Liquid3D

JONAS BERGER, ADRIEN COUTELLE, BARTHÉLEMY PALÉOLOGUE, Institut Polytechnique de Paris, France

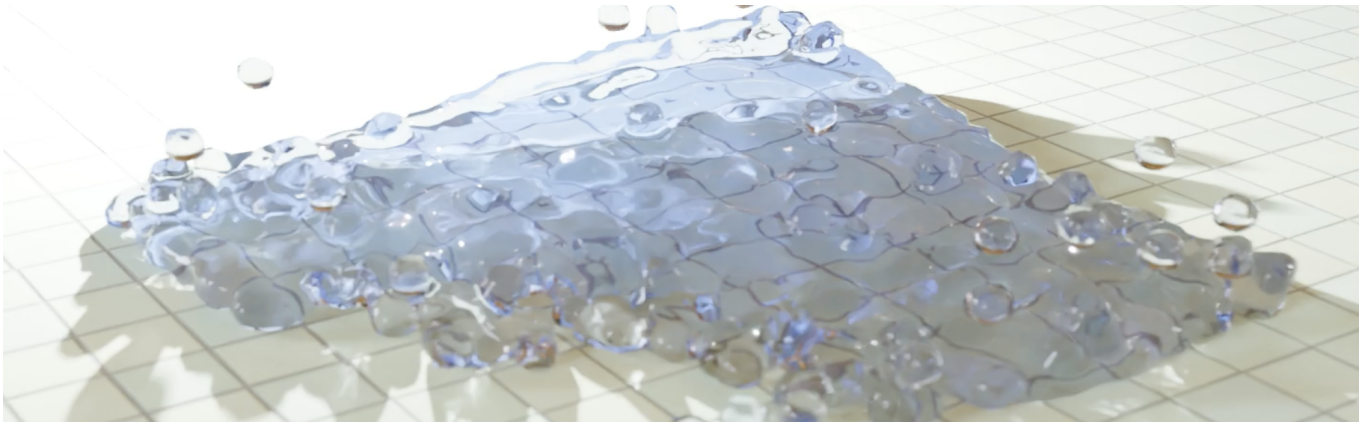

Fig. 1. Caption here fig 1

1 INTRODUCTION

This project aims to create a full pipeline for 3D fluid simulation rendering, which follows those three steps:

- A 3D particle-based fluid simulation which uses the IISPH method described in[1]. The simulation was combined with a spray/foam/bubbles generator inspired by the article [2].
- An OpenVDB converter which takes the particle positions from the simulation and return implicit level-set representations.
- A 3D scene on Blender, which uses a water shader to render a realistic fluid.

In this report, we will detail how each of the steps works, the issue we faced and what might be improved.

2 TECHNICAL DETAILS

2.1 Fluid Simulation

2.1.1 IISPH. The IISPH is a fluid simulation method derived from the flexible SPH method. It reuses the idea of approximating local physical quantities like density and pressure for tiny fluid particles in order to shift them using Newton's second law. The model wants to compute three different forces applied to each particles:

$$\text{Gravity: } \mathbf{F}_i^g = m_i \mathbf{g} \tag{1}$$

$$\text{Viscosity: } \mathbf{F}_i^v = 2\nu m_i \sum_j \frac{m_j}{\rho_j} \mathbf{v}_{ij} \frac{\mathbf{x}_{ij} \cdot \nabla W_{ij}}{\mathbf{x}_{ij} \cdot \mathbf{x}_{ij} + 0.01h^2} \tag{2}$$

$$\text{Pressure force: } \mathbf{F}_i^p = -m_i \sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \nabla W_{ij} \tag{3}$$

Author's address: Jonas Berger, Adrien Coutelle, Barthélemy Paléologue, Institut Polytechnique de Paris, France.

Where h is the particle radius and $W_{ij} = W(\mathbf{x}_j - \mathbf{x}_i)$ is a kernel function that is null from a certain distance. During implementation, we ensure that particles are stocked in a 3D grid of boxes with a unit size equal to the kernel's radius. That way, we can find the neighbors of a given particle just by checking the neighbor of its box, which drastically fasten the computation (we go from a quadratic complexity in the number of particles to a theoretical linear complexity, as boxes have limited capacity).

The specificity of IISPH is that it does not have an explicit expression of pressure. Instead, it constrains the density to $\rho_i(t + \Delta t) = \rho_0$ with ρ_0 the rest density, and by using the continuity equation of Navier-Stokes, ends up with a system of equation with the pressures as solution of it :

$$\forall i, b_i = \sum_j a_{ij} p_j \tag{4}$$

Oh no! This is a quadratic complexity in time AND in storage! We can't compute millions of fluid particles like this, but instead, IISPH use a clever trick called the weighted Jacobi method. Basically, we define a recursive sequence which converges to the vector solution of the SOE. In practice, the convergence needs between 4 and 30 iterations, and we do not need the coefficient a_{ij} but instead the sums $\sum_{j \neq i} a_{ij} p_j^l$, which take a linear storage (and a linear time since the kernel is hidden inside the a_{ij} terms).

The code used is an adaptation of the one from an IGR202 project about IISPH, but with slight modifications. Especially, the whole scene went from 2D to 3D, so the computation takes way more time : for 100k particles, a simulation of 250 frames (a frame equals 5 steps like described previously) took about 3 hours. In two dimension, a simulation with about 2K particles would be equivalent, and it thus goes about 50 times faster. It should be noted that the algorithm is easily made concurrent (one of the main advantages of IISPH), and thus by using the OpenMP API, we can run the simulation on multiple threads (12, in our tests and up to 144 using Telecom Paris' Lames).

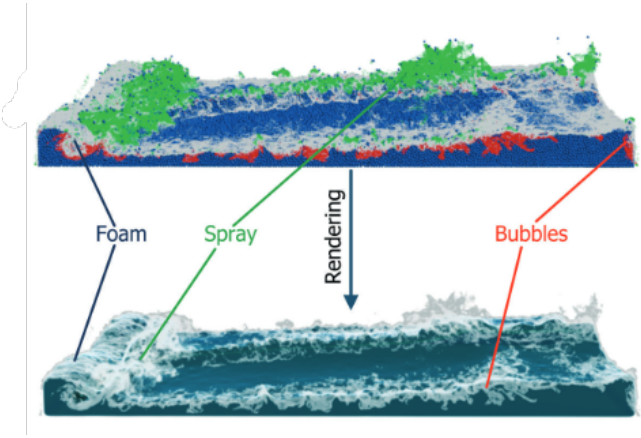

Fig. 2. Spray, foam and bubbles are generated, advected and dissolved on post-processing. (diagram from [2])

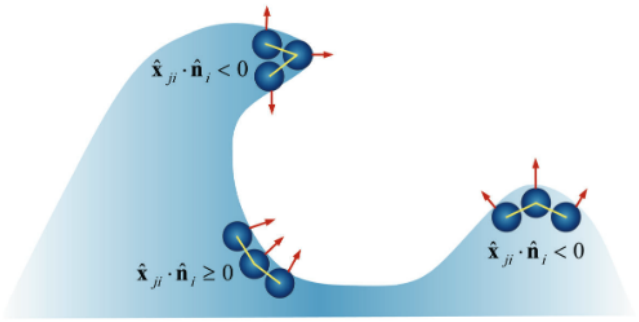

Fig. 3. A point is considered to be on a convex surface if its normal's direction is opposed to the direction of its neighbor. (diagram from [2])

A lot of comfort improvements has also been added, like using flags for defining the size of the simulation or the shape of the initial configuration. The parameters used had to be readjusted for three dimensions. And we also added a security in order to be sure that the Courant-Friedrich-Levy condition is always verified: $\Delta t \leq \lambda \frac{h}{\max_i |\mathbf{v}_i|}$. It consists of lowering the timestep Δt of each step when the velocity is too high (it thus means that we need to compute more steps per frame).

2.1.2 Foam. In order to add another layer of detail, we implemented a foam simulator following the model described in [2]. The algorithm only needs to know the positions and velocities of particles over time, and so can be applied in post-processing, but we made it work simultaneously with the IISPH in order to ease the code and fasten the computations (since a few things like the grid are already computed each frame).

The simulator actually generated three kind of "diffuse particles": spray in the air, foam at the fluid surface, and air bubbles inside the fluid. On each step, a diffuse particle is classified as spray if it has 6 or less fluid particles neighbors, bubble if it has 20 or more, and foam otherwise. We won't detail all the equations in the report, but instead try to grasp the general idea of each phase of the simulation: particle generation, advection, and dissolution.

During the generation phase, a fluid particle will generate a certain amount of diffuse particle according to three different factors. The first one is the kinetic energy $E_i^k = \frac{1}{2} m_i \mathbf{v}_i^2$, as it is needed to create diffuse particle (after all they are a mix of air and liquid). Then, at least one of the two other factors must be non null, as they represent the local context. The factor number two is scaled velocity difference, a value representing a "quantity" of collisions between the fluid particle and its neighbor. When impact occurs, air may be dragged under the surface and trapped.

$$v_i^{diff} = \sum_j ||\mathbf{v}_{ij}||(1 - \hat{\mathbf{v}}_{ij} \cdot \hat{\mathbf{x}}_{ij}) W_{ij} \tag{5}$$

The last factor is the convexity, since foam is often created at the crest of waves (due to strong wind exposure and instability). We thus need to check if the curvature is high and if the particle goes "outside" of the surface. The convexity suppose that we are at the surface of the liquid and that we know the normal of this surface, however the article doesn't explains how to check this condition and to compute the normal (it seems that is considers the liquid as a mesh, but at this point in the pipeline we cannot consider the mesh). So we cooked a solution to those two problems : we compute the normal of a particle according to the relative position of its neighbor, and then we consider the particle to be at the surface (the normal is thus valid) if the normal doesn't point at too much neighbors.

$$\hat{\mathbf{n}}_i = -\frac{\sum_j \mathbf{x}_{ij}}{||\sum_j \mathbf{x}_{ij}||} \tag{6}$$

$$\hat{\mathbf{n}}_i \text{ valid if } \hat{\mathbf{n}}_i \cdot \mathbf{x}_{ij} \geq 0 \text{ for more than 75\% of the neighbors.} \tag{7}$$

$$\tilde{\kappa}_i = \sum_j \kappa_{ij} \text{ with } \kappa_{ij} = \begin{cases} (1 - \hat{\mathbf{n}}_i \cdot \hat{\mathbf{n}}_j) & \text{if } \hat{\mathbf{n}}_i \cdot \hat{\mathbf{x}}_{ij} \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

When a fluid particle generates diffuse particles, those are created with a random position around the fluid and with a corresponding velocity. They are also given a fixed lifetime. Then, during the advection phase, the diffuse particles are moved according to the forces involved: gravity for spray, buoyancy and fluid currents for bubble, and surface movements for foam. Then, during the dissolution time, every foam particle sees its lifetime shortened by the timestep. If the lifetime reaches zero, then the particle is dissolved. Note than bubbles cannot be dissolved since there are under water, and spray is slightly reduced by a fraction of the timestep (because it is more stable than foam).

When the simulation is done, for both fluid and diffuse particles, their positions at each frame are written in a .txt file, with a different file for each type of particle (fluid, spray, foam and bubble). It thus is ready to be processed by the level set generator.

2.2 Processing with OpenVDB

The goal was to use the industry-standard library OpenVDB to create a level-set representation from our particle data. OpenVDB is a very powerful library and not very beginner friendly so we first tried to use their python module to get started.

2.2.1 Limitations of PyOpenVDB. Looking at the documentation, there was not a lot of things the python module could do for us. It

has functions to convert meshes to vdb files and the reverse, but almost nothing on particles and level-sets. Moreover the install process of just this python module was tedious and we ended up using a docker image from Github (that we had to fix) just to make it work.

We tried to make the processing of the particles ourselves using PyOpenVDB voxel grids, and in the end we could generate vdb files, but they were not of a good quality and moreover the process was super slow.

2.2.2 First implementation in C++. To circumvent the issues with PyOpenVDB, we went using the C++ library instead, which was the right choice. First, we tried to export directly the point cloud of particles without doing meshing as a starting point. OpenVDB provides functions to do just that and we were able to create vdb point data grid files. The issue was then Blender not being compatible with VDB point clouds.

Therefore, we focused on making a level set out of the particle data. Understanding how to do it using OpenVDB was hard as the library uses indifferently the terms SDF and levelset which was very confusing at first.

A good starting point was to look at the unit tests of the project to find usage examples. [Unit tests]. From here we created a C++ function that would parse the particle data from our simulations to match the interface given by OpenVDB and create the vdb files out of the data.

As it was not clear how we would use the vdb files as mesh in Blender, we added an option to also export .obj files to get regular meshes in Blender. It proved unecessary in the long run as we could perform everything we wanted with only the vdb files.

As our tutor expressed interest in reusing this part of the project for more research, we made extra care to document the program and make it very flexible for any kind of particle data. Almost every part of the program is customizable using execution flags and the documentation is generated by Doxygen and available online at our repository.

2.2.3 Docker interface. As we wanted to make it python compatible, we created a docker image to run the c++ project. The docker could then be called from python to convert the particle data.

The docker image is available at Docker Hub

While this subpipeline works, it was proven impractical as modifications to the converter necessitated to rebuild the docker image to expose the new features to the python module.

2.2.4 Python implementation. Moreover, the docker image is not maintainable by a 3rd party. Therefore we went back to simple plain python. We re-implemented the lacking logic of PyOpenVDB directly in python with numpy. We adapted the implementation from another repository, and we ended up with a working script that ran 500x slower than the original C++ implementation we made earlier.

2.3 Rendering with Blender

In this project we wanted to render water (or other liquids) from levelset .vdb files. We render using cycle and Blender 3.5.1.

2.3.1 Import .vdb files in Blender.

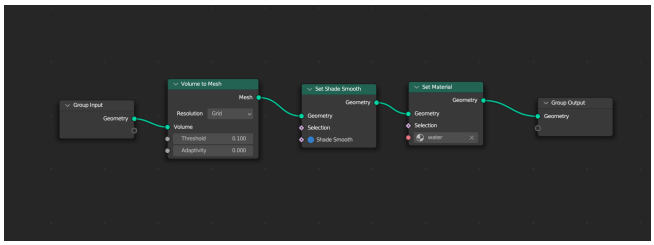

Fig. 4. Geometry Node from volume to mesh

- Create a new **General** file
- **Shift+A**, **Volume → Import OpenVDB...** Go to your folder with all your .vdb files, press **A** to select them all and **Import OpenVDB Volume** to complete the importation.
- You can now see in the Outliner that a Volume has been added. We can see that there is only one file, which contains all the .vdb files. If we press the **Space bar** to start the animation, we can see that the volume is moving.

2.3.2 Geometry Nodes. After the importation process, we have an animation (the amount of frames is equal to the number of .vdb files imported), but it looks like a shadow. To shade it properly, we first want to convert the volume into a mesh. The purpose of this is to have a volume with a surface that we can shade.

We figured out that we could do this conversion using geometry nodes. See **Fig. 4**.

If the file is a levelset, the mesh should appear instead of the point cloud.

- **Volume to Mesh** converts the point cloud into a Mesh (to see surfaces).
- **Set Shade Smooth** makes a smooth normal field on the surface, so we don't see the edges of the mesh.
- **Set Material** select the Shader we want to apply to the mesh. We will create the Shaders afterward.
- **Group Input** and **Group Output** should be linked to the rest to make the Geometry Nodes editor work.

2.3.3 Shader Editor. Once we have the mesh representing the liquid, we want to shade it properly to make it look like water, or pancake batter for example. For the water shader, we started with the transparent element of the aspect of the water with a transparent shader to see through but the result was not convincing, because water difracts and reflects light. We replaced the transparent shader with a glass shader, in which we can setup the roughness to zero and the IOR to 1.333 to represent water. To add a litte more realism, we add a volume absorption shader to make the water darker if the light has traversed a long distance in it. We can see the difference in the **Fig. 5**.

The **Fig. 6** shows the water shader.

We could also render other more or less newtonian liquids, like pancake batter. For this purpose we only shaded the surface, with a pancake-like color, and added a little bit of noise to make it more realistic.

2.3.4 Caustics. At this points the rendering is good, but we could do even better. In fact, in reality when the light hits the water, a part

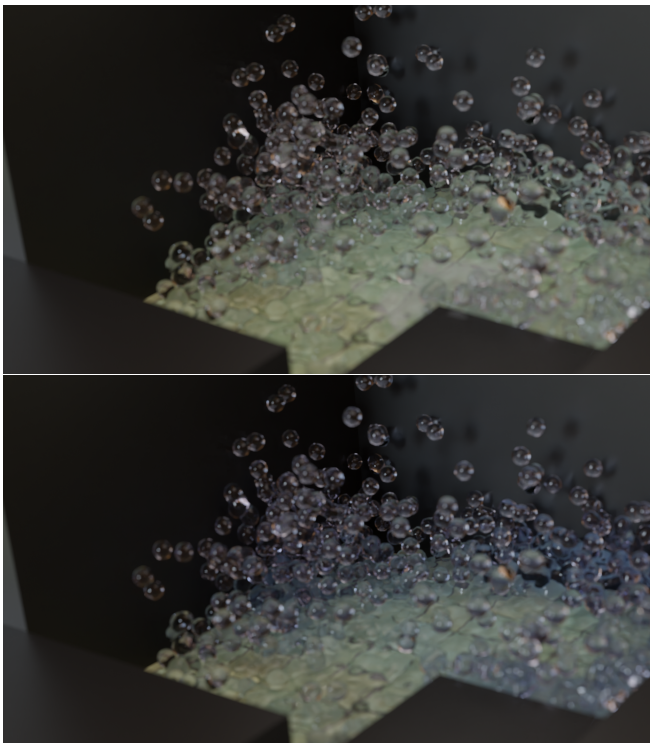

Fig. 5. Without and with volume absorption

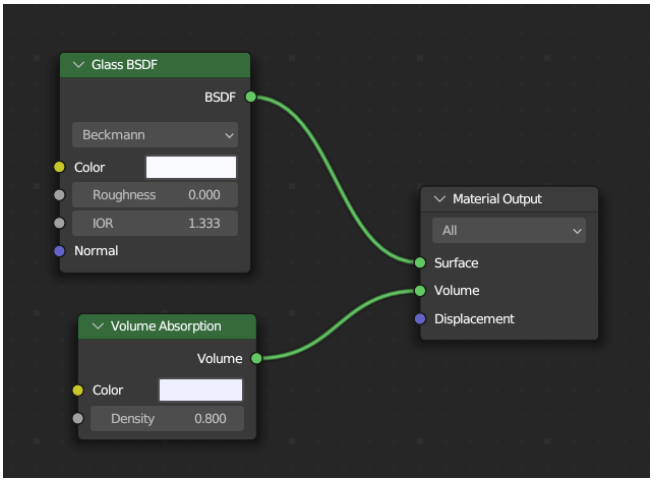

Fig. 6. Water shader

of it is reflected and another is refracted. The purpose of caustics is to render properly the refraction. To setup the caustics, we followed [this] tutorial on YouTube from 4:26, only Blender 3.2 is needed.

To visualize better this effect, we first placed the water volume on a plane, with a low-angle light to see a large projection of the effect on the plane. We got the results on **Fig. 7**.

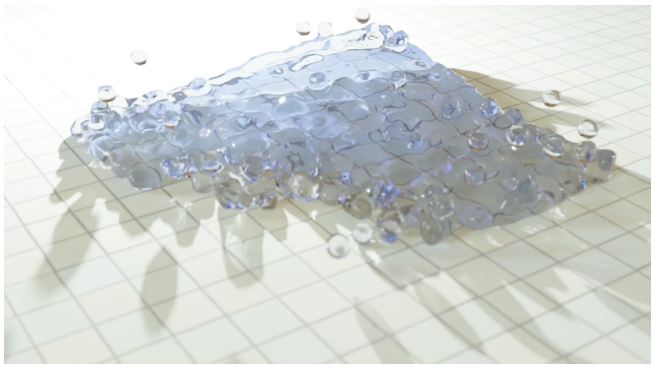

Fig. 7. Caustics on the shadows

3 CONCLUSION & POSSIBLE IMPROVEMENTS

In the end, there was one major issue we discovered too late and thus couldn't fix: it seems that the boundary is not dense enough to physically block the fluid particles when there are too much of them (>150k). As in classic SPH simulation, we use wall particles, which behave almost like a fluid particle except they don't move and have a different rest density. For security, we clamped out of bounds particles. In theory it should not happen, but some particles still manage to go out of bounds, and when they are clamped the "teleportation" creates overpressure in the fluid and the simulation "explodes". We tried different solutions like adding more wall particles, increasing the rest density, or making the wall thicker, but we could not find a convincing solution among those. The paper [3] proposes a complex boundary handling for IISPH, so with more time it would be nice to implement it.

Except this issue, the whole pipeline works quite well and we manage to produce convincing and appealing fluid videos. Here are a few things we could improve:

- For the simulation, in-depth debugging would be necessary to solve the remaining issues. Moreover, we could try to implement a more modern and performant simulation method called DFSPH[4]. It is faster and handles well highly-viscous fluids. Finally, we could make the system a bit more flexible by adding moving boundaries (valves) or handling non-newtonian fluids.
- We could improve surface reconstruction by using anisotropic kernels[5]. This method is more costly to run but produces smoother surfaces.
- We could create more accurate shaders for the water to be more realistic. It would be nice to create more complicated scenes too, but for this purpose it would be necessary to be able to create more advanced bounding boxes (not only rectangles) in C++ to fit a scene we want to create.

REFERENCES

[1] Ihmsen, M., Cornelis, J., Solenthaler, B., Horvath, C. and M. Teschner, 2014, "Implicit incompressible SPH" IEEE Transactions on Visualization and Computer Graphics 20, 426– 435.
[2] Ihmsen, M., Akinci, N., Akinci, G. and Tescher, M. "Unified Spray, Foam and Bubbles for Particle-Based Fluids" In CGI2012 Papers
[3] Band, S., Gisser, C., Ihmsen, M., Cornelis, J., Peer, A. and Teschner, M, 2018, "Pressure Boundaries for Implicit Incompressible SPH" in ACM Transactions on Graphics 37-2, Article 14

[4] Bender, J. and Koschier, D., 2016, "Divergence-Free SPH for Incompressible and Viscous Fluids" IEEE Transactions on Visualization and Computer Graphics 23, 1193 - 1206.

[5] Yu, J. and Turk, G. "Reconstructing Surfaces of Particle-Based Fluids Using Anisotropic Kernels" in ACM SIGGRAPH 2010 Papers