Final Report: Powder Master

2D Particle Simulation Game

Roy Guo Christine Sheng Mingcheng Zhu

ABSTRACT

This project explores options to optimize real-time 2D particle simulation without sacrificing simulation accuracy. We simulate particle to particle interaction as well as external forces such as wind. The particle system simulates different fluid properties, such as incompressibility, tensile instability, vorticity, and viscosity. We also add an interactive layer on top of the simulation, which enables the user to create different types of particles and to explore the interactions across those particles.

1 Introduction

Our goal is to produce a 2D simulation game modeling various "powders" that can float around in a fluid as well as interact with each other in real-time. This involves a grid-based fluid simulation using Navier-Stokes equations, as well as a particle system for the powders. The particles will interact with each other using a simple cellular automata. We choose OpenGL as our graphics engine for this project.

Several challenges this project solves include how to simulate interactions between our particle system and our grid-based system, and how to optimize the simulation so that it can be run in real-time as a game. The simulation is able to deal with thousands of particles, as well as hundreds of cells in our fluid model, so there will have to be trade-offs made in how fine-grained we want the different parts of our simulation to be.

2 Related Work

The particle system is implemented referencing Position Based Fluids, by Macklin and Muller. We modify the Smoothed Particle Hydrodynamics, as described in the paper, for 2D simulation. We are also using Google's LiquidFun as a reference.

To accelerate the simulation, binning is implemented. After binning, the simulation runtime complexity goes from O(n^2) to roughly O(n).

The underlying fluid simulation is done using a eulerian approach. The algorithm is based on Mick West's "Practical Fluid Mechanics" for game development.

3 METHOD

The project runs in a simulation loop. In each loop iteration, the program first applies the external force, including grid-based fluid force and gravity, to calculate a tentative updated position. Then for each particle, the program finalizes its position through the incompressibility constraint, and re-bin the particle if it has moved out of its original bin. Then the program renders the particles and the user interface.

3.1 Particle System Simulation

3.1.1 Incompressbility Constraint

As described in Position Based Fluids, the system has one constraint for density per particle, which is defined as follows for particle *pi*:

$$\frac{\rho i}{\rho 0} - Ci(p1...pn) = 1 \quad (1)$$

p1...pn are the neighbor particles of *pi*. ρi is the SPH density estimator:

$$\rho i = \sum_{j=1}^{n} W(pi - pj, h) \quad (2)$$

W is the weight function and *h* is particle radius. We use the poly6 kernel as our weight function:

$$Wpoly6(r,h) = \frac{315}{64\pi h^9}(h^2 - |r|^2)^3 \quad (3)$$

For every frame, our goal is to find a Δp for each particle such that $C(p + \Delta p) = 0$. Δp is found by a series of Newton steps along the constraint gradient:

$$\Delta p \approx \nabla C(p)\lambda \quad (4)$$

We need to solve for particle density constraint λ in order to find Δp, which is found by

$$\lambda i = -\frac{Ci(p1,\ldots,pn)}{\sum k |\nabla pkCi|^2+\varepsilon} \quad (5)$$

ε in the denominator is a small relaxation parameter. Then we use Δp to update the particle location for pi. The equation to update Δp is:

$$\Delta pi = \frac{1}{\rho 0}\sum_{j=1}^{n}(\lambda i + \lambda j)\nabla W(pi - pj,\ h) \quad (6)$$

To calculate ∇W, we use Spiky kernel:

$$\nabla W spiky(r,h) = \frac{45}{\pi h^6}(h-|r|)^2\frac{r}{|r|} \quad (7)$$

3.1.2 Tensile Instability

In fluid-like particle simulation, a common problem is clustering, which is caused by a negative pressure when a particle is unable to satisfy the rest density constraint. We add a s term as a smoothing kernel to mitigate this effect:

$$s_{corr} = -k\left(\frac{W(pi-pj,h)}{W(\Delta q,h)}\right)^n \quad (8)$$

Δq is a point some fixed distance inside the smoothing kernel radius and k is a small positive constant. According to Macklin et. al, $|\Delta q| = 0.1h,\cdots,0.3h$, k = 0.1 and n = 4 worked well.

$$\Delta pi = \frac{1}{\rho 0}\sum_{j=1}^{n}(\lambda i + \lambda j + s_{corr})\nabla W(pi - pj,\ h) \quad (9)$$

This prevents the pressure from being a negative value, with the tradeoff of lower surface tension strength.

3.1.3 Vorticity and Viscosity

To account for vorticity, our program uses an additional external force used to prevent the particle from undesirable additional damping force. The equation for vorticity is:

$$\omega = \nabla\times v = \sum_{j=1}^{n} v_{ij}\times\nabla pj W(pi-pj,h) \quad (10)$$

v_{ij} is vj - vi. The equation for external force is:

$$f_i^{vorticity} = \varepsilon(N\times\omega) \quad (11)$$

Where $N = \frac{\eta}{|\eta|}$ with $\eta = \nabla|\omega|$.

To achieve coherent motion of the fluid, we also apply XSPH viscosity to each particle, an additional velocity to each particle based on its neighbor's velocity and position:

$$v_i^{new} = v_i + c\sum_{j=1}^{n} v_{ij}\cdot W(pi-pj,h) \quad (12)$$

3.2 Grid-based Fluid Simulation

The simulation uses advection on a grid-based model (also known as an Eulerian Model) to simulate the underlying fluid ("air") in the game. We use a simplified version of the incompressible Navier Stokes equations that ignores viscosity and no external forces:

$$\frac{\partial u}{\partial t} + \frac{1}{\rho}\nabla p = 0$$

$$\nabla\cdot u = 0$$

Where p is the pressure, ρ is the density, and u is velocity. To simplify the math, we assume density and pressure are equivalent for our fluid. To calculate the equations, we decided to use explicit Euler integration for its speed and ease of implementation.

3.2.1 Grid-based Fluid Implementation

The game space is divided into equally sized square cells, and each cell contains three fields: pressure, x velocity, and y velocity.

At each step in the simulation, each of these fields is advected by the velocity at the cell. That is, for each cell, the value in that cell is moved based on the velocity, and the value is distributed to the surrounding cells based on proximity. This is known as forward advection. If any of the surrounding cells is not within the bounding box of the game, the advected value is distributed within the nearest cells that are in bounds. This allows for realistic behavior when interacting with the "walls" of the game.

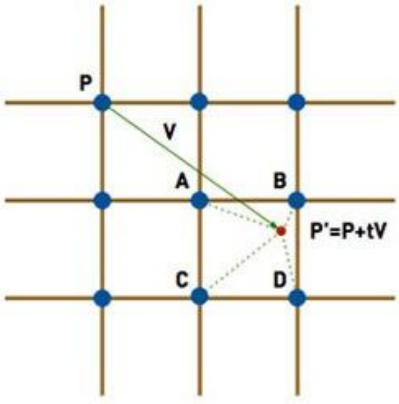

Figure 1: Forward Advection

In our program, first, the pressure is advected, and then the velocities are advected. However, forward advection alone does not simulate an incompressible fluid. We chose to implement pressure as a simple and fast way to simulate this. Based on the difference in pressure between neighboring cells, the velocity of those two cells is changed to simulate the flow of the fluid from areas of high pressure to low pressure. This ideally causes the simulation to tend towards states where pressure is equal across all cells.

In order to compensate for the inherent instability of explicit Euler integration, we decided to repeat this process multiple times each simulation cycle at a very small time step. The result still computes in real time and allows for realistic motion.

In summary, for each simulation cycle:

---

calculate dt
for i in range(fluid_steps):
    advect pressure to temporary buffer
    advect velocities to temporary buffer
    solve for pressure forces
    update pressure and velocity from buffer

3.2.2 Fluid Force on Particles

The forces applied to each particle by the fluid is calculated by a simple drag force

$$F = Cv^2$$

Where C is an arbitrarily set drag coefficient, and v is the difference in velocity between the particle and the fluid velocity at the particle position. In order to sample the fluid velocity at the particle position, bilinear interpolation is used. This force is applied to each moveable particle at every frame of the simulation.

3.3 Binning Optimization on Finding Neighbor

The optimization divides the original 2-d space into grids, each of size cutoff*cutoff. Here, we set the cutoff size to be slightly larger than the diameter of the particle to ensure inclusion. In each grid, we have a "bin", represented by a list, to store all particles that are in the grid in any arbitrary timestamp. Using the 2d grid, for each particle in a given time, we only need to check the particles that are in the neighboring bins, which reduced our inner loop runtime from O(n) to O(9d), where d is the density of each grid. Since the density is a predefined constant number, the overall runtime decreased from O(n^2) to O(9dn), which is roughly linear time O(n). Fig. 2 is a visualization of the binning algorithm.

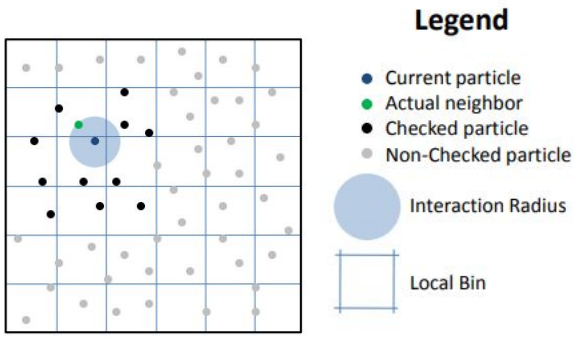

Figure 2: Binning visualization for a single particle

4 Results

4.1 Binning Optimization

Since OpenGL capped its framerate to 75, we tested the number of particles in the scene when the framerate starts to decrease for both the unoptimized version and the optimized version:

Framerate (Capped at 75)	Unoptimized	With Binning
70	170	1500
60	185	1550
45	215	1650
30	260	1750

Table 1: The number of particles that the program has at the corresponding framerate.

The result has shown a significant improvement upon the implementation of binning. Through this trial, we learned that binning is very effective to achieve a large scale real-time particle simulation.

4.2 Particle System

4.2.1 Rest Density

We set up a trial to test the effect of rest density on the fluid's behavior. For each rest density configuration, we keep adding particles from the top left corner until it reaches 300 particles, and save the stable state of the liquid as the final result:

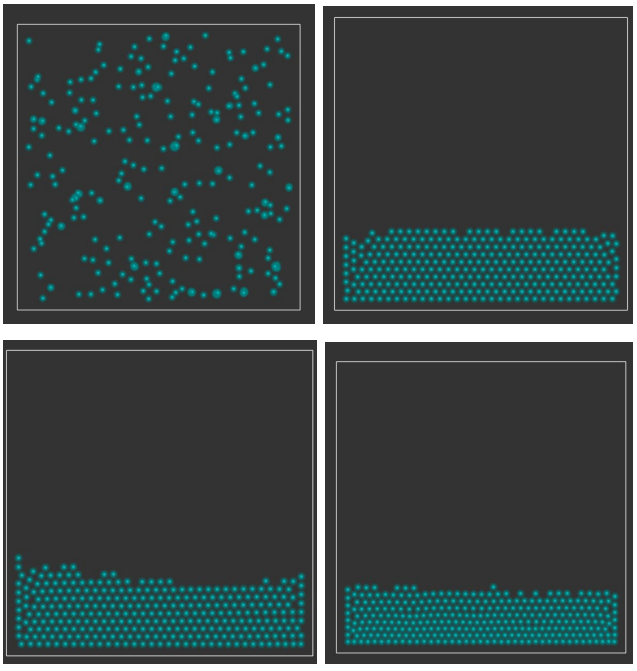

Figure 3(left to right):

3a. Rest density = 100; 3b. Rest density =300;

3c. Rest density = 600; 3d. Rest density = 5000.

As shown in the figure, the particles may never reach a stable state with a rest density that is too low, which implies that the accumulated force acting on the particles gradually increases over time. When the rest density is 300, the particle reaches a liquid-like state, where there will be waves when a particle is dropped into the system, but the system will eventually reach a stable state. When the rest density is 600, dropping a particle will not cause a wave in the system since there is not enough force acting in between the particles, and more particles accumulate on the left side, where we dropped the particles. When the rest density is 5000, the inter-particle force is too low, the bottom particles cannot repel the upper ones, which caused a higher density on the bottom part of the system and lower density on the top part. To properly simulate fluid, we choose the rest density to be 300.

4.2.2 Inter-Particle Interaction

We currently have two different types of particles: rock and water. Rock is not movable and acts as a solid obstacle; while water is the fluid-like particle. Fig. 4 is a snapshot of how they interact with each other.

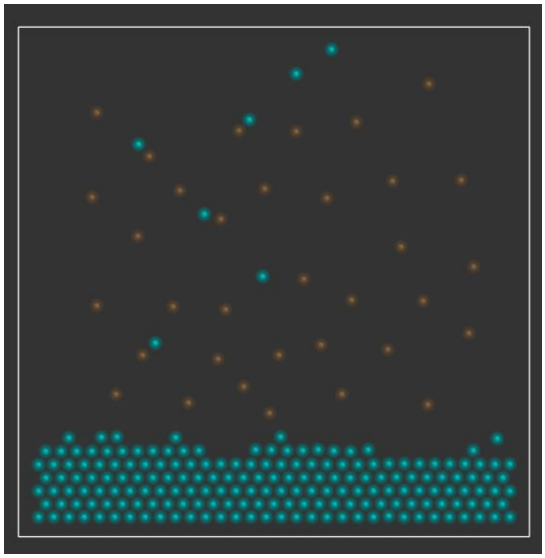

Figure 4. The brown particle represents rock; The blue particle represents water.

Only the rocks exert force on the water particles since the rocks are unmovable.

4.3 Grid-based Fluid Simulation

Although the grid-based fluid model makes many simplifying assumptions, it allows for realistic seeming behavior rendered in real-time. The fluid accelerates particles as expected and tends towards equilibrium with equal pressure throughout the fluid. This met our goal for this application, as we aimed to create an interactive game.

4.4 User Interface and Interaction

The GUI shows the in-game statistics on the top left corner, which is mainly used for performance evaluation purposes.

For user interaction, we have implemented a mouse click control for particle generation. On top of that, the user could hold the mouse left button for continuous generation of particles.

To select a type of particle, we ask the user to press the corresponding key on the keyboard for the desired particle. The selected particle is indicated on the GUI as highlighted.

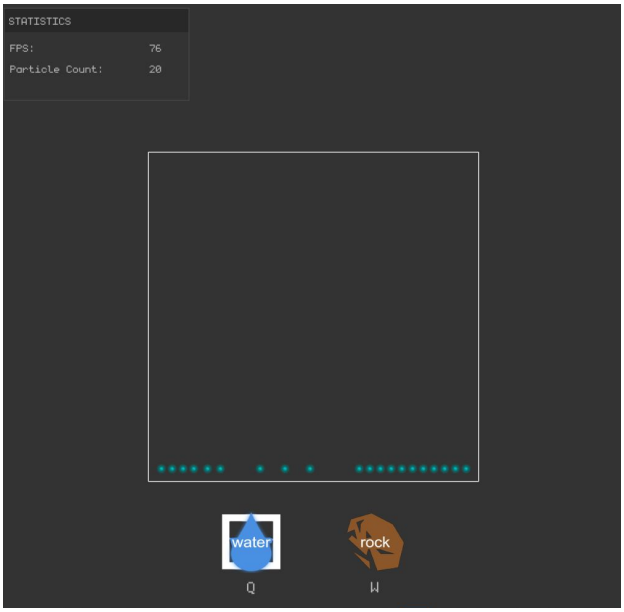

Figure 5. The GUI overview: The top left corner shows the statistics such as FPS and the particle count; the highlighted water icon indicates that the water particle is currently being selected.

5 Conclusion and Future Steps

In this project, we implemented a real-time 2D particle simulation game that supports thousands of particles to be rendered at the same time. To simulate fluid, we used the Smoothed Particle Hydrodynamics method and Grid-Based Fluid method. We implemented binning which effectively reduces the runtime in each game loop. We also added GUI to enable more intuitive control for the project.

For the future, we would like to add more types of particles and implement a real-time rendered fluid surface for better visual effect. We would also consider adding more audio effects and polish our GUI.

6 References

LiquidFun, google.github.io/liquidfun/.

Macklin, Miles, and Matthias Muller. *Position Based Fluids*. mmacklin.com/pbf_sig_preprint.pdf.

Oat, Christopher. *Efficient Spatial Binning on the GPU*. www.chrisoat.com/papers/EfficientSpatialBinning.pdf.

"Practical Fluid Mechanics." *Cowboy Programming*, cowboyprogramming.com/2008/04/01/practical-fluid-mechanics/.

7 Team Member Contribution

Roy Guo: GUI, Position based fluid

Christine Sheng: Grid-based fluid

Mingcheng Zhu: benchmark, binning optimization