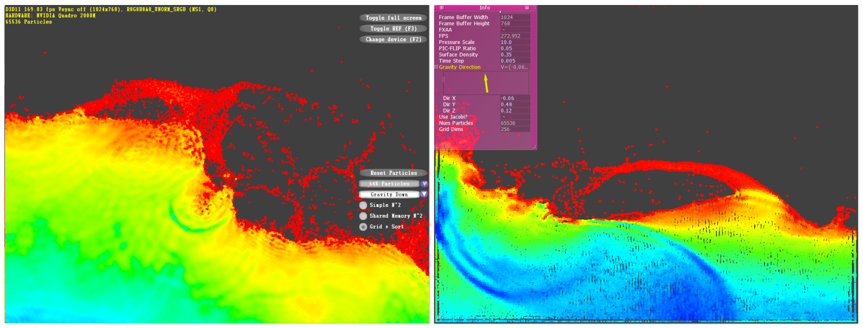


Figure 1.4. The shallow water equations describe the evolution of water height over time. By looking at the height as the density of a 2D fluid, we see the equations becoming similar. Hence, our method can be used directly to simulate height fields. This figure shows a SWE simulation with 42k particles using our method.

In Fig. 1.4, we see a shallow water simulation using 42k particles on a 256^2 grid. The simulation and rendering together run at 130 FPS using the same graphics card.

We compared our solver *qualitatively* with the SPH GPU implementation in the DirectX 11 SDK (Fig. 1.5). Their solver is implemented using the Direct Compute API, has shared memory optimizations and particle neighbor search acceleration. Ours uses HLSL shaders only. In table 1.6, we compare the performance of both methods with different particle quantities. We can see that our method scales better with the number of particles involved for a given grid size and splatting radius on the hardware we used.



```
D3D11 169.03 fps Vsync off (1024x768), R8G8B8A8_UNORM_SRGB (MS1, Q0)
HARDWARE: NVIDIA Quadro 2000M
65536 Particles
```

```
Frame Buffer Width    1024
Frame Buffer Height   768
FXAA                  -
FPS                   272.952
Pressure Scale        10.0
PIC-FLIP Ratio        0.05
Surface Density       0.35
Time Step             0.005
Gravity Direction     V={-0.06..
  Dir X               -0.06
  Dir Y               0.48
  Dir Z               0.12
Use Jacobi?           -
Num Particles         65536
Grid Dims             256
```

Figure 1.5. Comparison between an optimized SPH solver implemented with Compute Shaders on the left and our method implemented with rasterization APIs. Our method performs at 296 FPS using while the optimized SPH solver runs at 169 FPS.

Particle Amount	Grid Size Dim($\phi, \mathbf{u}$)		**Our Method** (FPS)		**DX SDK 11** (FPS)		**Speedup** Ratio
$64,000$	256^2		296		169		1.75
$32,000$	256^2		510		325		1.55
$16,000$	256^2		830		567		1.45

Figure 1.6. Comparison between our method and the optimized SPH solver found in the DirectX 11 SDK.

1.8 Conclusion

In GPU Pro 2, we described the $1 fluid solver: by combining the simplicity of weakly incompressible fluids, with the simplicity of grids, we could simulate a single phase fluid (smoke or fire) in a single pass [Guay et al. 11]. In this chapter, we described the $1 liquid solver for rasterization APIs by combining the simplicity of the particles for dealing with liquids, with the simplicity of the grids to compute the forces. This is useful in getting a liquid solver running quickly on platforms that do not necessarily implement compute APIs.

Bibliography

[Amada et al. 04] T. Amada, M. Imura, Y. Yasumoto, Y. Yamabe, and K. Chihara. "Particle-based Fluid simulation on gpu." In *ACM Workshop on General-Purpose Computing on Graphics Processors*, pp. 228–235. ACM, 2004.

[Bayraktar et al. 08] S. Bayraktar, U. Güdükbay, and B. Özgüç. "GPU-Based Neighbor-Search Algorithm for Particle Simulations." *journal of graphics, gpu, and game tools* 14:1 (2008), 31–42.

[Becker and Teschner 07] M. Becker and M. Teschner. "Weakly compressible SPH for free surface flows." In *Proceedings of Eurographics/ ACM SIGGRAPH Symposium on Computer Animation 2007*, pp. 209–218, 2007.

[Blinn 82] J. F. Blinn. "A generalization of algebraic surface drawing." *ACM Transactions on Graphics (TOG)* 1:3 (1982), 235–256.

[Crane et al. 07] K. Crane, I. Llamas, and S. Tariq. *Real-time simulation and rendering of 3d fluids*, GPU Gems, 3, Chapter 30. Addison Wesley, 2007.

[Desbrun and Cani 96] M. Desbrun and M.P. Cani. "Smoothed particles: A new paradigm for animating highly deformable bodies." In *Proceedings of Eurographics Workshop on Animation and Simulation*, pp. 61–76, 1996.

[Enright et al. 02] D. Enright, S. Marschner, and R. Fedkiw. "Animation and rendering of complex water surfaces." In *Proceedings of the 29th SIGGRAPH*, pp. 736–744, 2002.

[Guay et al. 11] M. Guay, F. Colin, and R. Egli. *Simple and Fast Fluids*, GPU Pro, 2, Chapter VII-3, pp. 433–444. A.K. Peters Ltd, 2011.

[Kolb and Cuntz 05] A. Kolb and N. Cuntz. "Dynamic particle coupling for gpu-based fluid simulation." In *Proceedings of the 18th Symposium on Simulation Technique*, pp. 722–727, 2005.

[Rozen et al. 08] T. Rozen, K. Boryczko, and W. Alda. "GPU Bucket Sort Algorithm with Applications to Nearest-neighbour Search." *In Journal of the 16th Int. Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision* 16:1-3 (2008), 161–167.

[Stam 99] J. Stam. "Stable Fluids." In *Proceedings of the 26th ACM SIGGRAPH conference*, pp. 121–128, 1999.

[Zhu and Bridson 05] Y. Zhu and R. Bridson. "Animating Sand as a Fluid." In *Proceedings of ACM SIGGRAPH 2005*, pp. 965–972, 2005.