Implementation of a position-based fluid simulator

Pascal Béroule,* Alexandre Fournier Montgieux,† and Clément Galaup‡
IGR204 Télécom Paris
(Dated: July 2, 2021)

This paper relates our approach to implement a position-based fluid solver as described in Macklin Müller 2013 [3]. Our final version managed to feature all the solver's characteristics in a 2D particles simulation including Incomprehensibility, Viscosity, Vorticity and tensile instability

I. SUMMARY

This report will comport a brief explanation of position-based fluids theory and will then cover the followed implementation strategy as well as the observations that led us to revise it and come up with new ideas. The final source code and demo are available on the following link `https://github.com/afm215/igr205`.

II. THEORY

A. General Concept

A fluid solver is an algorithm that mimic the movement of particles to create a fluid-like body. It needs to calculate for each frame the new position of each particle so that the movement of the ensemble is in adherence to the laws of physics. The first and naive approach is to simply implement Navier-Stokes's equations to compute the effect of pressure and density forces and then integrate those forces to find the new positions. This idea is behind the SPH fluid solver [2] and poses a few problems, however a position-based fluid simulator [3] will instead of doing difficult fluid mechanics computations try to find the position of the particle that minimize :

$$C\left(p_i\right) = \frac{\delta_i}{\delta_0} - 1. \, . \tag{1}$$

where $\delta_i = \sum_j m_j W(p_i - p_j, h)$

B. Position Solver

To find for each particle, the position p_i that minimizes (1) we perform a gradient descent algorithm

$$\begin{aligned} C(p_i + \Delta p_i) &\approx C(p_i) + \nabla C^T . \Delta p_i \\ &\approx C(p_i) + \nabla C^T . \nabla C \lambda_i \end{aligned} \tag{2}$$

We then compute λ_i with (2) and (3)

$$\lambda_i = \frac{-C_i\left(p_1, \ldots, p_n\right)}{\sum_k \left|\nabla_{p_k} C_i\right|^2} \tag{3}$$

$$\nabla_{p_k} C = \frac{1}{\delta_0} \begin{cases} \sum_j \nabla W(p_i - p_j, h) & if \quad k = i \\ -\nabla W(p_i - p_j, h) & if \quad k = j \end{cases} \tag{4}$$

To find the updated position of the particle $p_i^{t+1} = p_i + \Delta p_i$ with (4).

$$\Delta p_i = \frac{1}{\delta_0} \sum_j (\lambda_i + \lambda_j) \nabla W(p_i - p_j, h) \tag{5}$$

For best results we repeat this calculation either a fixed time between each frame or until all the constraint functions are low enough.

C. Other Forces

To make the fluid movements look more natural we also have to compute each particle velocity and compute the effect of external, viscosity and vorticity (6) forces (5). So, we update the velocity.

$$v_i \leftarrow v_i + c \sum_j (v_j - v_i) . W(p_i - p_j, h) + \delta t . f_i^{vorticity} + \delta t . f_i^{ext} \tag{6}$$

$$f_i^{vorticity} = \epsilon \left(\frac{\eta}{||\eta||} \times \omega_i\right) \tag{7}$$

where

$$\omega_i = \sum_j (v_j - v_i) \times \nabla_{p_i} W(p_i - p_j, h) \tag{8}$$

* pascal.beroule@telecom-paris.fr
† afournier@telecom-paris.fr
‡ clement.galaup@telecom-paris.fr

III. IMPLEMENTATION

A. code structure

Algorithm 1: Pseudo Code

```
1  Function:
2      applyBodyForce()   // velocity+ = dt * m * ∑ f^ext
3      predictPosition()          // x* = x + velocity * dt
4      applyPhysicalConstraints()
5      buildNeighbour()
6      i=0
7      while i < NBIT do
8          computeDensity()
9          computeλ()
10         computeΔp()
11         updatePrediction()          // x* = x* + Δp
12         applyPhysicalConstraints()
13         i++
14     updateVelocity()            // velocity = (x* − x)/dt
15     computeVorticityField()
16     applyVorticityForces()
17     applyViscousForce()
```

- Lines 4 and 12 : The function applyPhysicalConstraints() was only implemented later in the development. It is meant to prevent particles from getting out of the grid.
- Line 5 : This function implements a grid-based neighbors detection, grouping particles by grid ID to be more easily detected later.

B. First running version

For the first iteration, we used a grid based neighbours detection and a layer of anchored particles as external boundaries.

The initial idea of relying only on the solver and particles wall to keep the particles sealed inside of the grid proved to be inefficient as shown in Fig. 1 and 2 and we had to consider using another method to handle border collisions. This is why the applyPhysicalConstraints() function was added.

IV. HANDLING COLLISIONS

As a solution to avoid crashes, we added a physical constraints function which prevents the particles from leaving the grid. We first tested a naive approach which limits the coordinate values in both axis. When a particle crosses the boundaries on one axis, it is arbitrarily set back at the boundary of that axis.

After using this method to handle borders collisions we noticed the apparition of singular particles that

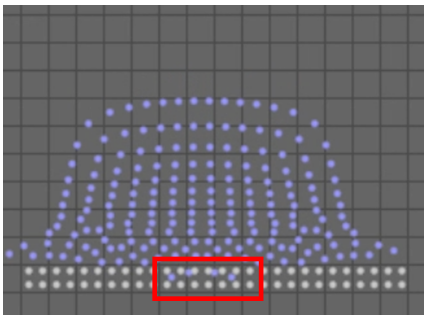

Figure 1. Too much pressure on top of a particle can make it go through the boundary.

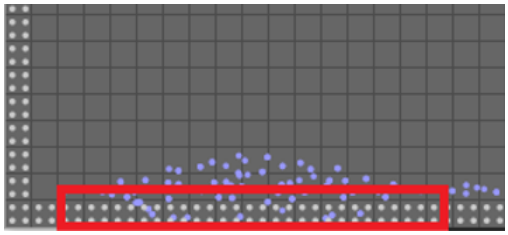

Figure 2. Particle walls are not sufficient to keep particles inside the detection grid.

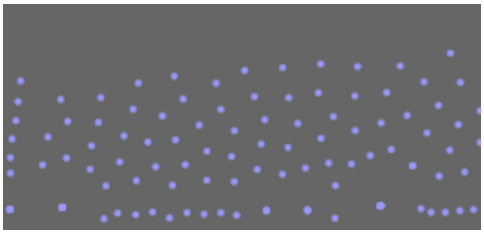

Figure 3. Merged particles causing clusters.

seemed to have a higher density as shown in Fig.3, approximately twice the normal value. Since all particles are supposed to have the same mass, we noticed that this issue was caused by two particles merging when interacting with the collision handler as described in Fig. 4. Having a distance between two particles equal to zero messes up the calculation of the gradient and prevents them from exerting forces on each other and splitting up.

To address this issue, the collision function was changed to introduce an offset when re-positioning particles, that offset depending of the distance crossed out of the boundaries. Adding a degree of variability greatly reduces the odds of two particles colliding and merging together.

The absence of solver based boundaries can lead to unwanted behaviours as shown in Fig.5. Particles tend to accumulate on the bottom and sides of the box and build up density until exploding. Based on those observations, we finally opted for an hybrid solution

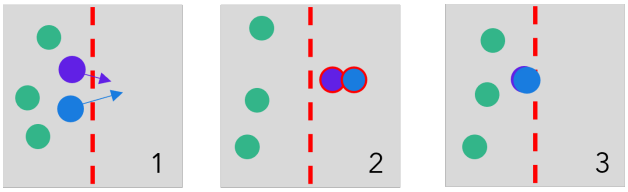

Figure 4. Example of cluster creation caused by hard-coded boundaries.
1. Two particles cross the boundary during the same frame.
2. The two particles are detected as off borders while being on the same y axis.
3. Both particles are placed back on the x axis boundary, they now have the exact same coordinates.

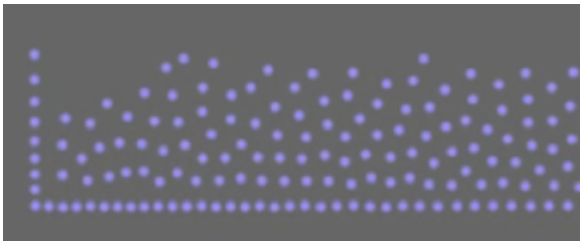

Figure 5. A hard coded collision function induces physically unrealistic behaviours on the edges of the simulation

by keeping the physical constraint function and adding some fixed particles on the sides to prevent further accumulation.

V. VISCOSITY AND VORTICITY

A. Vorticity

Even tough our fluid is now quietly staying in the box, its movement still looked unnatural and some accumulation still occur despite the solution described above.

We thus decided to check the implementation of the fluid property we were the least familiar with, the vorticity. In the article [3] there is a huge confusing step with $\eta = \nabla|\omega|_i$, since we directly associated η with the function $\nabla_{spiky}(\omega_i)$. Yet, as we look closer to it, this interpretation cannot be correct. Indeed, it turns out that $\nabla_{spiky}(\omega_i) \ \|\omega_i$. Thus, as $f^{vorticity} \parallel \omega_i \times \eta$ we were in fact computing a null vector...

The source [1] mentioned in [3] is not helpful since the definition of η is totally different. However with [4] it becomes clear that $\nabla|\omega|_i$ is the gradient of the scalar field $|w_i|$. Still this article doesn't precise how to implement it. The intuition we had was this formula :

$$\nabla|\omega|_i = \sum_{j \in neighbours} (|\omega_j| - |\omega_i|) * \nabla_{spiky}(x_j^* - x_i^*)$$

. This intuition was somehow confirmed by the repository mentioned in the acknowledgements, except some differences we disagree with, the biggest one being not taking into account $|\omega_i|$.

This computation is the cause of the added function computeVorticityField in our algorithm. This function do compute the vorticity vector, but the vorticity field is indeed represented as a scalar array in our implementation. This is explained in equation (8) in which we can see that ω_i is orthogonal to the 2d plane. Hence it only exists with regard to a "virtual" 3d axis and can be thus represented by a scalar. Finally we have $\eta \times \omega_i = (\eta.y * \omega_i, -\eta.x * \omega_i)^T$.

Once the vorticity force is correctly computed, the few remaining accumulation artifact were definitely solved, as , instead of colliding into each other, the particles will rather swirl around each other.

B. Viscosity and explosive behaviour

In the current implementation, the fluid tends to explode when colliding with a surface. We did some research in order to solve this issue and found an interesting statement in Mulller 2003[2] that concerns the viscosity. The viscosity force should normally act like a frictional force. However, according to [2], with the use of a standard kernel, sometimes the effects observed are contradictory. For two particles coming toward each other, the force computed with the classic poly6 kernel can become negative and therefore increase the relative velocity between the particles instead of decreasing it. This artifact is rare but can appear for real time simulations where the number of particle is limited (which was our case before the gpu implemntation) and can lead to some instability. To avoid this effect it's possible to use the laplacian of another kernel $W_{viscosity}$ dedicated to the computation of the viscosity as explained by [2], and [5] as well.

$$W_{viscosity}(r,h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} & if \quad 0 \le r \le h \\ 0 & otherwise \end{cases} \quad (9)$$

However it seems that the mentioned phenomenon doesn't occur in our implementation since we noticed no improvement after implementing this solution. It is quite likely that the issue is due to the physical constraints artificially breaking the incompressibility property of the fluid.

VI. PARALLELIZATION AND GPU IMPLEMENTATION

A. CPU multithreading

In order to drastically increase the performances of our implementation, it was obvious that it was necessary to

parallelize our calculations. This step was in fact straight forward since most of the function written in the pseudo code implement a loop over each single point of the simulation. It is quite clear that each iteration of these loops are entirely independant. One function is still hardly parallelizable, the function buildNeighbour.

From a performance point of view, computing neighbors through a grid is most likely the most efficient way. However, this solution comes with a major drawback. Because each a cell of the grid can contain a various number of particles, the three dimensions (four when you consider a 3d space for the particles) of the Neighbour grid are not well defined (especially the last dimension). That is why, we use a dynamic array in order to represent it. However the operation of adding values in the grid makes the whole buildingneighbour function not easily parallelizable. Still, by using CPU multithreading, we multiply the computation speed by ten.

In order to increase the performances even more, we decide to use multithreading on GPU. Thus, in our final version, the line seven to seventeen in the pseudo code are executed directly on GPU.

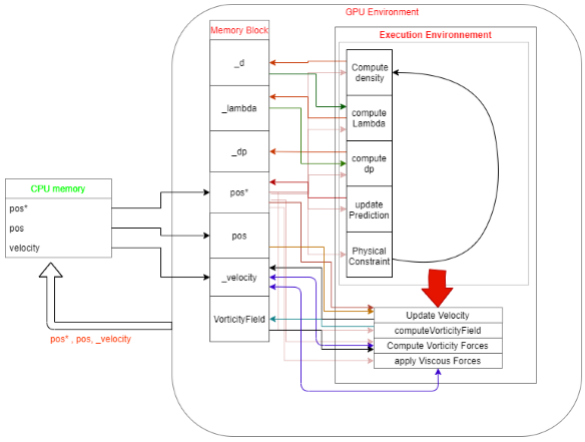

Figure 6. The input/output relation between memory block and GPU execution steps

B. GPU implementation

We implemented the line seven to seventeen by using opencl. The reason behind the fact that we didn't choose to execute all the parallelizable code on a GPU was to limit the data exchange between the GPU memory and the CPU memory, and by extension to avoid the CPU/GPU bottleneck. As you can see in Figure 6, the GPU can thus execute quite independently a huge bunch of code.

As we explained above, a Neighbours grid is an optimized way to deal with the neighbours research. However there is a huge problem with the opencl implementation. Opencl only accepts 1D array while the neighbour grid is not only a 3D array but also an array with a non defined dimension. That is why, flattening the grid is not straight forward. The idea we come with is to roughly flatten the neighbour grid so that the content of a cell comes one after the other. We create a new array called indexes that stores each cell's ending index, so that we now can access to the content of a targeted cell. This solution made our cpu code compatible for the most part with the GPU implementation.

To conclude, the GPU implementation can look eventually heavy. This can be explain by the Figure 6: As you can see the dependencies between each steps of the GPU code are highly intricate which requires a sequential implementation. Each line of the pseudo code must be executed only once the previous line is finished. That is why you can't execute the line 7 to 17 in only one opencl kernel, but you have to encode one kernel for each line, which makes our final implementation quite heavy.

VII. DISCUSSION AND CONCLUSION

Our final implementation supports CPU and GPU parallelization, showing a pretty good performance. The rendered result is pretty smooth with a low amount of particles but sadly, increasing the number of particles is not only a question of performance and we started to face unexpected particle behaviors with more than 5 000 particles on screen.

The main setbacks we faced are for the most part linked to miss-understandings or miss-interpretations of the reference article[3]. Most of our time was devoted to debugging and figuring out empirical values eluded in the reference material.

ACKNOWLEDGMENTS

Huge thanks to Kiwon Um for his precious advises and comments as well as his patience with us during this project. We also have to give credit to k-ye/PbfVs which provided us with much needed help when balancing the parameters of the simulation.

[1] Jong Chul Yoon Chang Hun Kim Jeong Mo Hong, Ho Young Lee. Bubbles alive. *ACM SIGGRAPH 2008*, 2008.
[2] David Charypar Matthias Müller and Markus Gross. Particle-based fluid simulation for interactive applications. *The Eurographics Association 2003*, 2003.
[3] Miles Macklin Matthias Müller. Position Based Fluids. *ACM Transactions on Graphics*, 32, 2013.
[4] NVidia. Gpu gems. 38(5.1), 2007.

[5] David Illes Peter Horvath. Sph-based fluid simulation for special effects. *CESCG 2007*, 2007.