Graphics Final: Position Based Fluids

Ty Trusty

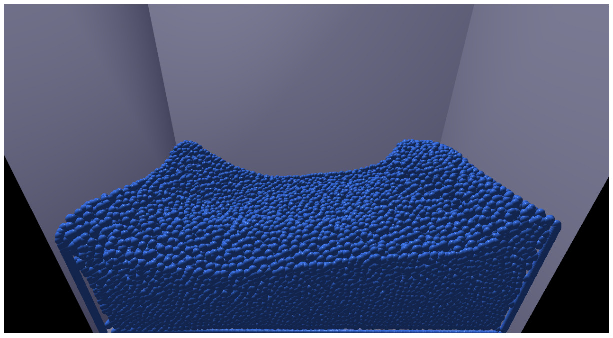

Fig. 1: Example 20k particle simulation

I. Introduction

For my final project, I implemented a particle-based fluid simulation based on the paper "Position Based Fluids" [1]. My goal for the project was to create a realistic, real-time 3D fluid simulation. This report consists of two parts: an overview of the fluid simulation algorithm and the relevant equations, and a description of the implementation details and issues.

II. Algorithm Overview

A. Simulation Step

```
1:  for all particles $i$ do
2:      - apply external forcex $v_i \leftarrow v_i + \Delta t \mathbf{f_{ext}}$
3:      - integrate $x_i^* \leftarrow x_i + \Delta t v_i$
4:      - find particle neighbors
5:      - calculate $\lambda_i$
6:  end for
7:  for all particles $i$ do
8:      - calculate $\Delta p_i$
9:      - update position $x_i^* \leftarrow x_i^* + \Delta p_i$
10:     - handle boundary collisions
11:     - apply XSPH viscosity
12:     - update velocity $v_i \leftarrow \frac{1}{\Delta t}(x_i^* - x_i)$
13:     - update position $x_i \leftarrow x_i^*$
14: end for
```

III. Equations

A. Density Constraint

In order for the fluid to behave properly, it must satisfy the incompressibility condition. This means that the density of an individual fluid particle is always constant. This is a necessary condition for providing a good approximation of fluid flow. To satisfy this condition, the following constraint is used:

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

where

$$\rho_i = \sum_j m_j W(p_i - p_j, h) \tag{2}$$

In these equations, for each particle i we have a constraint C_i on each particle, a position p_i, mass m_i, and neighbors j. ρ_i is an estimate of a particle's density and ρ_0 is the rest density constant. The function W is the SPH kernel function which is described in the following section. With our density constraint function, we can find an update to the a particle's position Δp by satisfying the following constraint:

$$C(p + \Delta p) = 0 \tag{3}$$

This is solved by finding a position update that moves along the gradient of the constraint function. The constraint can then be written as

$$C(p + \Delta p) \approx C(p) + \nabla C^T \nabla C \lambda + \epsilon\lambda \tag{4}$$

$$\Delta p \approx \nabla C \lambda \tag{5}$$

The extra term $\epsilon\lambda$ is referred to as the "constraint force mixing" [6] term and is used to regularize the constraint. The Lagrange multiplier λ_i is computed by

$$\lambda_i = -\frac{C_i(p_1, ..., p_n)}{\sum_k |\nabla_{p_k} C_i|^2 + \epsilon} \tag{6}$$

The gradient of the constraint for a particle i with respect to a particle k's position is

$$\nabla_{p_k} C_i = \frac{1}{\rho_0} \begin{cases} \sum_j \nabla_{p_k} W(p_i - p_j, h) : \text{if } k = i \\ -\nabla_{p_k} W(p_i - p_j, h) \quad : \text{if } k = j \end{cases} \tag{7}$$

B. Density Kernels

SPH kernel functions allow us to create a smooth scalar weighting function from discrete differences between particle positions. In the following kernel equations, r is the distance between two particles i and j, and h defines the smoothing radius that dictates how fast the kernel function "falls off" as the distance between two particles increases. The kernel functions used were:

$$\text{Poly6 kernel: } W_{ij} = \frac{315}{64\pi h^9}(h^2 - r^2)^3 \tag{8}$$

$$\text{Spiky kernel: } \nabla W_{ij} = -\frac{45}{\pi h^6}(h - r)^2 \tag{9}$$

C. Viscosity

This viscosity term is referred to as XSPH viscosity [5], and is essential for a fluid-like flow. The scalar c is a constant that sets the strength of the viscosity update. A higher c will make the fluid more viscous.

$$v_i = v_i + c \sum_j v_{ij} \cdot W(p_i - p_j, h) \tag{10}$$

D. Tensile Instability

A problem that arises in particle-based fluid simulations is densities being too small due to particles having too few neighbors, which results in particle clustering. A solution to this is to add an artificial pressure term [4] that effectively provides surface tension. In this equation, k is the artificial pressure strength and $|\Delta q|$ is some constant distance from the particle.

$$s_{corr} = -k \frac{W(p_i - p_j, h)}{W(\Delta q, h)}^n \tag{11}$$

With the addition of the artifical pressure term, we can write the position update as

$$\Delta p_i = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j + s_{corr}) \nabla W(p_i - p_j, h) \tag{12}$$

IV. Rendering & Implementation

A. Neighbor Finding with Hash Grid

The algorithm described in the report is heavily dependent on a fast neighbor searching algorithm. For each simulation step, a particle's nearest neighbors must be found so that density can be computed. For this reason, it is critical that neighbor searching be efficient. For my implementation, I chose to use spatial hashing. In this technique positions in 3D are hashed into a 1D hash table where similar positions are placed in the same or "nearby" bins. The hash grid is essentially a map between keys and arrays of particles. A key is a tuple (x, y, z) which can be found by a hash function that computes $key = (floor(p_i.x/width), floor(p_i.y/width, floor(p_i.z/width))$ where $width$ is the size of a cell in the hash grid. With this, all the neighbors of a particle can be found by first hashing its position, then computing the distance between all other particles in its current bin, and nearby bins (depending on kernel size).

B. Collision Handling

I handled basic boundary collision by simply moving particles inside the box if they exit the bounds. I attempted using a spring penalty force but this produced unstable results.

C. Particle Rendering

To render individual particles, I used a technique call imposter rendering. Rendering the geometry of individual particles for large scenes would be too computationally expensive. In this technique, instead a square billboard is rendered for each particle, and the corresponding point on the sphere can be determined by the current UV position on the billboard. The results of this can be seen in Figure [1].

D. Surface Smoothing & Rendering

To generate a smooth fluid surface, my approach was to capture the depth map from the particle rendering, smooth this depth map, and then use the smoothed depth map to render the final fluid surface. To capture the depth map, I use an FBO to save the depth buffer from the particle rendering to a texture. With the depth buffer texture, I feed this texture to a shader where filtering is performed.

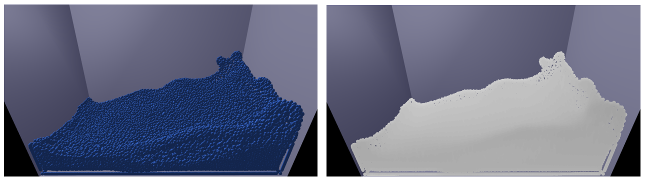
(a) Particle rendering (b) Depth map

Fig. 2: Particle rendering and the corresponding depth map

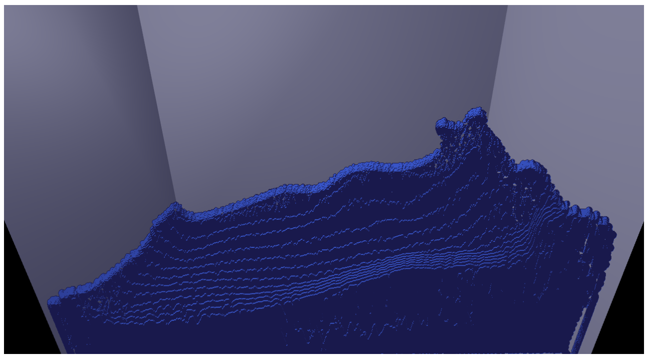
Fig. 3: Surface extracted from smoothed depth map

To filter the depth buffer, I apply a bilateral filter to blur the depth values. The bilateral filter functions as a gaussian blur that preserves edges by accounting for distances between pixels and radiometric differences (i.e. depth distances in this case). As you can see in Figure [3], the fluid surface's boundaries are not smoothed out. However, there are some smoothing artifacts in the center of the fluid (long streaks along surface). From this smoothed depth buffer, I next generate per-pixel normals by approximating gradient directions using neighboring pixels depth differences and then using these gradient directions to compute a view-oriented normal. Finally, with the normal and depth textures I render the final fluid surface.

V. Issues

The primary issue with my implementation is that the resulting fluid surface is not properly smoothed. One problem is that a simple bilateral filter has a number of artifacts, such as the streaks along the surface. On top of this, my scheme for generating normals seems to be flawed as well. I tried a number of methods but was unable to produce a reasonably good normal map. Finally another area for improvement could be performance. I parallelized the simulation step with OpenMP, but I could achieve much better performance by executing the entire simulation loop on the GPU.

References

[1] Miles Macklin and Matthias Mller. 2013. Position based fluids. ACM Trans. Graph. 32, 4, Article 104 (July 2013), 12 pages. DOI: https://doi.org/10.1145/2461912.2461984

[2] Erin J. Hastings and Jaruwan Mesit and Ratan K. Guha. Optimization of Large-Scale, Real-Time Simulations by Spatial Hashing
[3] Spatial Hashing in C++. http://www.sgh1.net/posts/spatial-hashing-1.md
[4] Monaghan, J. J. 2000. Sph without a tensile instability. J. Comput. Phys. 159, 2 (Apr.), 290311.
[5] Schechter, H., and Bridson, R. 2012. Ghost sph for animating water. ACM Trans. Graph. 31, 4 (July), 61:161:8.
[6] Smith, R. 2006. Open dynamics engine v0.5 user guide.
[7] OpenGL Tutorials, Chapter 13. Lies and Imposters https://paroj.github.io/gltut/Illumination/Tutorial%2013.html
[8] Robert Bridson, Ronald Fedkiw, and Matthias Mller-Fischer. 2006. Fluid simulation: SIGGRAPH 2006 course notes