Position-Based Fluid Simulation

Samantha Cohen Sarah Leung

I. Introduction

For our final project, we chose to implement water simulation using position-based fluid dynamics. This is a method developed by Macklin and Muller [1] that is essentially a combination of traditional smoothed-particle hydrodynamics (SPH) and position-based dynamics (PBD), which solves the dynamics of a system through constraint projection. We saw this topic as an opportunity to expand on several previous projects, as well as implement additional features such as solid object interaction and rendering.

II. Overview of Position Based Fluids

PBF vs. SPH

The position-based fluids (PBF) method combines SPH [2], which is the traditional Lagrangian approach to simulating fluids, and position-based dynamics [3]. The dynamics of fluids are governed by two main laws of physics. The first is the Navier-Stokes equation:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{1}{\rho}\nabla p + \nu \Delta u + g$$

The second is the conservation of momentum:

$$\nabla \cdot \mathbf{u} = 0$$

The Navier-Stokes equation relates the velocity $\mathbf{u}$, pressure p, density ρ, and viscosity ν, in the dynamics of fluids. Particle methods for simulation solve the Navier-Stokes equations on particles of water, looking primarily at the pressure and viscosity terms to describe its motion.

In SPH, a Lagrangian approach, each particle carries properties, such as mass, pressure, density, position, and velocity. Since position-based fluids builds on PBD, the base properties that each particle carries is mass, position, and velocity. In our datastructure, we also have each particle carrying its density, which is solved for in each step.

Some of the advantages the position-based fluids method has over SPH is integration stability, which allows for larger time steps. Additionally, we are able to use a smaller smoothing radius for the density constraint, while SPH requires a large number of neighbors to prevent deficiencies in the simulation. Finally, SPH has been known to have issues of tensile instability. In PBF, we add an artificial pressure term for surface tension, which reduces particle clumping and provides, for example, a more visually realistic splashing effect.

Position Update

Position-based fluids primarily solves a density constraint, which enforces the incompressibility of water. The density constraint is a function of the particle i and its neighbors.

$$C_i(\mathbf{p}_1, \ldots, \mathbf{p}_n) = \frac{\rho_i}{\rho_0} - 1$$

Here $\rho_0 = 1000\ kg/m^3$ is the rest density of water and ρ_i is the density at particle i given by the SPH density estimator, where m_j is the mass on particle j:

$$\rho_i = \sum_j m_j W(\mathbf{p}_i - \mathbf{p}_j, h)$$

The simulation uses a Jacobi solver to determine the position update, as a result of the effect of the neighboring particles. This requires the calculation of the scaling factor, λ and artificial pressure term, s_{corr}, for the overall position delta, $\Delta \mathbf{p}$.

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

$$s_{corr} = -k \left(\frac{W(\mathbf{p}_i - \mathbf{p}_j, h)}{W(\Delta \mathbf{q}, h)} \right)^n$$

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

XSPH Viscosity

PBF deviates from the SPH viscosity calculation by using and artificial XSPH viscosity. This is calculated as a velocity update, which is done as a post-processing step, after the Jacobi solver loop.

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

Kernel Functions

SPH and PBF use kernel functions to evaluate some of the quantities. In general, the Poly6 kernel is used for the density and viscosity calculations, while the gradient quantities for pressure or position update are evaluated using the gradient of the Spiky kernel. These kernels are listed below. Here, h is the neighborhood radius and $\mathbf{r}$ is typically the distance vector from the neighbor to the particle.

$$W_{poly6}(\mathbf{r}, h) = \begin{cases} \frac{315}{64\pi h^9}(h^2 - |\mathbf{r}|^2)^3, & \text{if } 0 \leq |\mathbf{r}| \leq h \\ 0, & \text{otherwise} \end{cases}$$

$$\nabla W_{spiky}(\mathbf{r}, h) = \begin{cases} \frac{45}{\pi h^6}(h - |\mathbf{r}|)^2 \frac{\mathbf{r}}{|\mathbf{r}|}, & \text{if } 0 \leq |\mathbf{r}| \leq h \\ \mathbf{0}, & \text{otherwise} \end{cases}$$

III. Aspects of the Simulation

Solver Loop

The simulation runs through a loop at each time step, Δt. The predicted positions are first updated using velocity change due to external forces, which in our simulation is only gravity. Next, neighbor search is performed on these predicted positions, and the neighbors of each particle are stored. The following step is the solver loop of the simulation, run for 5-10 iterations typically, to solve the density constraint and artificial pressure term to obtain the position update. As a post-process, we update velocity according to the XSPH viscosity term, and finally perform collision detection. The last step updates the current position to the predicted position.

Spatial Hashing

Initially, we started with an $O(n^2)$ neighbor search. Then we implemented spatial hashing for neighbor search as an acceleration structure for the simulation. The grid is discretized into cells, with width h equal to the neighborhood radius. At each time step, each particle's grid coordinates are determined and hashed to a scalar key as in the equations below, with neighborhood radius h, minimum grid cell coordinates $\mathbf{g}_{min}$, and grid dimensions $\mathbf{g}$.

$$(i, j, k) = \text{ceil}\left(\frac{\mathbf{p} - \mathbf{g}_{min}}{h}\right) - \mathbf{1}$$

$$\text{key}(i, j, k) = g_y g_z i + g_z j + k$$

The particle is then pushed into the vector mapped to the key. Neighbors for each particle are also determined at each time step, before the solver loop. We first determine the grid cell the particle belongs to and then consider the cell and its 26 neighboring cells. We check all particles in those 27 cells to see if they fall within the neighborhood radius.

Collision Detection

For collision detection, we implemented static collision detection with the bounds of the box. The predicted position is updated to the inner surface of the bounding box and the velocity is reflected with a coefficient of restitution. We also implemented collision detection with static object meshes, using a naive particle-triangle intersection test. This test is implemented by first performing a plane intersection test, and then evaluating the intersection point against the barycentric coordinates of the triangle. To make the mesh-intersection test slightly faster, we first perform an intersection test with the bounding box of the mesh. This optimizes the speed while most particles are far from the mesh, but once particles enter the box containing the mesh, they still need to be checked against all triangles of the mesh.

In the paper, the algorithm calls for collision detection and resolution to be performed within the Jacobi solver loop. However, we found that we could not update the velocity as required within the loop. We therefore moved the collision response part outside of the solver loop, after the viscosity calculation.

Rendering

For rendering in the simulation, we used openGL to render particles as spheres of a set radius. For visual debugging purposes, we colored the spheres according to the absolute values of the components of velocity. We attempted to use level sets to extract the water surfaces to a mesh for rendering, but ultimately did not get this working. Instead, we modified a script [4] to export particle positions and we wrote a MEL script to import the positions into Maya for rendering. All rendering was done on the CPU.

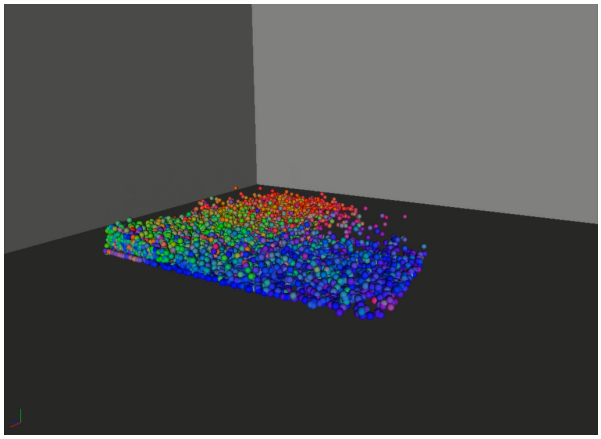
IV. Challenges

Some challenges we faced were with the basic dynamics of water using position-based fluids. It was difficult to simultaneously debug our code and tune parameters to achieve more realistic movement of water. Additionally, we had some struggles with collision detection, first with the bounding box, which was essential to be able to capture the motion of water in a contained environment. We had problems with unstable particles bouncing off the walls and gradually escaping the box.

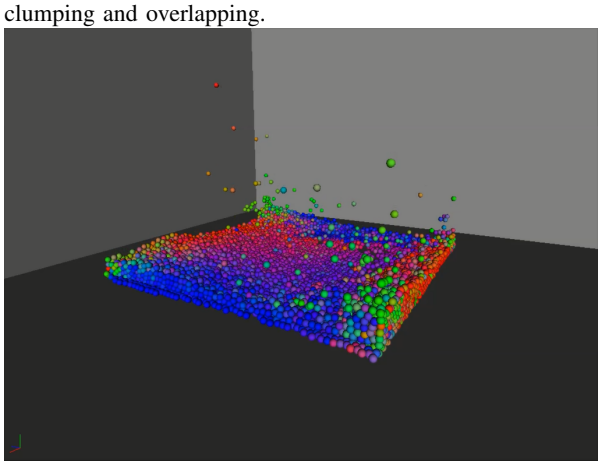
Our biggest challenge was probably the time. Our simulation does not run in real time. For 1000 particles and a time step of 0.01 seconds, each simulation step takes a little over 0.2 seconds. When we add more particles, and an object file to the scene, the simulation frame rate decreases even more. Spatial hashing was an attempt to speed up time, but the time decresase

for 1000 to 8000 particles was minimal, probably due to the overhead time for setup. OpenGL rendering also contributed to the slow speed of the simulation, since we are rendering a sphere for each particle, which need to be updated at each time step.

For rendering, we also tried to implement level sets, and briefly looked into marching cubes, but due to lack of time, were unable complete integration of a third-party code into our simulation. We therefore kept with using the spheres for water particles, but exported the positions to Maya for more aesthetic rendering.



(a) Without the artificial pressure term, s_{corr}, there is particle clumping and overlapping.



(b) With the addition of the artificial pressure term, s_{corr}, the particle distribution and surface tension is improved.

Fig. 1: Artificial pressure term for tensile stability

V. Results

The simulation achieves decent physically-based fluid motion for a smaller number of particles. We ran the simulation with the number of particles ranging from 1000 to 11375. The water was initiated as a uniform cube of particles, in the air, on the ground, or against a wall. We began with parameters suggested in the seminal paper, and tuned to stability and aesthetics. The neighborhood radius was one parameter that could be decreased to allow an increase in time optimality, while still retaining the motion of fluids. This is one advantage described in the paper that PBF has over SPH. Overall, the simulation is stables and provides fluid motion and splashing against walls.

Figure 1 shows the difference in the simulation as a result of adding the artificial pressure term. Without the term, there is clumping of particles, and deficient particle distribution. When we add the artificial term, the particle distribution is vastly improved, such that the particles are compact without overlapping, providing the effect of surface tension. Figure 2 shows an example of water interacting with a fixed rigid object. It shows how the simulation can be used with particles of different radii, and handle collisions with medium-sized meshes. With smaller particles, there tends to be more instability, especially in collisions with object meshes. Figure 3 illustrates the effect of rendering the particles in Maya, creating a more visually appealing model of water. Figure 4 compares the openGL rendering with the Maya rendering for the case of water falling on a cow.

Improvements and Future Work

Some improvements that could me made are fine-tuning of parameters and collision detection for greater stability and time optimization to achieve real-time simulation. Future work could include rendering using level set or marching cube methods.

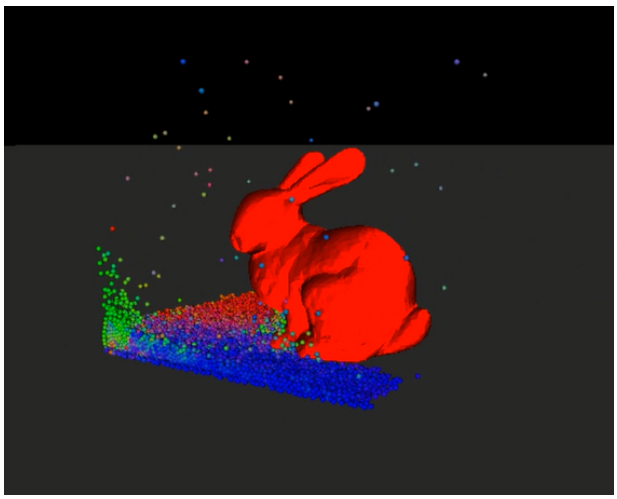


Fig. 2: Collision detection with object mesh, 8000 half-size particles

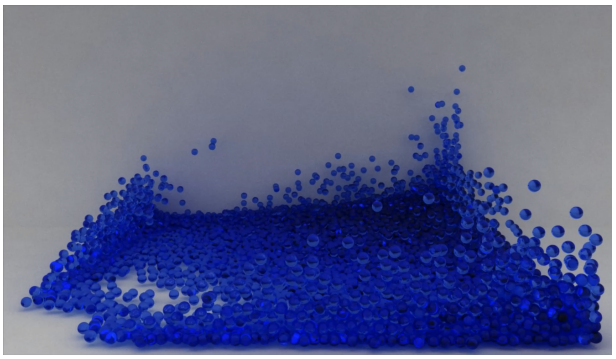
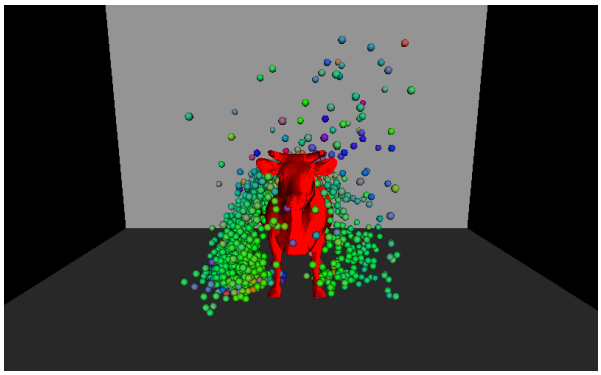
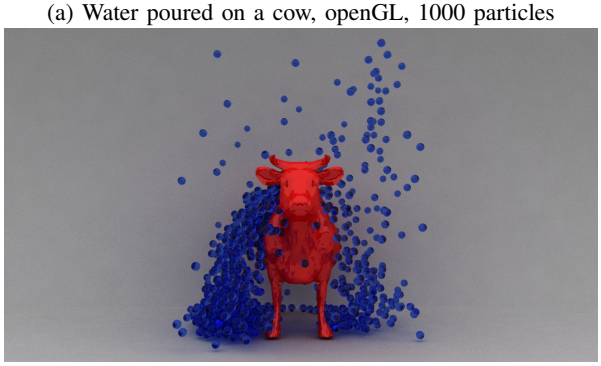


Fig. 3: Maya rendering of spherical water particles



(a) Water poured on a cow, openGL, 1000 particles



(b) Water poured on a cow, Maya, 1000 particles

Fig. 4: Comparison of rendering methods

VI. Base Code

For this project, we used the completed ClothSim as our base code, as it uses position-based dynamics. This gave us a basic framework for setting up a position-based fluid simulation, since it is a position-based dynamics simulation to begin with. We used the basic datastructure for the storing of particles and their properties. We also used the main function, with modifications, to set up the simulation, with the water and environment. We worked off of the built-in openGL rendering scheme, adding a class to render the particles as spheres. Apart from this, all the math for fluid simulation, as well as the setup of the overall algorithm loop, collision detection, and object file loading were done on our own, without the use of third-party libraries.

VII. Project Breakdown

Samantha worked on...

Sarah worked on the position delta calculation, which included the density constraint, λ for the constraint projection, and the artificial pressure term for tensile stability. She implemented the neighbor search and spatial hashing. She also worked on collision detection for the bounding box and object meshes.

References

[1] [Macklin & Muller 2013] Position Based Fluids.

[2] [Muller et al. 2003] Particle-Based Fluid Simulation for Interactive Applications.

[3] [Muller et al. 2006] Position Based Dynamics.

[4] [Batty] https://raw.githubusercontent.com/christopherbatty/Fluid3D/master/main.cpp