2D Fluid Dynamics and Density Simulator
Matthew Ishii 627-1726

1. Introduction

My project was to create a 2D particle based fluid system that simulated surface tension, viscoelasticity, and density of fluids. I also planned to add squares to the scene that could handle displacement, and density, but I wasn't able to implement that in time. I also wasn't able to add in a vector based visual depiction of the velocity of objects, or a density toggle. The particles do become whiter as their velocity increases though. Particle smoothing was also approximated by rendering the particles as larger than their given area on screen.

My primary reference for the project was Particle-based viscoelastic fluid simulation by Clavet et al (1). I used OpenGl, and SDL2 for the Window and input handling. My code was made to reflect the basic algorithms outlined in the paper.

Demo videos can be viewed here:
https://drive.google.com/folderview?id=0B8UD2XsZNvjdfjhMSG1Db1lUelpyT1BJM3RGYjAwWkpFU041TlgxOFhqc04zTTg5T3FOMDQ&usp=sharing

2. General Implementation Details

Section 7.1 of the paper, "Neighbor Search" outlines how one can find neighboring particles. The scene is broken down into a 2D grid, with each unit being the size of a particle. If particles overlap in a square, then a reference to the first particle is stored in the grid, and the second is stored in a reference within the first, and so on.

A list of neighboring particles is then gathered. The paper suggests implementing springs that act as a force on pairs of particles, which are implemented as impulses on particles.

SDL is used for the window and handling input events.

Structs were used to organize concepts used in the demo, and maintain public access to them. Structs made for this implementation are:
Vec2: Stores 2 floats, for locations in world space, though I don't use them for particles, because of difficulties I had using pointers to access variables within structs within structs.
Vec4: Used to store colors
Source: Sources generate particles on runtime. A Source is also used when the user generates particles. They store details on the orientation and speed of new particles, as well as their particle type.
Particles: These store position, velocity, density, pressure, and mass.

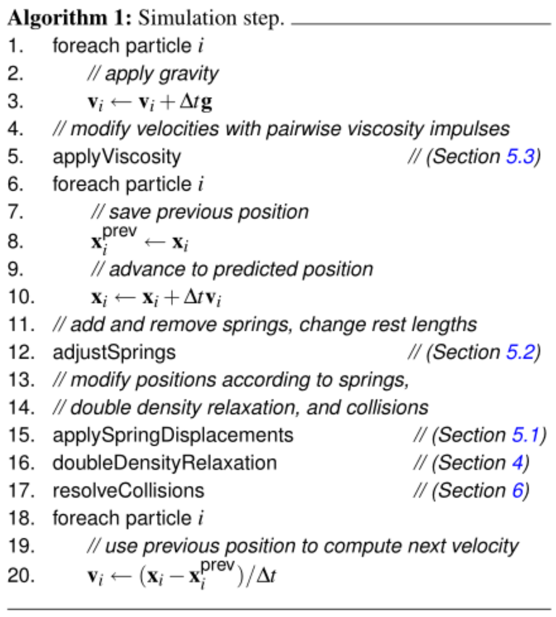
Springs: These store neighbor particles for the spring operations as well as every operation between particles. The variable **Lij** stores the value of the spring used in spring displacements.
ParticleType: Stores mass and color for a particle.
Boundary: Outlines the bounds of the window, to detect particles moving outside the window as collisions, and repel them.

3. Input

On running the demo, these are the options for input:

1) Arrow keys alter the orientation of gravity to their respective directions
2) Space cancels gravity
3) Left click generates particles at the source of the click. The scene is limited to 6000 particles.
4) Numpad ‘+’ and ‘-’ alter the material type. There are 3 materials, red, green, and blue, in ascending order of mass. ‘+’ moves up in mass, ‘-’ moves down. (Not sure if you’ll be able to see the green, but green isn’t used except as an option for a ‘middle weight’ fluid. It’s also the mouse’s default fluid).

4. Algorithm implementation

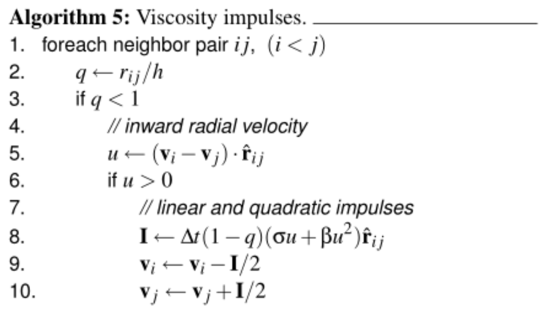


Algorithm 1: Simulation step.

```
foreach particle i
    // apply gravity
    $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t\mathbf{g}$
// modify velocities with pairwise viscosity impulses
applyViscosity                          // (Section 5.3)
foreach particle i
    // save previous position
    $\mathbf{x}_i^{\text{prev}} \leftarrow \mathbf{x}_i$
    // advance to predicted position
    $\mathbf{x}_i \leftarrow \mathbf{x}_i + \Delta t\mathbf{v}_i$
// add and remove springs, change rest lengths
adjustSprings                           // (Section 5.2)
// modify positions according to springs,
// double density relaxation, and collisions
applySpringDisplacements                // (Section 5.1)
doubleDensityRelaxation                 // (Section 4)
resolveCollisions                       // (Section 6)
foreach particle i
    // use previous position to compute next velocity
    $\mathbf{v}_i \leftarrow (\mathbf{x}_i - \mathbf{x}_i^{\text{prev}})/\Delta t$
```

I’ll break down how the code is structured, and my success and failure in implementing each feature.

Applying gravity and advancing positions and velocities are simple, so I will not go into detail on them.



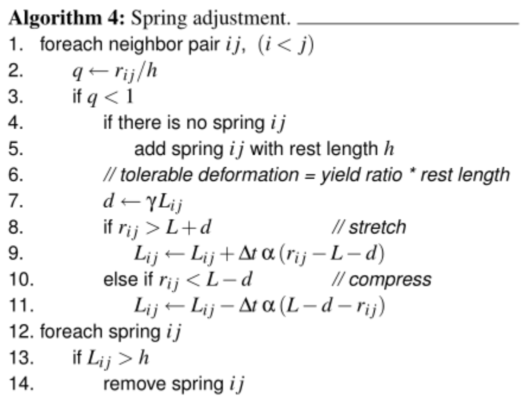
Algorithm 5: Viscosity impulses.

```
1.  foreach neighbor pair $ij$, $(i < j)$
2.      $q \leftarrow r_{ij}/h$
3.      if $q < 1$
4.          // inward radial velocity
5.          $u \leftarrow (\mathbf{v}_i - \mathbf{v}_j) \cdot \hat{\mathbf{r}}_{ij}$
6.          if $u > 0$
7.              // linear and quadratic impulses
8.              $\mathbf{I} \leftarrow \Delta t(1-q)(\sigma u + \beta u^2)\hat{\mathbf{r}}_{ij}$
9.              $\mathbf{v}_i \leftarrow \mathbf{v}_i - \mathbf{I}/2$
10.             $\mathbf{v}_j \leftarrow \mathbf{v}_j + \mathbf{I}/2$
```

1) applyViscosity()

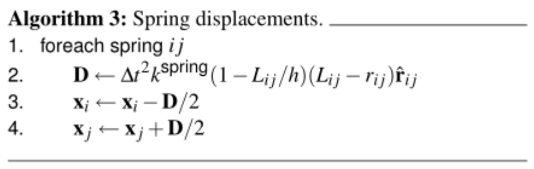
The σ value was set as 0.5 in order to increase viscosity, and let the effects of surface tension be more relevant.

I wasn't able to figure out how to access the neighbor *j* and apply an opposing impulse on its vector at the same time as particle *i.* However, that particle should behave appropriately because eventually the loop will cycle to it as particle *i*, and perform the appropriate operations then.



Algorithm 4: Spring adjustment.

```
1.  foreach neighbor pair $ij$, $(i < j)$
2.      $q \leftarrow r_{ij}/h$
3.      if $q < 1$
4.          if there is no spring $ij$
5.              add spring $ij$ with rest length $h$
6.          // tolerable deformation = yield ratio * rest length
7.          $d \leftarrow \gamma L_{ij}$
8.          if $r_{ij} > L + d$              // stretch
9.              $L_{ij} \leftarrow L_{ij} + \Delta t\, \alpha\, (r_{ij} - L - d)$
10.         else if $r_{ij} < L - d$         // compress
11.             $L_{ij} \leftarrow L_{ij} - \Delta t\, \alpha\, (L - d - r_{ij})$
12. foreach spring $ij$
13.     if $L_{ij} > h$
14.         remove spring $ij$
```

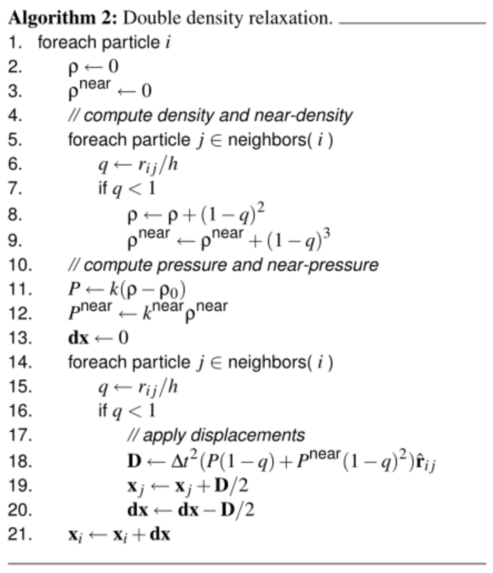


Algorithm 3: Spring displacements.

```
1.  foreach spring $ij$
2.      $\mathbf{D} \leftarrow \Delta t^2 k^{\text{spring}}(1 - L_{ij}/h)(L_{ij} - r_{ij})\hat{\mathbf{r}}_{ij}$
3.      $\mathbf{x}_i \leftarrow \mathbf{x}_i - \mathbf{D}/2$
4.      $\mathbf{x}_j \leftarrow \mathbf{x}_j + \mathbf{D}/2$
```

2) adjustSprings()

This combines both algorithms for spring adjustment and displacement. The main change made was that spring displacements occur in the loop when the value is assigned. Because the set of neighbors is reset every loop, the springs do not need to be removed manually.



Algorithm 2: Double density relaxation.

```
1.  foreach particle $i$
2.      $\rho \leftarrow 0$
3.      $\rho^{\text{near}} \leftarrow 0$
4.      // compute density and near-density
5.      foreach particle $j \in$ neighbors( $i$ )
6.          $q \leftarrow r_{ij}/h$
7.          if $q < 1$
8.              $\rho \leftarrow \rho + (1-q)^2$
9.              $\rho^{\text{near}} \leftarrow \rho^{\text{near}} + (1-q)^3$
10.     // compute pressure and near-pressure
11.     $P \leftarrow k(\rho - \rho_0)$
12.     $P^{\text{near}} \leftarrow k^{\text{near}} \rho^{\text{near}}$
13.     $\mathbf{dx} \leftarrow 0$
14.     foreach particle $j \in$ neighbors( $i$ )
15.         $q \leftarrow r_{ij}/h$
16.         if $q < 1$
17.             // apply displacements
18.             $\mathbf{D} \leftarrow \Delta t^2 (P(1-q) + P^{\text{near}}(1-q)^2)\hat{\mathbf{r}}_{ij}$
19.             $\mathbf{x}_j \leftarrow \mathbf{x}_j + \mathbf{D}/2$
20.             $\mathbf{dx} \leftarrow \mathbf{dx} - \mathbf{D}/2$
21.     $\mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{dx}$
```

3) doubleDensityRelaxation()

I couldn't get this to work as the paper suggested. The displacement wasn't working properly. I used an implementation of a different SPH formula (2) to add this and add mass to liquids. The differences this creates in the algorithm are:

Differences:

8: $\rho \leftarrow \rho$ +(1−q)^2 becomes $\rho \leftarrow \rho$ +(1−q)^3(mass*constant)

9: ρ near← ρ near+(1−q)^3 becomes

ρ near← ρ near+(1−q)^4(mass*constant)

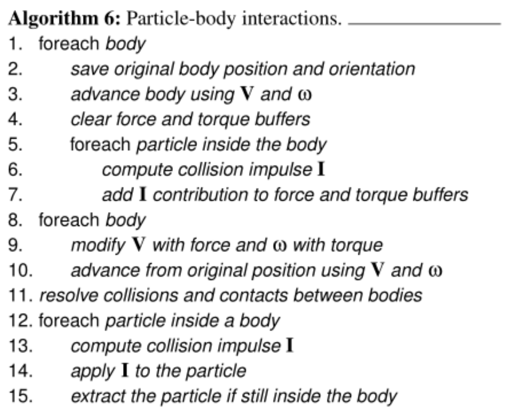
11: P←k($\rho - \rho$ 0) becomes P←k($\rho - \rho$ 0*mass)

18: D←Δt2(P(1−q)+Pnear(1−q)^2)ˆrij becomes

D←Δt2(*k*(Pi+Pj)(1−q)^3) + k(Pinear+Pjnear)(1−q)^2) * rij

20: dx←dx−D/2 becomes dx←dx−D/(mass*r)

Some lines that allow for separate surface tensions between materials are also added



Algorithm 6: Particle-body interactions.

```
1.  foreach body
2.      save original body position and orientation
3.      advance body using $\mathbf{V}$ and $\omega$
4.      clear force and torque buffers
5.      foreach particle inside the body
6.          compute collision impulse $\mathbf{I}$
7.          add $\mathbf{I}$ contribution to force and torque buffers
8.  foreach body
9.      modify $\mathbf{V}$ with force and $\omega$ with torque
10.     advance from original position using $\mathbf{V}$ and $\omega$
11. resolve collisions and contacts between bodies
12. foreach particle inside a body
13.     compute collision impulse $\mathbf{I}$
14.     apply $\mathbf{I}$ to the particle
15.     extract the particle if still inside the body
```

4) resolveCollisions()

I wasn't able to properly implement this either. The version that exists in the code is just a boundary check, to repel the particles if they go over the boundary.

References

(1) Particle-based fluid simulation for interactive applications - http://dl.acm.org/citation.cfm?id=846298

(2) Reference for this paper https://imdoingitwrong.wordpress.com/2010/12/14/why-my-fluids-dont-flow/

(3)Particle-based viscoelastic fluid simulation - http://dl.acm.org/citation.cfm?id=1073400

(4)Reconstructing surfaces of particle-based fluids using anisotropic kernels - http://dl.acm.org/citation.cfm?id=2421641