INF574 Project - Fluid simulation

Léo Heidelberger & Pascal (Chien-Hsi) Chang

December 13, 2019

1 Introduction

Being able to simulate fluids in computer graphics is essential for realistic rendering of water, smoke or any kind of gases. For this project, we decided to look into the classic paper of Jos Stam on Stable Fluids (1999) and to implement its algorithm on our own. We successfully reproduced the paper's method in 2D and 3D using Python and Tensorflow, with almost real-time performances for reasonable grid sizes.

2 The theory

As explained in the article, fluid dynamics are governed by the famous Navier-Stokes equations:

$$\begin{cases} \nabla \cdot \mathbf{u} = 0 \\ \dfrac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \dfrac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f} \end{cases}$$

where $\mathbf{u}$ is a vector field that represents the velocity and p is the pressure scalar field. The first equation means that the fluid conserves the mass while the second one comes from the fact that it conserves the momentum. The article uses the Helmholtz decomposition to assemble these two equations into one unique equation on the velocity field:

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{P}\left(-(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f}\right)$$

where $-(\mathbf{u} \cdot \nabla)\mathbf{u}$ is the advection term, $\nu \nabla^2 \mathbf{u}$ is the diffusion term and $\mathbf{f}$ is the force field applied to the fluid.

A grid is set to discretize space into cells, and velocity is defined at the center of each cell. The insight of Stam's paper is that it introduces an operator-splitting framework for solving the equation; it thus boils down to the computation of four fairly simple steps at each iteration. We will present in the next section how we implemented these steps in 2D and 3D.

3 Implementation

We decided to implement our project using Python 3 and Tensorflow 2 because of two reasons. On one hand, Python is an interpreted high-level programming language, so it allows quick coding without compilation. On the other hand, Tensorflow 2, a library mainly used for machine learning purposes, offers a wide range of methods for manipulating multidimensional arrays, which is relevant for this project. Also, using operations like convolutions and `take` instead of `for` loops makes our implementation compatible with GPU.

3.1 Operators

Our first task was to implement the various operators that we will be using in the partial differential equations: gradient, divergence, laplacian and vector laplacian. In 2D, these operators can be seen as convolutions between the main matrix (vector or scalar field) and a 3×3 or 5×5 kernel. Each of these kernels are combinations of first and second derivative kernels:

$$\mathcal{K}_{\partial x} = \frac{1}{2}\begin{pmatrix} 0 & 0 & 0 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \qquad \mathcal{K}_{\partial y} = \frac{1}{2}\begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \qquad \mathcal{K}_{lap} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Let $\mathbf{U}(i,j)$ be a 2-dimensional vector field of size $m \times n$ where each coordinate $\mathrm{U}_x(i,j)$ and $\mathrm{U}_y(i,j)$ is a scalar field. Let $\mathrm{S}(i,j)$ be a scalar field of size $m \times n$ too. Then we implemented the operators as following:

- $\nabla \mathrm{S} = \begin{pmatrix} \mathcal{K}_{\partial x} * S \\ \mathcal{K}_{\partial y} * S \end{pmatrix}$
- $\nabla \cdot \mathbf{U} = \mathcal{K}_{\partial x} * \mathrm{U}_x + \mathcal{K}_{\partial y} * \mathrm{U}_y$
- $\Delta \mathbf{U} = \begin{pmatrix} \mathcal{K}_{lap} * \mathrm{U}_x \\ \mathcal{K}_{lap} * \mathrm{U}_y \end{pmatrix}$
- $\Delta \mathrm{S} = \nabla \cdot \nabla \mathrm{S} = \mathcal{K}_{\partial x} * (\mathcal{K}_{\partial x} * S) + \mathcal{K}_{\partial y} * (\mathcal{K}_{\partial y} * S)$

where $*$ is the convolution operation. Note that the laplacian and the vector laplacian are not computed with the same kernels: the first one is defined as a composition of the gradient and the divergence operators while the latter uses the 3×3 laplacian kernel defined earlier. This is closely related to the nature of the equations we will be solving with these two operators in the third and fourth steps.

3.2 First step: adding the force

The first step of each iteration consists of adding the external force field $\mathbf{f}$ to the velocity field of the previous step:

$$\mathbf{w_1} = \mathbf{w_0} + \Delta t \cdot \mathbf{f}$$

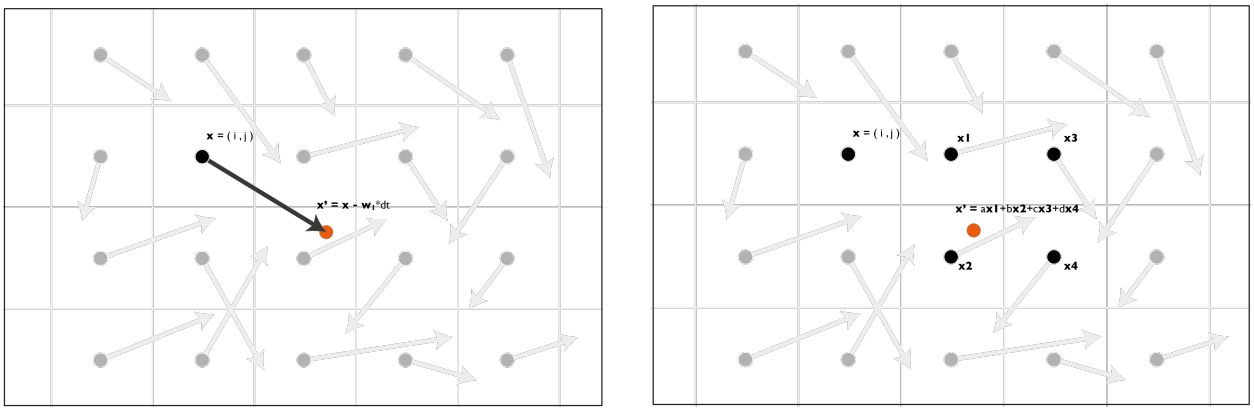
The force field cannot be constant, otherwise the velocity field would explode over time. In our demo, we define the force as being zero in the whole grid except in a tube in the middle. In this vertical tube, the force equals $1 - \mathbf{w_0}$. This will tend to keep the velocity near the value 1, which is relevant as we do not want particles in the grid to be able to cross more than one cell in a single time step.

3.3 Second step: advection

As explained in the article, the particles are moved by the velocity field during each time step. Therefore, to obtain the velocity at a point $\mathbf{x}$ at the new time, we only need to backtrace the point $\mathbf{x}$ through the velocity field $\mathbf{w_1}$ over a time Δt.

For each center of cell at position $\mathbf{x}$, we subtract the distance $\mathbf{w_1}\Delta t$ to it, giving us a new position on the grid: $\mathbf{x}' = \mathbf{x} - \mathbf{w_1}\Delta t$. Omitting border conditions for the moment, this position lies necessarily within four cell centers whose positions are denoted by $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ and $\mathbf{x}_4$ (see Figure 1). It can therefore by written as a convex combination of these four positions: $\mathbf{x}' = a\mathbf{x}_1 + b\mathbf{x}_2 + c\mathbf{x}_3 + d\mathbf{x}_4$, where $a + b + c + d = 1$. The new velocity $\mathbf{w_2}(\mathbf{x})$ at $\mathbf{x}$ that we are

looking for can thus be written as: $\mathbf{w_2}(\mathbf{x}) = a\mathbf{w_1}(\mathbf{x}_1) + b\mathbf{w_1}(\mathbf{x}_2) + c\mathbf{w_1}(\mathbf{x}_3) + d\mathbf{w_1}(\mathbf{x}_4)$



(a) (b)

Figure 1. (a) Tracing back the particle (b) Compute new interpolated velocity

In the case $\mathbf{x}'$ lies outside the grid, we clip its position values with the size of the grid to keep it within it. This solves the advection part of the equation.

3.4 Third step: diffusion

The third step is about solving the diffusion component of the equation, it boils down to solving this linear system:

$$\left(\mathbf{I} - \nu\Delta t\nabla^2\right)\mathbf{w}_3(\mathbf{x}) = \mathbf{w}_2(\mathbf{x})$$

with $\mathbf{w_3}$ being the unknown variable.

Since we are not explicitly defining the matrix of the linear system (because we use convolution to compute the laplacian), we first tried traditional gradient descent to solve this equation. The error was not very high yet the compute time was relatively long, which did not allow real-time rendering of the fluid. We then tried using the conjugate gradient as an iterative method. This approach gave us faster results.

We initialize our unknown with the previous velocity field $\mathbf{w_2}$ and update the various variables according to the method. While the `for` loop stops after 1000 iterations in theory, the algorithm generally breaks out of the loop after 100 iterations because the approximation is already good enough.

3.5 Fourth step: projection

This step ensures that the new resulting field is divergence free. Basically, we need to solve $\nabla^2 q = \nabla \cdot \mathbf{w}_3$ for q. The new field $\mathbf{w}_4 = \mathbf{w}_3 - \nabla q$ is then divergence free. Similarly, we use the conjugate gradient descent to solve this equation.

Note that the whole algorithm is indeed unconditionally stable for any time step since the velocity field computed at each step is a convex linear combination of the previous velocity field.

Furthermore, the fixed boundary condition is valid because convolutions on border cells implicitly assume that the velocity of all the ghost cells outside our grid equals 0.

4 Specificity of the 3D implementation

4.1 Convolution efficiency

In the 2D implementation, we represented most of our vector operations as simple 3×3 convolutions. Our initial idea for the 3D implementation was to use $3 \times 3 \times 3$ convolutions. However, doing so would mean that the convolution kernels would have been very sparse. Unfortunately, our library did not implement sparse convolutions. To improve the efficiency we use a combination of not sparse $1 \times 1 \times 3$ kernels.

4.2 3D renderer

To view our results we needed a 3D renderer, we had the following possibilities:

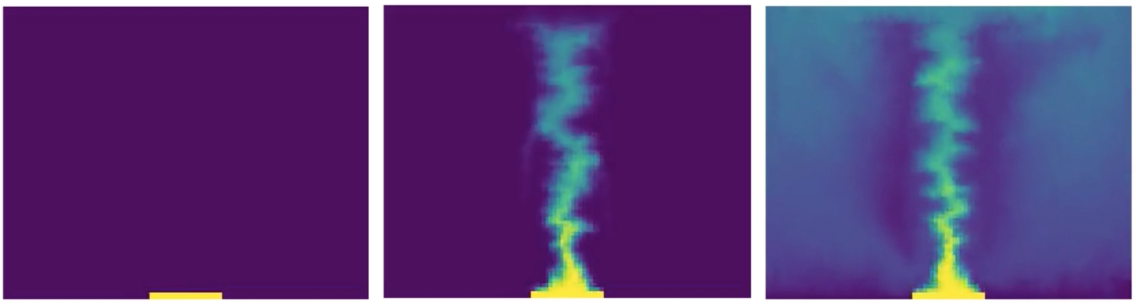
- Simply summing along an axis and displaying it as a 2D picture.
- Implementing a simple ray tracing algorithm.
- Using an external library.

We chose the second alternative as it is an interesting challenge and gives nice results.

5 Results

5.1 In 2 dimensions

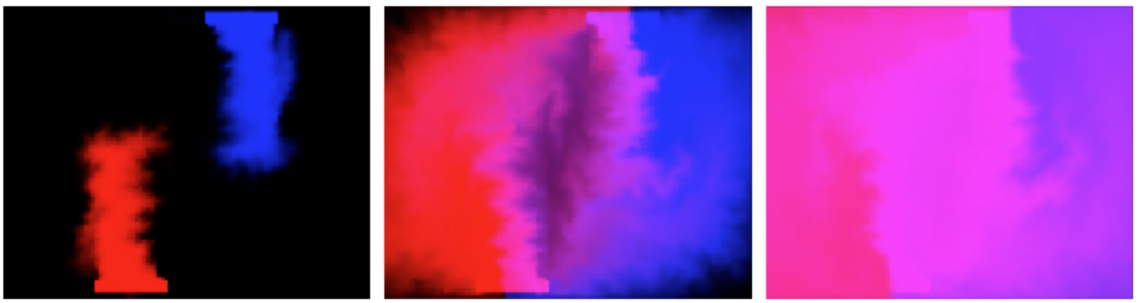
Our 2D implementation is able to compute fluid simulations in realtime with a grid of size 80×100. The results look like this:



(a) (b) (c)

Figure 2. 2D implementation on a 80×100 grid

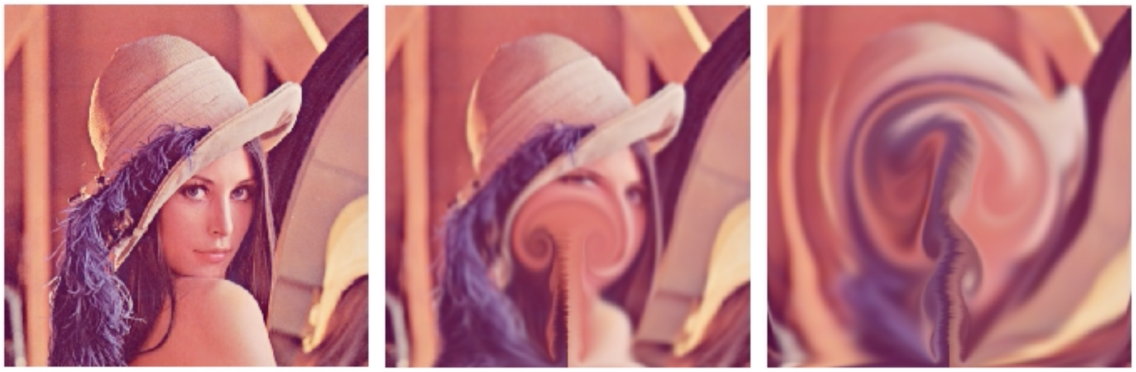
The same implementation can be used with different colors of "ink" and will blend them naturally. The results are shown in Figure 3.



(a) (b) (c)

Figure 3. Color blending with two types of ink

Instead of transporting ink, one could also use it to advect texture coordinates. In the example below, we use it to deform the famous image of Lena Söderberg, widely used in image processing.

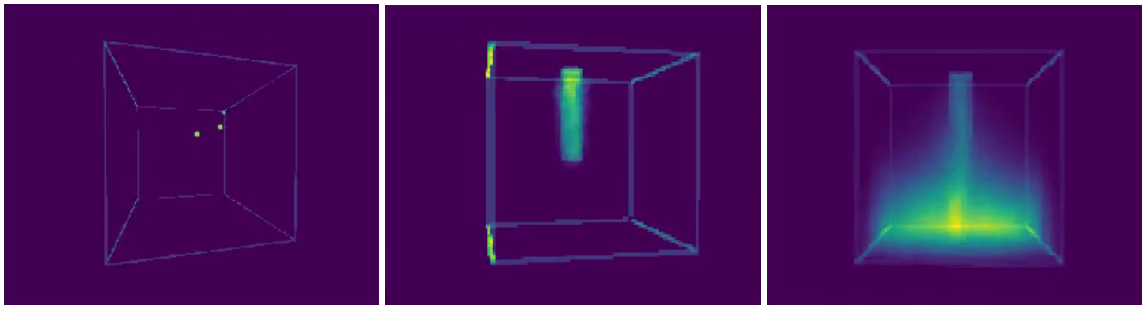


(a) (b) (c)

Figure 4. Advection of texture coordinates on an 512×512 image

5.2 In 3 dimensions

We have the following results in 3 dimensions on a $40 \times 41 \times 42$ grid. The rate is 9 FPS, including rendering.



(a) (b) (c)

Figure 5. (a) Demo of the renderer with 3 spheres (b) Start of the simulation (c) After some time

5.3 Performances

We observe the following performances:

Grid size	Frame rate (FPS)
100×110	41
200×220	37
400×440	19
1000×1100	4

When increasing the dimensions, we don't see a huge drop in the frame rate, this is due to the following three reasons:

- The conjugate gradient converges in less steps.
- A lot of processing time is lost in the setup of the GPU operations.
- The matplotlib rendering takes a constant time.

6 Conclusion

We have shown with our implementation that the operator-splitting scheme presented by Jos Stam in his paper is indeed able to simulate visually plausible effects of fluid dynamics. While switching from 2D to 3D in terms of core calculations was not difficult, it was complicated to find a way to display the fluid's movement in 3D. That is why we additionally coded a simple ray tracing algorithm to show our results.

Yet our implementation still has room for improvement. For one thing, we did not used the midpoint method (second order Runge-Kutta) presented in the paper for the advection step: our algorithm directly traces the particle back to its previous position using the current velocity field. This makes it only accurate to the first order, and we lose small-scale details in the fluid's movement. Also, we only implemented the no-slip border conditions (the fluid cannot get out of the box). The periodic conditions use Fourier transforms to facilitate the calculations of the steps, but we did not have time to try it ourselves.

Regarding the choice of the programming language, Python undeniably gave us an easy way to manipulate multidimensional matrices, which allowed us to focus on the algorithm rather than the technical problems that could arise from the debugging process. However, OpenGL would have been a good alternative too: while the syntax is more complex, it would have given us high performances easily with the paralleled computations done on GPU.