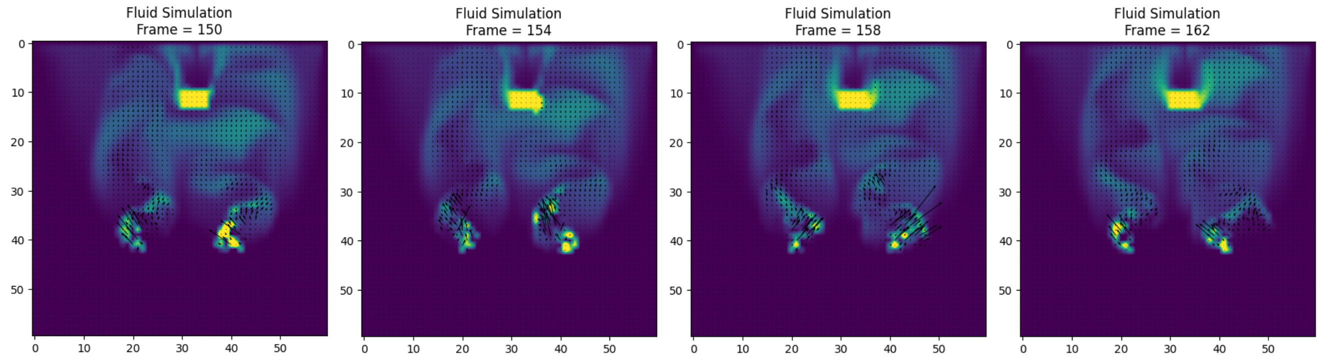
Fluid Simulation: An Extended Solver

Mariana Ávalos Arce
Universidad Panamericana
0197495@up.edu.mx



Introduction

Based on the legendary publication of *Real-Time Fluid Dynamics For Games* by Jos Stam[2], the following document presents an extended version of Stam's solver, meaning that the fluid simulation based on Navier-Stokes Equations and a density and velocity solver is used as Stam once suggested in his pioneering paper, but also some extensions are offered to the reader of this document. These extensions include the capability of the solver to simulate the presence of solid objects inside the fluid and different behaviours of the velocity/density emitters based on classic mathematical models.

1 Input File Structure

In order to add all the features (objects, sources, color scheme, velocity behaviours) I decide to use a JSON file as input for the program. The `config.JSON` file must be provided as a command line parameter when running the program. I decided to use JSON format because it comes very handy when indicating many parameters and user inputs. The general structure of the file is the following:

- `color`: string that contains any of the available options in colors.py.
- `frames`: integer that determines the frame duration of the video.
- `sources`: an array of dictionaries. Each dictionary in the array represents an **emitter**, which is a source of density and velocity. You cannot add an emitter of density and another for velocity in a separate way, because personally I thought it did not made much sense, since the Navier-Stokes equations state that density moves with the velocity. In reality, if you have an emitter of velocity, the density must also be there, and vice-versa. An emitter therefore has the parameters: `position`, `size`, `density` and `velocity`. The `position` parameter sets the density source position in the grid, and the velocity source position alongside it takes the same position. In other words, when you add an emitter in the solver, you are adding a source of density and velocity in the grid.

- `objects`: an array of dictionaries. Each dictionary in the array represents an object with the parameters `position`, `size` and `density`. The `size` parameter contains height and width integers, which will be the shape of a $height \times width$ rectangular object. Thus, objects are rectangles, giving the user the freedom to construct interesting scenarios with walls or shapes based on rectangles, as we will see in the sample images.

2 Objects In The Fluid

An important extension to the existing solver is the presence of arbitrary objects inside the fluid. These are assumed to be solid and heavy enough to not be carried along the fluid, but the latter should interact with the *walls* of each object.

In order to solve this task, I began by defining how the liquid will interact whenever it collides with the walls of an object. The way any force, in this case the **velocity** of the water stream, should *bounce* against the walls of the object needs to be just like a ball would bounce off a wall when thrown towards it.

I start by identifying the incoming force of velocity at any point in the fluid grid. For that, the solver has a function `set_boundaries(table)`, which is called several times throughout the existing solver in order to handle the bounding box of the fluid. We will add some code in this function, because having objects in the fluid means to handle objects as **internal boundaries**. The function receives *table*, which is a grid that contains the velocity bi-dimensional array, where each cell is a 2D vector (array of size 2) as well.

After the existing manipulation of the bounding box bounds, I added a for loop where I cover each object in the JSON config file. In each iteration of an object k that has top-left position as *pos* cell coordinates, we loop through the object's height and width, calling them i and j respectively in the loops, but we will start these loops in -1 and finish them in len(obj) + 1, for height (i's), and len(obj[0]) + 1, for width (j's). In this way, the object will be looped with an extra layer for its boundary cells. Next, whenever we are either on the top, bottom, left or right boundary, which we can know with some if conditions, we will gather the top, bottom, left or right neighbour of the current cell $table[pos[0] + i, pos[1] + j]$ by adding or substracting 1 depending on which object wall we are checking. This neighbour cell value is the **incoming velocity** that collides with the wall of the object, which we will call v.

Now with v at hand, we will perform a simple **vector reflection**, that reminded me a lot of the process I've used in **raytracing** to reflect objects[1]. The input of our function $reflect()$ will be v and N, where N is the normal vector of the wall we are checking: for the top wall, it would be $\langle 0, 1\rangle$; for the bottom, $\langle 0, -1\rangle$; for the left wall, $\langle -1, 0\rangle$; and for the right wall, $\langle 1, 0\rangle$. Inside the function we will need to compute the incoming force angle θ, which can be calculated with a **dot product**,

$$\theta = \arccos(v \cdot N), \tag{1}$$

where v and N are *normalized* vectors. By having the angle with respect to N, we can define the **reflected vector**, which we will call v', using trigonometric relationships shown best through the diagram in Figure 2.

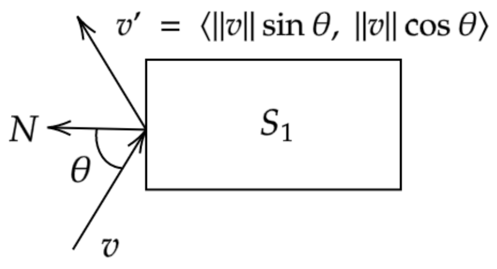


Fig 2. Incoming Force Reflection

Once we have v' vector as the return value of $reflect()$, we can set this vector as our current cell vector in $table[pos[0] + i, pos[1] + j]$, so that we now have the reflected or colliding force direction updated as our new velocity. Then, we simply need to set the density as the average of the values in he current cell $table[pos[0] + i, pos[1] + j]$ in its x and y and we will have the simulation of objects in the fluid. A resulting sequence in the simulation is shown best in Figure 1, where the big wave coming in the first image, collides with the yellow box object, resulting in the breaking of this wave and some accumulation of density in the sides of the box, but then resuming its wavy movement past the box, shown in the last capture of the sequence. Thus, the placement of an object means that there is no interaction with the fluid and the grid slots that are part of the object, but also means the incoming forces must be reflected in order to make the illusion of a force **bouncing** against the walls of the object. This was the most challenging part but also the one I enjoyed most.

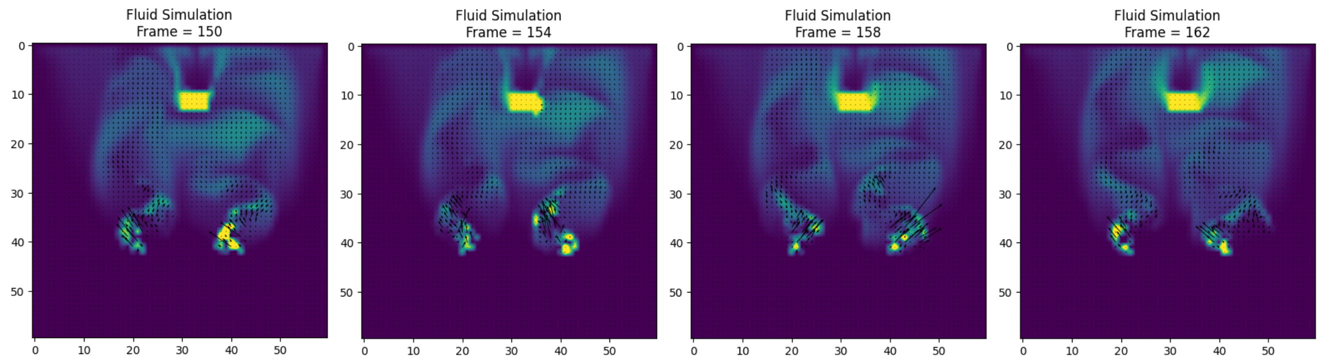


Figure 1: Objects In The Fluid and The Zigzag Behaviour

3 Velocity Behaviours

The next extension I tackled was to define **velocity behaviours** for the emitter's `velocity` attribute. In the `config.JSON` file, the `velocity` key in an emitter's dictionary contains also a `behaviour` key, which needs to be a string contained in behaviours.py. I added 6 behaviours in total, which are: zigzag vertical and horizontal, vortex, noise, fourier and motor. The **zigzag** ones where implemented with a $sin(\theta)$ function, where θ was the current frame, converted to radians and using the `factor` key (also inside `velocity` dictionary) as the frequency constant. Then, the x and y values of the velocity are the amplitude of the function in those axes. These behaviours are shown also in Figure 1, where the left one has less x-axis amplitude than the right one. The **vortex** came naturally after adding a $cos(\theta)$ to the x axis of the velocity vector and a $sin(\theta)$ function to the y axis, in order to form a circle as time moves on. The `factor` key represents the frequency in this case, resulting in a behaviour much like the one shown in Figure 2, where a vortex velocity behaviour in an emitter is interacting with a box placed around (10, 30) in the grid.

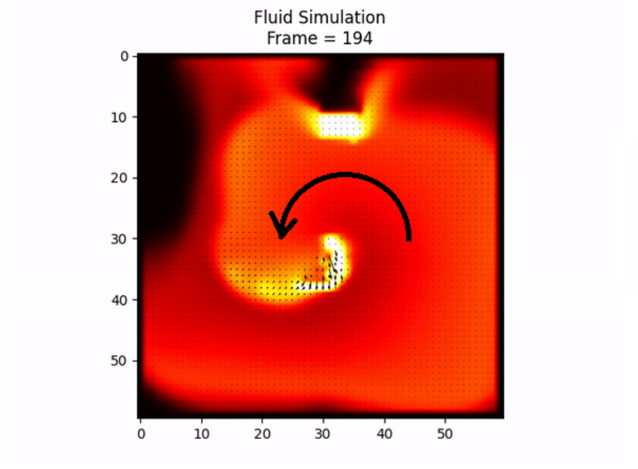


Figure 2: The Vortex Behaviour

The **noise** option consists of a random-changing θ angle. I generated a random number that could be either 0 or 1 and this defined if some small $\Delta\theta$ was either substracted or added to the current angle value. Then, with this angle and the x and y values of the velocity vector given, I used simple trigonometric functions to define the new x and y vector values of the velocity arrow tilted by this θ angle. The result is shown in Figure 3, where the fluid color shows that the angle makes the velocity rotate forward and backward by this small $\Delta\theta$.

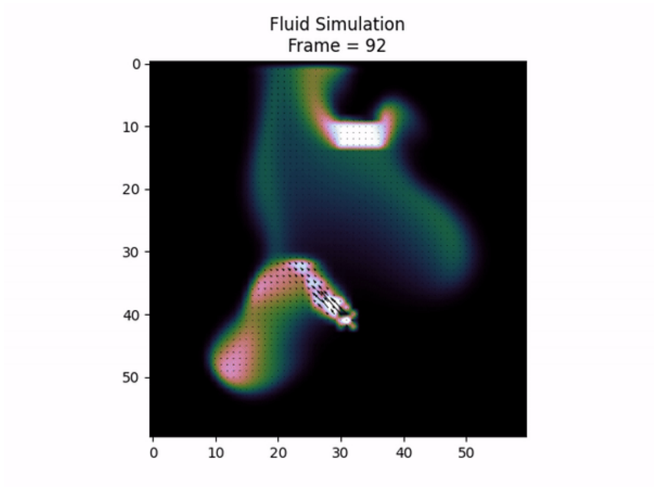


Figure 3: The Noise Behaviour

The **fourier** option was mainly an experiment I wanted to try. I used specifically the Fourier Series Squared Wave function, which is basically a sum of $sin(\theta)$ functions, that with each iteration gets smaller in amplitude but larger in frequency. In this case, the `factor` key represented the number of iterations the Fourier sum will have. The model for the i^{th} frame I used was the following [3],

$$f(i) = \frac{4}{\pi} \sum_{n=1}^{factor} \frac{1}{i} \sin \frac{n\pi i}{L} \tag{2}$$

And it ended up looking pretty similar to a zigzag behaviour, only a little bit more rough when observed closely and when iterations are low. Finally, the **motor** behaviour was also added as an option. This one used a function that I found on the web (unknown source) few months back and it looked pretty interesting,

$$\begin{aligned} x &= -\sin 7.3i \cos\cos 2.4i \\ y &= \cos^4 7.3i \sin\sin 2.4i \end{aligned} \tag{3}$$

Where i represents the i^{th} frame. The result of this function is a small periodic explosion that changes sides and almost looks like a motor of an old car, because it looks like it is gathering force and then explodes, and so on, changing sides as time passes, as the arrows pointing shown in Figure 4.

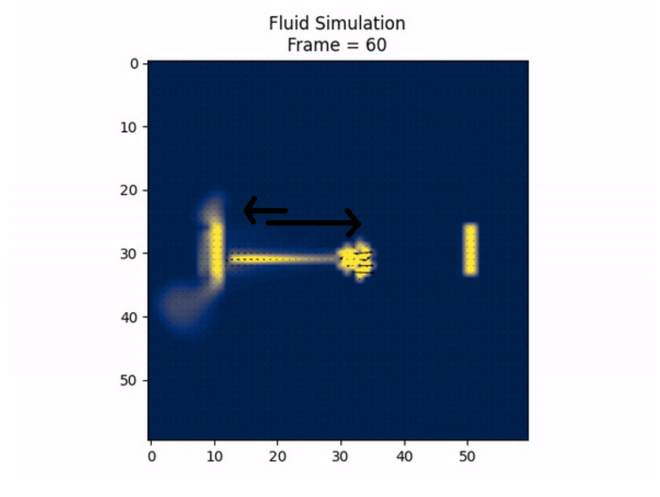


Figure 4: The Motor Behaviour

4 Color Schemes

The last part of the extensions was to use different color schemes for the animation frames, and I implemented this functionality as a parameter that must be added in the `color` key inside the `config.JSON` file. The options available are the ones I included in colors.py file, which are a selection of color maps from the Matplotlib library, so that the user finds them quickly without having to look for the color map options for Python's Matplotlib documentation.

Conclusions

The biggest challenge in this project was to implement the object placement inside the fluid, because it implied an understanding of the velocity matrix in order to reflect the vectors and also in order to ignore the fluid when the grid cell was part of an object. Then, the velocity behaviours were also tricky, but mainly because I wanted to experiment with mathematic functions I had seen. The emitter placement was also a big challenge because I had to understand the solver structure and summarize the information needed for the input file. Overall, it was a very interesting project to work on, and I hope all the challenges where solved in the most understandable and logical way.

References

[1] scratchapixel.com. *Reflection, Refraction (Transmission) and Fresnel.* URL: `https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-shading/reflection-refraction-fresnel`.

[2] J. Stam. "Real-Time Fluid Dynamics for Games". In: *Game Developer Conference* (2003). DOI: `https://www.dgp.toronto.edu/public_user/stam/reality/index.html`.

[3] E. Weisstein. *Fourier Series–Square Wave.* URL: `https://mathworld.wolfram.com/FourierSeriesSquareWave html`.