CS 184: Computer Graphics and Imaging, Spring 2017, Final Report: Rainbow Musical Fountain Simulation

BOSEN DING and SIJIA TENG, University of California, Berkeley

Position based dynamics is a crucial framework in fluid simulation. Recent works improved the efficiency, and enforced realism. This algorithm enforces incompressibility by computing corrective position vector, handles tensile instability by adding artificial surface tension, accounts for the loss of energy due to position based method by applying vorticity confinement and smoothens particle movement by adding viscosity. In this work, we apply the position based dynamics framework integrated with a density solver to a highly dynamic moving fountain simulation.

Additional Key Words and Phrases: fluid simulation, SPH, position based dynamics, musical fountain simulation

1 INTRODUCTION

Fluids, especially water, are responsible for many beautiful phenomena in the nature. However, the difficulty of water simulation used to prevent people from representing the beauty of water.

In [3], the famous Smoothed Particle Hydrogynamics (SPH) method is proposed, with many obvious advantages. But it does not work well when there are deficient neighbors of a particle. In [8], a method to deal with the neighbor deficiency problem is proposed, but it is limited by large-time step. In [6], the Position Based Dynamics framework is introduced, which is stable and can be integrated with many confinements to tackle the neighbor deficiency problem, ensure realism, and achieve real-time simulation.

In this work, we implement Position Based Fluids algorithm to simulate water. In section 2, we introduced the position based dynamics algorithm. Section 2.1 shows the Euler's method for position prediction. Section 2.2 introduced the efficient neighbor finding algorithm. Section 2.3 shows how we enforce incompressibility by computing corrective position vector. Section 2.4 handles tensile instability by adding artificial surface tension. Section 2.5 accounts for the loss of energy due to position based method by applying vorticity confinement and smoothens particle movement by adding viscosity. In section 3, we use the water simulation algorithm to build a rainbow musical fountain which dances to the music. Section 4 shows the several features we add to the GUI, such as rotating perspective and zooming in/out. Section 5 gives the results for the simulation.

2 POSITIN BASED DYNAMICS FRAMEWORK

In this work, we adopt the Position Based Dynamics framework integrated with an interative density solver for the main simulation loop, which is described in [2].

Our main simulation loop has four components. The first component updates velocity, computes next position using explicit Euler's method and updates bounding box coordinates for each particle. Then the second part finds neighbors for each particle using the updated bounding box system. Then the third part enforces incompressibility by iteratively computing the corrective $\Delta\mathbf{p}$ for each particle. An artificial force is applied to $\Delta\mathbf{p}$ calculation to handle tensile instability. Finally the last part creates surface tension, adds vorticity confinement and applies viscosity to the particles. The algorithm for the main simulation setp is illustrated in Alg. 1.

ALGORITHM 1: Main Simulation Step

```
for each Particle par do
    apply forces par.v+ = par.force · Δt;
    calculate next position par.p_next = par.p + par.v · Δt;
    /* Update and collide because of Δp                    */
    collision detection and responce;
    update bbox;
end
for each Particle par do
    find neighbors par.neighbors;
end
while iteration < limit do
    for each Particle par do
        calculate par.λ;
    end
    for each Particle par do
        calculate par.Δp;
        /* Collide because of Δp                           */
        collision detection and responce;
    end
    for each Particle par do
        par.p_next+ = Δp;
        /* Update because of Δp                            */
        update bbox;
    end
end
for each Particle par do
    par.v = (par.p_next − par.p)/Δt;
    add vorticity;
    add XSPH viscosity;
    par.p = par.p_next;
end
```

2.1 Explicit Euler's method for position and velocity update

In this part, we update velocity by applying external forces on each particle and then calculate each particle's next position by applying Euler's method. The velocity used for position update is the velocity calculated from last loop, so we are using explicit Euler's method. Since we assume the mass of each particle to be the same, so we directly apply acceleration to the velocity of each particle without

using Newton's second law.

$$X^* = X + \Delta t \dot{X}$$

$$\dot{X}^* = \dot{X} + \Delta t \ddot{X}$$

Since we apply the algorithm to fountain simulation, when the particles are not in the fountain, the external force applied here is only gravity, but when the particles are in the fountian faucet, a pumping force will be applied to the particles.

2.2 Neighbor Finding

We calculate the neighbors of each particle before proceeding to the density solver step. In order to avoid the $O(n^2)$ running time of the naive double for loop algorithm, we use a bounding box method. The simulation space, namely the cubic space encircled by six planes, is divivided into many subspaces, each of which is a cube of size length dim. Then when finding the neighbors of a particle, we calculate its bounding box coordinates and then check all the particles within this box and its adjacent boxes(up to 26 adjacent boxes). For each particle in the neighborhood boxes, if the distance between this particle and the target particle is less than h, the particle will be added to the neighbor list. In this way, we can find the neighbors for each particle efficiently. In order to enforce incompressibility, we iteratively solve a system of non-linear equations which would calculate the corrective $\Delta \mathbf{p}$ necessary in order to bring the density close to rest density of the water. The boudning box anf neighbor finding algorithm is illustrated in Alg. 2.

ALGORITHM 2: Bounding Box and Neighbor Finding Algorithm

```
/* Assign particles to bounding boxes                          */
Input: each Particle par, bounding boxes bbox
for each Particle do
    /* Get bounding box coords: get_bbox_coords()              */
    index = int(par.p/bbox.dim) + 0.5 · bbox.size;
    /* Assign to the corresponding box                         */
    bbox[index].push_back(par);
end
/* Update particles in bounding boxes                          */
Input: each Particle par, bounding boxes bbox
for each Particle do
    index_old = get_bbox_coords(p);
    index_new = get_bbox_coords(p_next);
    if index_old != index_new then
        bbox[index_old].erase(par);
        bbox[index_new].push_back(par);
    end
end
/* Find neighbors                                              */
Input: each Particle par, bounding boxes bbox
for adjacent bounding boxes b within bbox size do
    for each Particle par_ngb in b do
        if |par_ngb.p − par.p| < bbox.dim then
            par.neighbors.push_back(par_ngh);
        end
    end
end
```

2.3 Enforcing Incompressibility

To estimate the density of a particle, we use its position and the positions of its neighbor particles. We adopt the standard SPH density estimator. Because all particles have the same mass, so the mass term m would be dropped from the density estimator.

$$\rho_i = \sum_j m_j W(\mathbf{p_i} - \mathbf{p_j}, h)$$

Then, the constraint for each particle can be written as

$$C_i(\mathbf{p_1}, ..., \mathbf{p_n}) = \frac{\rho_i}{\rho_0} - 1$$

where ρ_0 stands for the rest density ρ_i is the estimated density. In our case, because each particle is initialized 0.1 distance away, one unit cube has 1000 particles and therefore ρ_0 in our project is set to 1000.

When the density of a particle is equal to the rest density, the constraint for this particle is equal to 0. Thus the desired corrective $\Delta \mathbf{p}$ satisfies C($\mathbf{p} + \Delta \mathbf{p}$) = 0. The solution can be found by Newton steps along the constraint gradient as shown in [2].

$$\Delta \mathbf{p} \approx \nabla C(\mathbf{p}) \lambda$$

$$C(\mathbf{p} + \Delta \mathbf{p}) \approx C(\mathbf{p}) + \nabla C^T \Delta \mathbf{p} = 0$$

$$C(\mathbf{p} + \Delta \mathbf{p}) \approx C(\mathbf{p}) + \nabla C^T \nabla C(\mathbf{p}) \lambda = 0$$

The gradient of the constarint function with respect to a particle k is given by

$$\nabla_{\mathbf{p}_k} C_i = \frac{1}{\rho_0} \sum_j \nabla_{\mathbf{p}_k} W(\mathbf{p_i} - \mathbf{p_j}, h)$$

The gradient has two cases.

$$\nabla_{\mathbf{p}_k} C_i = \frac{1}{\rho_0} \begin{cases} \sum_j \nabla_{\mathbf{p}_k} W(\mathbf{p_i} - \mathbf{p_j}, h) & if\ k = i \\ -\mathbf{p}_k W(\mathbf{p_i} - \mathbf{p_j}, h) & if\ k = j \end{cases}$$

Therefore, we can solve for λ with the previous Newton step equation, which gives

$$\lambda_i = \frac{-C_i(\mathbf{p}_1, ..., \mathbf{p}_n)}{\sum_k |\nabla_{\mathbf{p}_k} C_i|^2}$$

When particles are close to seperating, namely when they do not have many neighbors around, the denominator of the above solution becomes unstable, therefore a relaxation parameter is needed and the solution becomes

$$\lambda_i = \frac{-C_i(\mathbf{p}_1, ..., \mathbf{p}_n)}{\sum_k |\nabla_{\mathbf{p}_k} C_i|^2 + \epsilon}$$

Then, we have the solution to the corrective $\Delta \mathbf{p_i}$

$$\Delta \mathbf{p_i} = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j) \nabla W(\mathbf{p_i} - \mathbf{p_j}, h)$$

We use the Poly6 kernel function to calculate density, and use the Spiky kernel function for gradient calculation. Both of the kernel functions are given in [5]. Since [5] does not give a gradient expression for the Spiky kernel, we give the expression here:

$$\nabla_{\mathbf{W}_{\text{spiky}}}(\mathbf{r}, h) = \begin{cases} -45/(\pi \cdot h^6)(h - ||r||)^2 \frac{\mathbf{r}}{||r||}, & if\ 0 < r < h \\ 0, & else \end{cases}$$

We create the simulation inside a box frame with six planes, so the particles are all restricted within the frame box. Alg. 3 shows how we deal with collision between particles and the planes of the frame box. Specifically, there will be two kinds of collision due to position update, one is caused by velocity which is the Euler's method introduced in section 2.1, the other is caused by the correction vector $\Delta \mathbf{p}$ solved from the density constraint as is shown in this section.

ALGORITHM 3: Collision Detection and Response

Input: Particle position p_in, particle velocity v_in, SPH step $\Delta \mathbf{p}$, time step $\Delta \mathbf{t}$.
Output: Boolean value of whether bonce happens, new particle position p_out, new velocity v_out.

```
for each of the six planes do
    p_out = p_in + v_in · t + Δp;
    if p_out lies out of the frame box then
        if collision because of v_in then
            t1 = dot(point − p_in, normal)/dot(v_in, normal);
            v_out = reflect(v_in, normal);
            p_out = p_in + t1 · v_in+Cv_out · (Δt − t1);
        end
        if collision because of Δp then
            t1 = dot(point − p_in, normal)/dot(Δ p, normal);
            p_out = p_in + t1 · Δp + ϵ;
        end
    end
    if t1 is the minimun in the loop then
        return ture, p_out, v_out
    end
end
```

2.4 Tensile Instability

The traditional SPH algorithm would cause particle clustering or clumping when a particle does not have enough neighbors and cannot satisfy the rest density. Therefore, we follow the work [4] to add an artificial pressure term to address this problem.

$$s_{corr} = -k(\frac{W(\mathbf{p_i} - \mathbf{p_j}, h)}{W(\Delta \mathbf{q}, h)})^n$$

$$\Delta \mathbf{p_i} = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j + s_{corr}) \nabla W(\mathbf{p_i} - \mathbf{p_j}, h)$$

In our case, $|\Delta \mathbf{q}| = 0.2h$, $k = 0.001$, and $n = 4$.

2.5 Vorticity Confinement and Viscosity

To account for the loss of energy due to position based framework, we implement vorticity confinement to compensate the loss, as is descreibed in [1]. We estimate vorticity by the estimator introduced in [3]:

$$\omega_i = \nabla \times \mathbf{v} = \sum_j (\mathbf{v_j} - \mathbf{v_i}) \times \nabla_{\mathbf{p}_j} W(\mathbf{p}_i - \mathbf{p}_j, h)$$

Then the force upon each particle due to vorticity confinement is

$$f_i^{vorticity} = \epsilon(\frac{\nabla |\omega|_i}{|\nabla |\omega|_i|} \times \omega_i)$$

We also apply XSPH viscosity for coherent movement of particles [7].

$$\mathbf{v}_i^{new} = \mathbf{v}_i + c \sum_j (\mathbf{v}_j - \mathbf{v}_i) \cdot W(\mathbf{p}_i - \mathbf{p}_j, h)$$

In our foutain simulation, we tune the parameters for the best realism and choose $c = 0.001$, and $\epsilon = 0.002$.

3 MUSICAL FOUNTAIN IMPLEMENTATION

3.1 Fountain Implementation

When water paticles flow into the cylindrical space of the fountian, the fountain will pump up the water through the fountain mouth. In order to simulate the fountain effect, we specify the bounding box coordinates of the fountain space, when the water particles are within this specified space, their velocity will be modified to the pumping speed of the fountain. To represent the amorphous state of the water, we add a random variation to the pumping speed. For example the pumping speed in one of our simulations is $10+5 \cdot rand()$.

3.2 Synchrony with Music

A timestamp is added to each simulation step to match the progress of music. In the music fountian simulation, a piece of music notes is loaded. In each simulation step, the corresponding music note to this step would be read and then the pumping speed at this step is decided by the pitch of this note. The higher the pitch, the larger the pumping speed.

4 GUI FEATURES

We add several features to improve the user experience of the GUI. Users can zoom in/out and rotate to change their perspectives. The GUI also can pause and resume simulation at any time. Furthermore, A step-by-step simulation is also made possible in this GUI, so users can view exactly how particles move during each time step.

5 SIMULATION RESULTS

5.1 Falling Simulation

We first simulate cube of water particles falling to the bottom plane, as is shown in Fig. 1. We used 96k particles in this simulation.

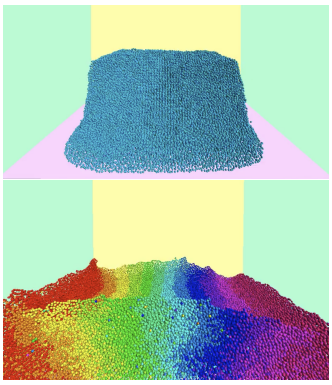

Fig. 1. Simulation of a cube of water particles falling to the ground. (): a blue cube of particles collide with the ground; (): a rainbow cube of particles waves after falling.

In Fig. 2, the blue line shows the density of the particles falling process, and the orange line shows the rest density. In the first 40 frames, the density is converging to 1000. As can be been, the convergence takes less than 5 frames. At 50th frame, collision happens. As can be seen, there is a little peak of the density after collision happens, but it recovers fast and the density converge to a stable value at 1000.

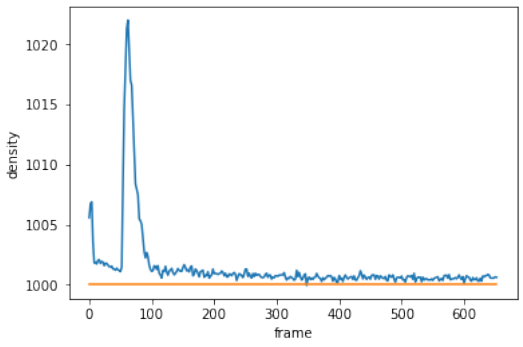

Fig. 2. The density curve of water falling simulation.

5.2 Fountain simulation

In this section, we show some results of fountain simulation. The faucet is located in the middle of the bottom plane with a diameter 0.15. The size of the bottom is $9.45 * 5.85$. In Fig. 3, we show two screenshots of the fountain.

In Fig. 4, we show different shape of the fountain. In Fig. 5, we show different height of the fountain. The faucet size in Fig. 4 and

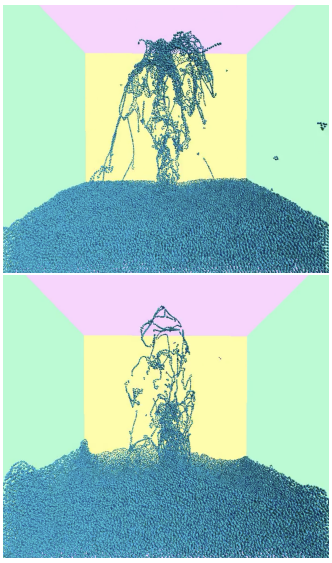

Fig. 3. Simulation of the fountain.

Fig. 5 double the faucet size in Fig. 3 so the particles in the air looks denser and has more connections with each other. There are always some waves in the water on the ground due to the continuous pumping and falling of the particles. In Fig. 4 there are some small group of particles in the scene, while in Fig. 5 better viscosity and bouncing effect is used and less small group of particles can be seen.

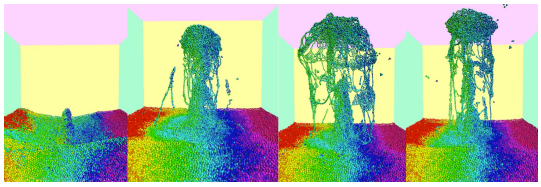

Fig. 4. The rainbow musical fountain in different shape.

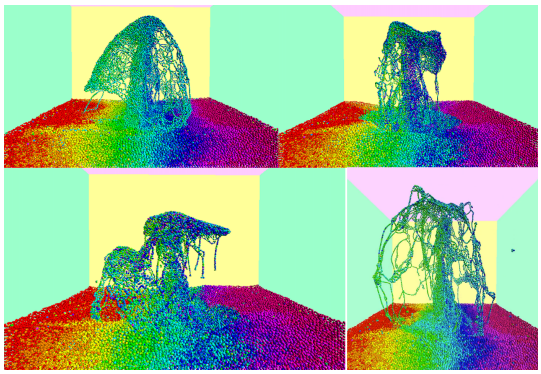

Fig. 5. The rainbow fountain in different height.

In Fig. 6, we show the density of the falling process followed by the fountain effect. The simulation of the figure is tested with a constant fountain, i.e., the velocity the faucet gives to the particles is a constant value at 15. As can be seen, the convergence in the beginning is fast (less than 5 frames). The density also has a small peak when collision happens, but during the fountain effect, the density stays stable. These results show our fountain simulation works well. It can generate a natural shape, achieve the desired height, as well as maintain a n aimed density.

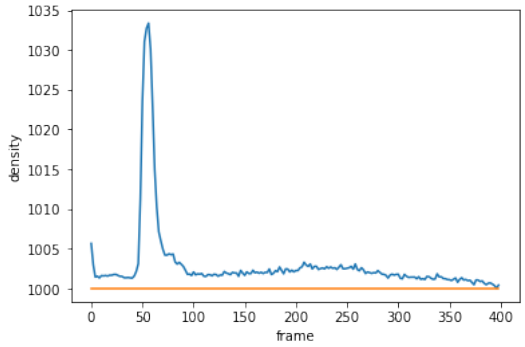

Fig. 6. The average density curve in fountain simulation.

6 FUTURE WORK

Since the authors do not have a working GPU, all the simulation is implemented on CPU and therefore this is not a real time simulation. In future work, GPU implementation can be done, and code efficiency and performance is to be improved. For example, we can only render particles that we can see. After real-time simulation is achieved, we can do real synchronization with music, and instead of reading notes, we can read amplitude of the music sound, or even read people's voice. Thus more interactive and applications can be achieved.

Another direction of future work lies on the rendering of the water surface. [9] gives a good method for smooth fluid surface rendering. The authors have tried to implement this but did not have enough time to make it work.

ACKNOWLEDGMENTS

The authors would like to thank Prof. Ng and all the GSIs in this course. The authors enjoyed the whole semester really much and got a lot of inspiration and courage from them during the implementation of this project. The authors also want to thank swl who provide the starter code for this project with based particle rendering. This gives the authors a choice to not modify other open source starter codes implemnted with CUDA.

REFERENCES

[1] Jeong-Mo Hong, Ho-Young Lee, Jong-Chul Yoon, and Chang-Hun Kim. 2008. Bubbles alive. In *ACM Transactions on Graphics (TOG)*, Vol. 27. ACM, 48.

[2] Miles Macklin and Matthias Müller. 2013. Position based fluids. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 104.

[3] Joe J Monaghan. 1992. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics* 30, 1 (1992), 543–574.

[4] Joseph J Monaghan. 2000. SPH without a tensile instability. *J. Comput. Phys.* 159, 2 (2000), 290–311.

[5] Matthias Müller, David Charypar, and Markus Gross. 2003. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Eurographics Association, 154–159.

[6] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. 2007. Position based dynamics. *Journal of Visual Communication and Image Representation* 18, 2 (2007), 109–118.

[7] Hagit Schechter and Robert Bridson. 2012. Ghost SPH for animating water. *ACM Transactions on Graphics (TOG)* 31, 4 (2012), 61.

[8] Barbara Solenthaler and Renato Pajarola. 2009. Predictive-corrective incompressible SPH. In *ACM transactions on graphics (TOG)*, Vol. 28. ACM, 40.

[9] Wladimir J van der Laan, Simon Green, and Miguel Sainz. 2009. Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. ACM, 91–98.