Computer Animation - 3D Fountain using SPH

Elise Bonnail
elise.bonnail@telecom-paris.fr
Soline Hayes
soline.hayes@telecom-paris.fr

March 2021

Contents

Figure 1: Fountain to create

1 Introduction

This project is part of the "INF585 - Computer Animation" course, that we took for our "Interaction, Graphics and Design" master at IP-Paris. The goal of this project was to implement something related to this course and computer animation. We both especially enjoyed the fluid simulation part of the course, and we wanted to use it in this project. We decided to simulate a fountain (a projection of water from a fixed point). The objective was to have both a correct physical simulation (the water particles should behave according to physics laws), and a satisfying visual result, which corresponds to the mental image we have of a fountain (Fig. 1). For the implementation of this project as for the course labs, we used C ++ and OpenGL (with the VCL library).

2 Implementation

2.1 Smoothed Particle Hydrodynamics

The first step of the fountain simulation we want to create is to be able to simulate fluid and fluid particles. In order to do this, we are implementing the smoothed particle hydrodynamics algorithm used to simulate continuums models. This is a physically based simulation of particles developed by Gingold, Monaghan in 1977 firstly for astronomical studies and it was later on adapted to fluid simulation in computer graphics. This method uses a Lagrangian approach. The volume is discretized in particles, and each particle represents a certain amount of volume with physical properties (like density, pressure...) that varies continuously in space. Therefore, it is the particles that stores the position, velocity, density and pressure attributes. A quantity can be evaluated as a weighted average of particle values using this formula that we can differentiate (here, the quantity q at the particle i) (Fig.2).

W is the smooth kernel function used to interpolate, it gives the weight of the impact of one particle on another. We use this to compute density and pressure at each particle (Fig.3, 4), pressure forces (Fig.5) and viscosity (Fig.6). Knowing the fundamental principle of dynamics (Fig.7), we can then compute the acceleration of a particle.

$$
\begin{aligned}
q_i &= \sum_j \frac{m_j}{\rho_j} q_j W_{ij} \\
\nabla q_i &= \sum_j \frac{m_j}{\rho_j} q_j \nabla W_{ij} \\
\nabla \cdot \mathbf{q}_i &= \sum_j m_j \mathbf{q}_j \cdot \nabla W_{ij} \\
\nabla^2 q_i &= \sum_j \frac{m_j}{\rho_j} q_j \nabla^2 W_{ij}
\end{aligned}
$$

Figure 2: Quantity evaluation at particle i

$$
\begin{aligned}
\rho_i &= \sum_j \frac{m_j}{\rho_j} \rho_j W_{ij} \\
&= \sum_j m_j W_{ij}
\end{aligned}
$$

Figure 3: Density at particle i

$$p_i = k(\rho_i - \rho_0)$$

Figure 4: Pressure at particle i

$$\mathbf{f}_i^{\text{pressure}} = -\frac{m_i}{\rho_i} \nabla p_i = -\frac{m_i}{\rho_i} \sum_j \frac{m_j}{\rho_j} p_j \nabla W_{ij}$$

Figure 5: Pressure force at particle i

$$\mathbf{f}_i^{\text{viscosity}} = \nu m_i \sum_j \frac{m_j}{\rho_j} \mathbf{u}_{ji} \nabla^2 W_{ij}$$

Figure 6: Viscosity force at particle i

$$\frac{D\mathbf{v}_i}{Dt} = \mathbf{a}_i^{\text{body}} + \mathbf{a}_i^{\text{pressure}} + \mathbf{a}_i^{\text{viscosity}} = \mathbf{a}_i$$

Figure 7: Fundamental principle of dynamics

To sum up, the principle of the algorithm is the following :

- For all particles i, compute its density and pressure

- For all particles i, compute body, pressure and viscosity accelerations, and compute the total acceleration

- Compute the new velocity and position of the particle from the acceleration (Fig.8).

We used the code developed during the last lab class to start our simulation.

$$
\begin{aligned}
\mathbf{v}_i(t + \Delta t) &= \mathbf{v}_i(t) + \Delta t \mathbf{a}_i(t) \\
\mathbf{x}_i(t + \Delta t) &= \mathbf{x}_i(t) + \Delta t \mathbf{v}_i(t + \Delta t)
\end{aligned}
$$

Figure 8: Fundamental principle of dynamics

2.2 Adapting SPH to 3D

The SPH method develop during the lab was for a two dimension simulation of fluid. In order to get the real fountain simulation we needed to adapt it to 3 dimensions. The first step was to create particles with a non-zero value for the z coordinate. The particles were now moving around in 3D. Then, in order to get a full view of the scene we needed to add some controls of the mouse to move around in the 3D scene and see the simulation from different angles. The following code allowed us to correlate movement of the mouse onto particular movement of the scene depending on the buttons and key pressed.

```
1  //p0 previous position of the mouse
2  //p1 current position of the mouse
3  if(!user.cursor_on_gui){
4    if(state.mouse_click_left && !state.
         key_ctrl)
5      scene.camera.
           manipulator_rotate_trackball(
           p0, p1);
6    if(state.mouse_click_left && state.
         key_ctrl)
7      camera.
           manipulator_translate_in_plane
           (p1-p0);
8    if(state.mouse_click_right)
9      camera.
           manipulator_scale_distance_to_center
           ( (p1-p0).y );
10 }
```

Finally we also had to adapt the boundaries into 3D. In the `simulate` function we added the possible conditions on the z axis.

2.3 Throwing particles

Now that we have a simulation of the behavior of the particles thanks to the SPH, we want to simulate a fountain. Instead of creating particles in a cube we had to shoot the particles out from a source point. To do this, we created the function `updateParticle()` to add the particles one after the other with an initial speed to propel them. This function is called at every time update and each time one particle is created.

```
1  particle_element particle;
2  particle.p = {0.0f,-0.5f,0.0f};
3  particle.v = {h/8*rand_interval(),5.0f
       ,h/8*rand_interval()};
4  particles.push_back(particle);
```

We first chose identical initial conditions (starting point and speed) for all the particles. With

the SPH simulation, the particles will have different trajectories even with the same initial conditions, thanks to the pressure which will orient the particles differently, pushing them away from each other. For example, if we simultaneously add several particles with the same initial conditions (for example, same position and zero speed), we would obtain a "fireworks" effect (see Fig.9, where the blue line shows the trajectory of the particles) thanks to the pressure, and the particles would not all have the same trajectory. By adding the particles one after

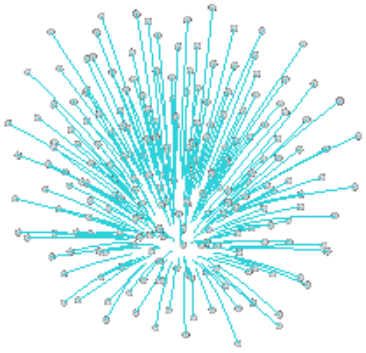

Figure 9: Firework effect

the other with a speed of [0,5,0], we get the following result (Fig.10).

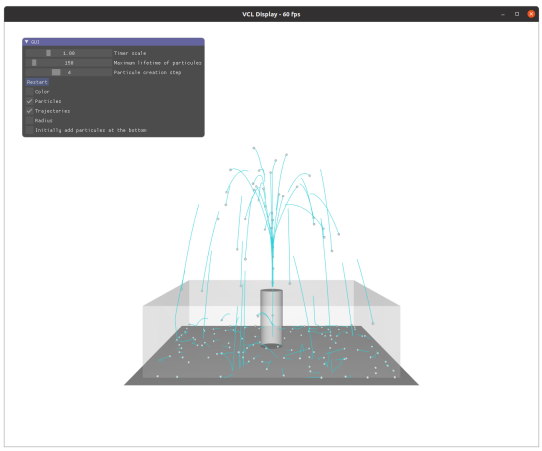

Figure 10: Particles fired one after the other at a speed of [0,5,0]

Then we wanted to achieve this "fountain with different levels" effect. For this, we set the particles with different initial speed values, one for each desired level. The idea is to fire once with a define speed and the next one at the other speed and over again. In order to know at each given time step with what speed we have to fire the particle we've used a counter `fountain_water_storey`.

```
if(fountain_water_storey == 0)
    height = 4.0f;
else if (fountain_water_storey == 1)
    height = 7.0f;

//Create the particles HERE

fountain_water_storey+=1;
fountain_water_storey =
    fountain_water_storey % 2;
```

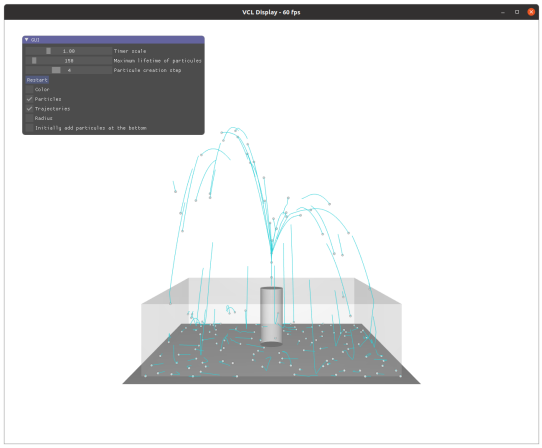

Figure 11: Particles fired with two initial speeds

However, we still don't get the wanted look for the fountain. We can see in (Fig.11) that the particles were mainly shot at one angle. The look we wanted was as in (Fig.1) with at each height, we want spurt of water all around at given angles. In order to get this effect we added the variable `fountain_angle` that will take alternatively given angles. The angle will start from 0 rad to 2*π and the step is 10 rad. The fountain angle will be incremented and re-initialized when it reaches its limit just like `fountain_water_storey`. Each new is thrown with a speed of `[cos(fountain_angle)/50,height,sin(fountain_angle)/50]`. Now that the particles are thrown alternatively at each defined angles to get a more clean fountain look as we can see in the (Fig.12).

2.4 Creating the water surface

In the lab done in class, the water surface was visually simulated by a blue color texture created by a field function, representing the density of the area. By adapting the lab to 3D we couldn't use this method adapted to a 2D simulation.

We had to think of other ways to simulate the water surface. In the article SPH Fluids in Computer Graphics [1], they present different solutions for surface reconstruction including the Marching Cubes algorithm. However those solutions appeared complex and as they were not the main fo-

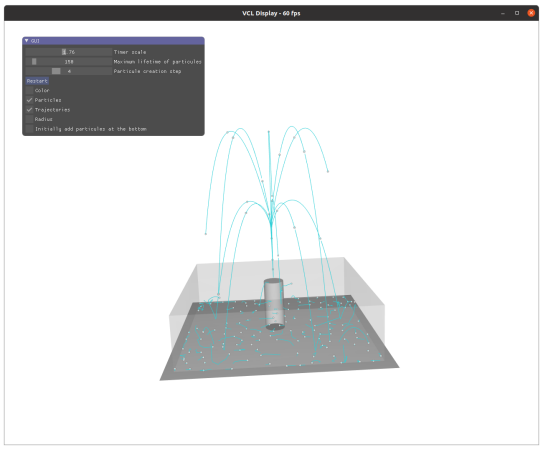

Figure 12: Particles fired with two initial speeds and varying angles

cus of the project, we've decided to try and find another less perfect but simpler solution to best fit our project's purpose.

The first idea was to have smaller particles but more of them and color code them according to their velocity in order to get a more fluid like aspect as we can see in (Fig.13) taken from the SPH article [1]. However, due to computational issues it was

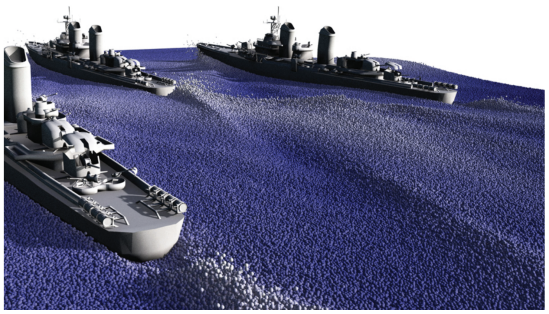

Figure 13: Image created with 19 million particles color coded with velocity

not possible to have that many or enough particles to have this aspect as we are going to explore in the next section.

The second idea we had was to work with billboards. We created a blue billboard with a Gaussian distribution to create the water aspect. There were two possibilities then. The first one was to compute a density field on a grid like in the lab. The billboard is added with a certain opacity on this grid depending on the field value. The second one was to put one billboard for each particle. This is the idea that we preferred as the density field gave an idea of blue smoke instead of water and it was long to compute which led the simulation to drop in fps down to 5fps even with a low number of particles.

However because of our computational limit, we could not get much particles onto our simulation as previously stated. Hence when we added one billboard per particle we did not get the wanted effect, there were really sparse particles. We thought of adding a blue trajectory of every particle to have the idea of a "flow" of particle from one particle. For this we've used the `trajectory_drawable` class of the VCL library. Then by adding a billboard over one out of two positions on the trajectory its gives the impression there are more particles to our simulation and gives a better idea of the water flow of the fountain even with a low number of particles.

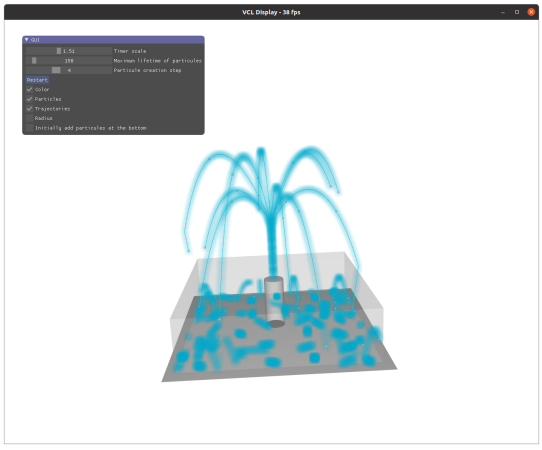

Figure 14: Billboards on the trajectory

2.5 Dealing with many particles

With this simulation, particles are continuously created. The computers we used for the simulation have a limited capacity, and with too many particles,the simulation becomes too slow. To avoid having too many particles, we recycle them. After a certain number of particles created, the particles which are propelled by the fountain are the oldest particles, already created. This is actually a realistic simulation for a fountain since generally the water which gushes out of the fountain is pumped off the basin and reused continuously.

In order to know which particles to reuse, we added a `lifetime` to each particles that is incremented at every clock time. When the lifetime of a particle is higher than a certain threshold, it is pushed back on a list. If this list is not empty, instead of creating a new particle we will pop the front of the list and recycle the particle. This will insure that the number of particles will not skyrocket but be contained to a certain threshold that is settable with the GUI.

However, it was still not enough to have a nice, not explosive and really fluid simulation. Since the particles are either created or recycled at every clock time, there are popping out really fast

and sometimes thrown particles collide and give an explosive aspect especially with the two height of the simulation. Plus, it also made the number of particles increase really fast which made the frame per second rate drop really fast. In order to prevent all of this and have a more realistic fountain of particles, we've added a particle generation step to control the flow of fluid. This step is configurable from the GUI and is initially set to 4, which mean a particle is either created or recycled every 4 clock times.

Finally, since we still get a fast drop in frame per second, its takes quite a long time before having enough water inside the basin to have a realistic look. In order to get this realistic look right away, we've decided to add the possibility to add initially water in the basin. From this we get a fair result (Fig.15).

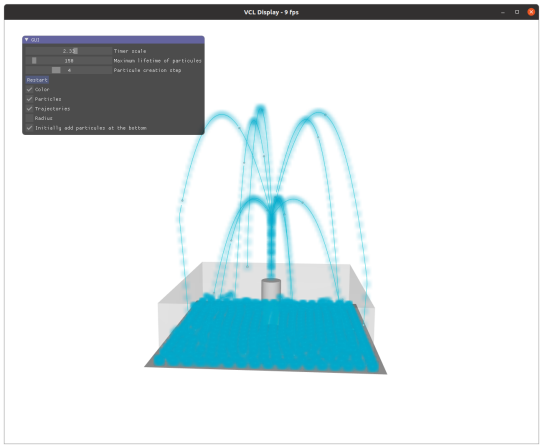

Figure 15: Fountain with initial particles at the bottom

3 User guide

In order to be able to run the code on your computer you need to install:

- C++ compilation tools
- CMake
- GLFW library

In your terminal:

```
cd path/to/folder
mkdir build
cd build
cmake ..
make
cd ..
./build/3DFountainSimulation
```

When the simulation is launched, in order to move around you can use the following controls:

- De-zoom with the left click
- Rotate with the right click
- Move horizontally and vertically by holding the Control key and using the right click

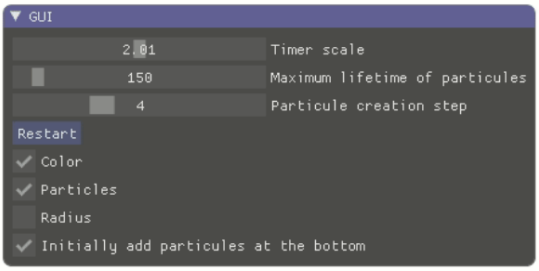

Figure 16: Interface to configure the simulation

4 Conclusion

4.1 Future Perspectives

The main problem with this simulation is that we are limited in number of particles, because the simulation is too slow. One solution that we could implement is to use a particle neighbor search. Indeed, in the calculation of the quantities (Fig. 2), it isn't necessary for j to go through all the particles, since from a certain radius between particles i and j, the weight Wij becomes zero. We could therefore limit j to the neighboring particles of i, with a neighborhood radius to be defined depending on the kernel.

To be able to efficiently find the neighbors of a particle, we would have to define a structure. One method is to use a uniform grid (Fig.17). Then, we register each particle to a cell of this grid, and each cell contains indices of particles that belong to the cell. To find the neighbors of a particle, we then just need to find the cell to which it belongs, and get the particles which belong to the neighboring cells. In the algorithm, we would need to build the neighbors at each step before computing the quantities.

4.2 Conclusion

Despite the slowness of the simulation, we obtain a fairly satisfying result. As we used the Navier-Stokes equation, the particles respect the physical laws of fluids. We ajusted some parameters like the initial speed of the particles to have a rendering that matches more our visual representation of a fountain. The rendering of the fountain comes close to what we expected (Fig.1), so our two main goals are completed. We would have liked to be

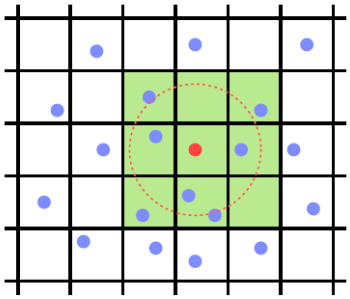

Figure 17: Grid neighborhood

able to implement the method described in "Future perspectives", but unfortunately we did not have the time to do it because of our internships which started at the same time as this project.

References

[1] B. S. A. K. M. T. Markus Ihmsen, Jens Orthmann, "Sph fluids in computer graphics," 2014.