Fluid Simulation Technical Report

(Eric Knapik)

This is a fluid simulation based on the Navier-Stokes equations for the movement of a fluid over time. The rendering of this was initially ray ray traced but as the number of fluid particles increased the render time for each frame increased dramatically causing me to switch to a rasterized OpenGL version of the project. The desired goal was to get a real time realistic fluid simulation. The code can be found at: https://github.com/EKnapik/Simple_Fluid_Sim

Approach:

For a fluid simulation I used the idea of incompressible individual particles that would represent my 'fluid'. Each particle would then be governed by the Navier-Stokes equations to determine the density and pressure at each timestep then update each particle given the current forces on my fluid. To improve the calculation of each timestep I used the smooth particle hydrodynamics approach, utilizing smoothing kernels for the approximation of the Navier-Stokes solution.

User Guide:

The program was made using XCode and was compiled using the XCode version of g++. For modifying the fluid simulation the definitions for the fluid can be changed within the FluidParticle.hpp file. This is a very static program will only show the demonstration of the current compiled fluid.

Changing the Program:

This program supports any version of fluid given the proper definitions. To change the timestep for each frame the #define within main.cpp can be changed to create any desired time iteration.
Within the FluidParticle.hpp file several #defines can be changed to create any possible fluid that can be described.
#define FLUID_NUM_PARTICLES (The number of particles, must be cubic number 1, 8, 27...)
#define FLUID_PARTICLE_MASS (The individual mass of each particle)
#define FLUID_CONSTANT_K (The fluid constant applied to the density)
#define FLUID_FRICTION_MU (The viscosity of the fluid)
#define FLUID_H_VALUE (The Checking distance for the SPH smoothing kernels)
#define FLUID_RADIUS (The individual particle's radius)
#define FLUID_REST_DENSITY (The intrinsic fluid density)

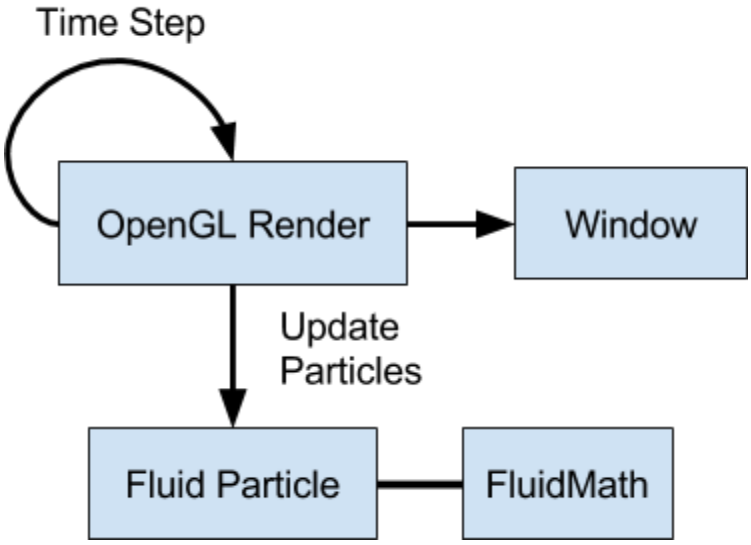
Running:

Once everything is setup and saved within the files. Compile the program and run. For my implementation I used XCode and the g++ compiler within MacOSX. This requires that the given system has OpenGL and GLUT installed on it; this is for the graphics

interface and the windowing system.

Technical Docs:

The Overview of the program operates on update loop described below.



There is a number of fluid particles that know how to update themselves given the particles in the scene and its intrinsic properties. Each particle is a sphere that is rendered to the scene using OpenGL. At every timestep all particles positions, densities, velocities and pressures are updated and changes according to the Navier-Stokes equations and Semi-Implicit Euler integration.

The entire program works to solve the Navier-Stokes equations for an arbitrary number of fluid particles.

Incompressible Navier-Stokes

- ρ - Density
- p - Pressure
- μ - Friction
- υ - Velocity

$$\rho_i \frac{dv_i}{dt} = - \bigtriangledown p_i + \mu \bigtriangledown^2 \cdot v_i + f_{external}$$

I use Smooth Particle Hydrodynamics to solve the above equations using smoothing kernels.

New Equations

$$\rho_i \approx \sum_j m_j \frac{315}{64\pi h^9} (h^2 - \|r - r_b\|^2)^3$$

$$\frac{\nabla p_i}{\rho_i} \approx \sum_j m_j (\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}) \frac{-45}{\pi h^6} (h - \|r - r_b\|)^2 \frac{(r - r_b)}{\|r - r_b\|}$$

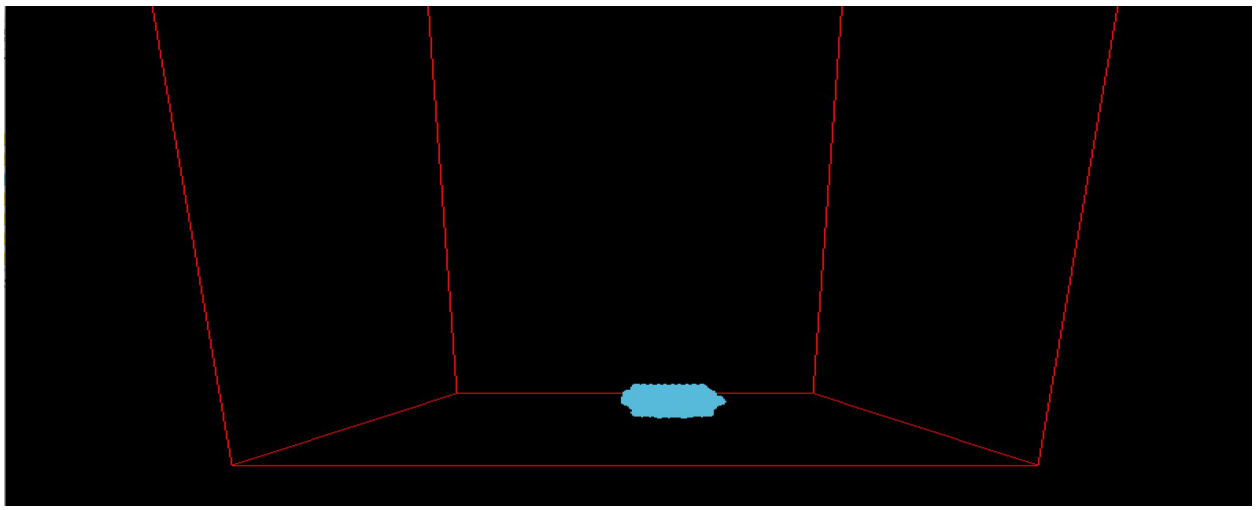
$$\frac{\mu}{\rho_i} \nabla^2 v_i \approx \frac{\mu}{\rho_i} \sum_j m_j (\frac{v_j - v_i}{\rho_j}) \frac{45}{\pi h^6} (h - \|r - r_b\|)$$

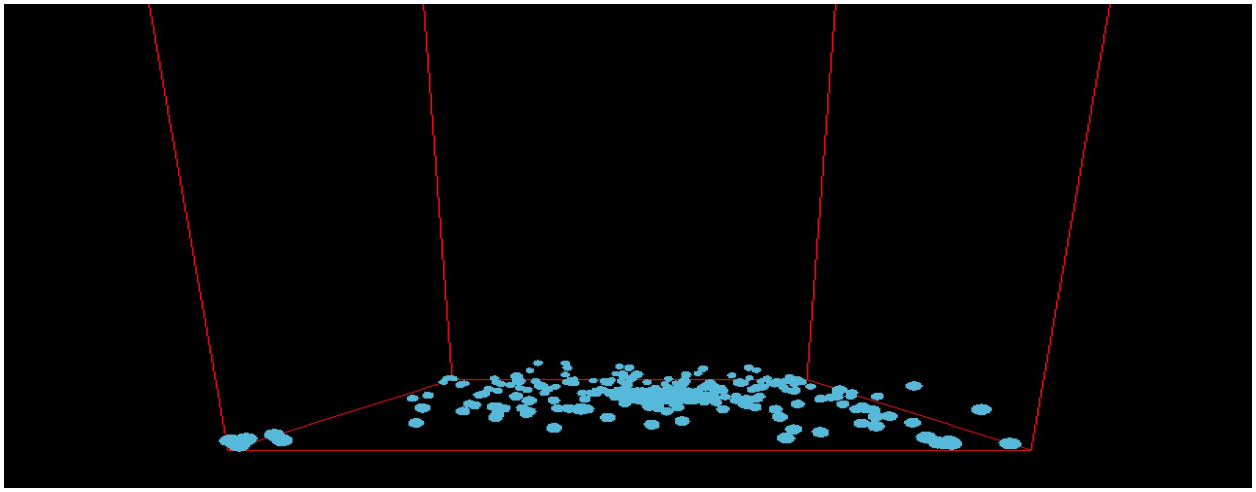
The program works by instantiating a single sphere that is re rendered to the scene at all the different fluid particle locations. This is done very simply but significantly cuts down on the required gpu object geometry and positional data transfers. Besides that the program works very simply to codize the above equations and apply them to each particle for each timestep. I utilize the dot product distance for determining if two particle will collide or influence

another's pressure, density or viscous abjection. The dot product distance is finding the squared length of the vector between any two points.

Results:

Some images from the ran simulation.





Future Enhancements:

One of the best enhancements would be to increase the speed at which the nearest 26 particles can be determined. I could not get it to work and used the dot product distance optimization; but, if the nearest 26 particles can be grabbed quickly this would be the largest optimization. Other optimizations would be easy fluid property changes such as on the fly density changes or making this a finalized software product instead of a single program.

Appendix of Code:

```
//
//  main.cpp
//  Simple_Fluid_Sim
//
//  Created by Eric Knapik
//  Copyright © 2016 EKnapik. All rights reserved.
//

#include <GLUT/GLUT.h>
#include <OpenGL/gl.h>
#include <iostream>
#include <unistd.h>
#include <time.h>
#include "Sphere.hpp"
#include "FluidParticle.hpp"
#include "BoundingBox.hpp"


// 16 x 9
#define WINDOW_HEIGHT 800
#define WINDOW_WIDTH 800
#define MICRO_FRAME_TIME 33333
#define TIME_DELTA 0.03333

// GLOBAL VARIABLES
FluidParticle *particle;
FluidParticle **particles;
int numParticles;
BoundingBox *box;
int waitTime;


// FUNCTIONS
void initOpenGL(void);
void render(void);
void initParticles(void);
void updateParticles(void);



int main(int argc, char * argv[]) {
    glutInit(&argc, argv);
```

```
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
    glutInitWindowSize(WINDOW_WIDTH, WINDOW_HEIGHT);
    glutCreateWindow("Simple Fluid Simulation");
    glutDisplayFunc(render);
    initOpenGL();

    // particle = new FluidParticle(glm::vec3(0.0, 0.0, 0.0));
    box = new BoundingBox();
    initParticles();

    waitTime = 0;
    glutMainLoop();
    return 0; // Cause GLUT allows us to even get back here..... (it doesn't)
}



void initOpenGL(void) {
    glEnable(GL_DEPTH_TEST | GL_CULL_FACE | GL_BLEND);
    glEnable(GL_TEXTURE_2D);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glClearColor( 0.0, 0.0, 0.0, 1.0 );
    glCullFace(GL_FRONT);
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

    return;
}



void render(void) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // poll events
    // bind and draw
    clock_t t;
    t = clock();
    float renderTime;
    // DO SOMETHING
    box->drawObj();
    //particle->drawObj();
    //particle->pos += glm::vec3(0.0, -0.1, 0.0);
    for(int i = 0; i < numParticles; i++) {
        particles[i]->drawObj();
    }
```

```
    updateParticles();


    t = clock() - t;
    renderTime = ((float)t)/CLOCKS_PER_SEC;
    //printf("Render took (%.4f seconds)\n", renderTime);

    //waitTime = MICRO_FRAME_TIME - t;
    //if(waitTime > 0) { usleep(waitTime); } // sleep for n microseconds

    glutSwapBuffers();
    // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!#!#!#!#!#!#!
    glutPostRedisplay();
}



void initParticles(void) {
    numParticles = FLUID_NUM_PARTICLES;
    particles = new FluidParticle *[numParticles];

    float currX = 0.0;
    float currY = 1.0;
    float currZ = 0.0;

    int tmp = powf(numParticles, float(1.0/3.0));
    int count = 0;
    for(int i = 0; i < tmp; i++) {
        currZ = 0.0;
        for(int j = 0; j < tmp; j++) {
            currX = 0.0;
            for(int w = 0; w < tmp; w++) {
                particles[count] = new FluidParticle(glm::vec3(currX, currY, currZ));
                count++;
                currX += (4*FLUID_RADIUS);
            }
            currZ -= (4*FLUID_RADIUS);
        }
        currY += (4*FLUID_RADIUS);
    }
}
```

```
void updateParticles(void) {
   // TODO  THERE SHOULD BE A PARTICLE ENGINEE....
   // Update the density and pressure of each particle
   for(int i = 0; i < numParticles; i++) {
      particles[i]->updateDensity(particles, numParticles);
      // printf("ID: %d, Density: %.2f\n", this->scene->particles[i]->id,
this->scene->particles[i]->density);
   }
   for(int i = 0; i < numParticles; i++) {
      particles[i]->updatePressure();
      // printf("ID: %d, Pressure: %.2f\n", this->scene->particles[i]->id,
this->scene->particles[i]->pressure);
   }
   // update the pressure over density term for each particle
   for(int i = 0; i < numParticles; i++) {
      particles[i]->updateGradPressureOverDensity(particles, numParticles);
   }
   // update the viscosity term for each particle
   for(int i = 0; i < numParticles; i++) {
      particles[i]->updateViscosityGradSquaredVelocity(particles, numParticles);
   }
   // Now move the particle
   for(int i = 0; i < numParticles; i++) {
      particles[i]->updateParticle(TIME_DELTA, particles, numParticles);
   }
   // -----------FLUID STUFF  END------
   // collide particles with the scene objects
   for(int i = 0; i < numParticles; i++) {
      particles[i]->boundsConstraint();
   }
   for(int i = 0; i < numParticles; i++) {
      particles[i]->collisionDetection(particles, numParticles, TIME_DELTA);
   }

}
```

```
//
//  FluidParticle.hpp
//  Simple_Fluid_Sim
//
//  Created by Eric Knapik on 5/7/16.
//  Copyright © 2016 EKnapik. All rights reserved.
//

#ifndef FluidParticle_hpp
#define FluidParticle_hpp

#include "Sphere.hpp"
#include "FluidMath.hpp"

// SETUP FOR WATER
/*
 The H value is very dependant on the number of particles in the simulation
 witht the value too low the kernel functions won't work properly and with it
 too high the fluid will explode
 */
// FUN NUMBERS  15x15x15: 3375   30x30x30: 27000
#define FLUID_NUM_PARTICLES 1000
#define FLUID_PARTICLE_MASS 0.02
#define FLUID_CONSTANT_K 3
#define FLUID_FRICTION_MU 3.5
#define FLUID_H_VALUE .045
#define FLUID_RADIUS 0.01685  // the particle's radius
#define FLUID_REST_DENSITY 998
#define MAX_DISTANCE 0.4

// Bounding Box
#define xMin -2.0
#define xMax 2.0
#define yMin 0.0
#define yMax 30.0
#define zMin -2.0
#define zMax 2.0

class FluidParticle: public Sphere {
public:
    FluidParticle(glm::vec3 pos);
    FluidParticle(glm::vec3 pos, float radius);
```

```
    void updateParticle(float timeStep, FluidParticle **fluidParticles, int numParticles);
    void updateDensity(FluidParticle **fluidParticles, int numParticles);
    void updatePressure();
    void updateGradPressureOverDensity(FluidParticle **fluidParticles, int numParticles);
    void updateViscosityGradSquaredVelocity(FluidParticle **fluidParticles, int numParticles);
    void collisionDetection(FluidParticle **fluidParticles, int numParticles, float timeStep);
    void collisionHandle(FluidParticle *particle);
    void collisionHandle(FluidParticle *particle, float distance);
    void boundsConstraint();

    // position covered by inheritance
    static int _id;
    int id;
    float mass;
    float restDensity; // THIS IS BASED ON THE totalMass / totalVolume of fluid
    float density;
    float pressure;
    glm::vec3 velocity = glm::vec3(0.0);
    glm::vec3 pressureTerm = glm::vec3(0.0);
    glm::vec3 viscosityTerm = glm::vec3(0.0);

};
#endif /* FluidParticle_hpp */


//
//  FluidParticle.cpp
//  Simple_Fluid_Sim
//
//  Created by Eric Knapik on 5/7/16.
//  Copyright © 2016 EKnapik. All rights reserved.
//

#include "FluidParticle.hpp"

// Initialize Static data
int FluidParticle::_id = 0;

FluidParticle::FluidParticle(glm::vec3 pos) : Sphere() {
    this->id = FluidParticle::_id++;
    this->pos = pos;
    this->radius = FLUID_RADIUS;
```

```
    this->density = 0.0;
    this->pressure = 0.0;
    this->mass = FLUID_PARTICLE_MASS;
    this->restDensity = FLUID_REST_DENSITY;
}

FluidParticle::FluidParticle(glm::vec3 pos, float radius) : Sphere() {
    this->id = FluidParticle::_id++;
    this->pos = pos;
    this->radius = radius;

    this->density = 0.0;
    this->pressure = 0.0;
    // MASS AND DENsity NEED TO BE CALCULATED
    this->mass = FLUID_PARTICLE_MASS;
    this->restDensity = FLUID_REST_DENSITY;
}

/*
 * This is applying the physics for this particle for this time delta
 * the current particle should not be in the array of particles given
 */
void FluidParticle::updateParticle(float timeStep, FluidParticle **fluidParticles, int
numParticles) {
    glm::vec3 gravity = glm::vec3(0.0, -9.8, 0.0);

    // solve for the change in velocity at this time according to Navier-Stokes
    glm::vec3 dvdt = gravity - this->pressureTerm + this->viscosityTerm;

    // printf("Acceleration: %.2f, %.2f, %.2f\n", dvdt.x, dvdt.y, dvdt.z);
    // Update with Semi-implicit Euler integration
    this->velocity += dvdt * timeStep;
    // printf("TimeStep: %.2f\n", timeStep);
    // printf("Velocity: %.2f, %.2f, %.2f\n", this->velocity.x, this->velocity.y, this->velocity.z);
    this->pos += this->velocity * timeStep;
    // printf("Position: %.2f, %.2f, %.2f\n", this->pos.x, this->pos.y, this->pos.z);
    // doing fluid fluid collision detection here
    collisionDetection(fluidParticles, numParticles, timeStep);
}

/*
 * the current particle should not be in the array of particles given
```

```
 */
void FluidParticle::updateDensity(FluidParticle **fluidParticles, int numParticles) {
    float density = 0.0;
    float dist;
    for(int i = 0; i < numParticles; i++) {
        if(this->id != fluidParticles[i]->id) {
            dist = glm::length(this->pos - fluidParticles[i]->pos);
            if(dist < MAX_DISTANCE) {
                density += fluidParticles[i]->mass * W(this->pos, fluidParticles[i]->pos,
FLUID_H_VALUE);
            }
        }
    }
    this->density = density;
}


void FluidParticle::updatePressure() {
    this->pressure = FLUID_CONSTANT_K * (this->density - this->restDensity);
}


/* The gradient pressure of a particle divided by the density of that particle
 * This is an estimation method using smoothing kernals
 * the current particle should not be in the array of particles given
 */
void FluidParticle::updateGradPressureOverDensity(FluidParticle **fluidParticles, int
numParticles) {
    glm::vec3 result = glm::vec3(0.0);
    float pressureScale;
    float dist;
    for(int i = 0; i < numParticles; i++) {
        if(this->id != fluidParticles[i]->id) {
            dist = glm::length(this->pos - fluidParticles[i]->pos);
            if(dist < MAX_DISTANCE) {
                pressureScale = (this->pressure / (this->density * this->density));
                pressureScale += (fluidParticles[i]->pressure / (fluidParticles[i]->density *
fluidParticles[i]->density));
                result += fluidParticles[i]->mass * pressureScale * gradientW(this->pos,
fluidParticles[i]->pos, FLUID_H_VALUE);
            }
        }
    }
```

```
    this->pressureTerm = result;
}

/* The viscosity times the squared gradient of velocity for a particle
 * This is an estimation method using smoothing kernals
 * the current particle should not be in the array of particles given
 */
void FluidParticle::updateViscosityGradSquaredVelocity(FluidParticle **fluidParticles, int
numParticles) {
    glm::vec3 result = glm::vec3(0.0);
    float viscosityScale = FLUID_FRICTION_MU;
    float dist;
    glm::vec3 velocityVar;
    for(int i = 0; i < numParticles; i++) {
        if(this->id != fluidParticles[i]->id) {
            dist = glm::length(this->pos - fluidParticles[i]->pos);
            if(dist < MAX_DISTANCE) {
                velocityVar = (fluidParticles[i]->velocity - this->velocity) / fluidParticles[i]->density;
                result += fluidParticles[i]->mass * velocityVar * gradientSquaredW(this->pos,
fluidParticles[i]->pos, FLUID_H_VALUE);
            }
        }
    }
    result *= viscosityScale;

    this->viscosityTerm = result;
}


/* Since we know the particles that are close to this one we can do collision
 * collision detection with the nearby particles
 * The collision 'mirror' reflection was taken from a paper that uses the twice
 * the checking distance for collision, the distance to the collision, the normal
 * it is similar to a perfect mirror reflection but through testing looks better
 * for water
 */
void FluidParticle::collisionDetection(FluidParticle **fluidParticles, int numParticles, float
timeStep) {
    // Check Bounding Box
    float dist;
    for(int i = 0; i < numParticles; i++) {
        if(this->id != fluidParticles[i]->id) {
            dist = glm::length(this->pos - fluidParticles[i]->pos);
```

```
            if(dist < MAX_DISTANCE) {
                collisionHandle(fluidParticles[i], dist);
            }
        }
    }
}

void FluidParticle::collisionHandle(FluidParticle *particle) {
    float dist = glm::length(this->pos - particle->pos) - this->radius - particle->radius;
    float epsilon = 0.0001;
    if(dist < epsilon) {
        glm::vec3 particleNorm = glm::normalize(this->pos - particle->pos);
        // change the velocities
        this->velocity = glm::reflect(this->velocity, particleNorm) * float(0.9);
        particle->velocity = glm::reflect(particle->velocity, -particleNorm) * float(0.9);
        // move outside of the object
        this->pos += particleNorm*(dist+epsilon);
        particle->pos += -particleNorm*(dist+epsilon);
    }
}

void FluidParticle::collisionHandle(FluidParticle *particle, float distance) {
    float dist =  distance - this->radius - particle->radius;
    float epsilon = 0.0001;
    if(dist < epsilon) {
        glm::vec3 particleNorm = glm::normalize(this->pos - particle->pos);
        // change the velocities
        this->velocity = glm::reflect(this->velocity, particleNorm) * float(0.9);
        particle->velocity = glm::reflect(particle->velocity, -particleNorm) * float(0.9);
        // move outside of the object
        this->pos += particleNorm*(dist+epsilon);
        particle->pos += -particleNorm*(dist+epsilon);
    }
}

void FluidParticle::boundsConstraint() {
    float epsilon = 0.0166;
    bool fixed = false;
    glm::vec3 norm;
    if(this->pos.x < xMin) {
        this->pos.x = xMin + epsilon;
        fixed = true;
        norm = glm::vec3(1.0, 0.0, 0.0);
```

```
    }
    if(this->pos.x > xMax) {
        this->pos.x = xMax - epsilon;
        fixed = true;
        norm = glm::vec3(-1.0, 0.0, 0.0);
    }
    if(this->pos.y < yMin) {
        this->pos.y = yMin + epsilon;
        fixed = true;
        norm = glm::vec3(0.0, 1.0, 0.0);
    }
    if(this->pos.y > yMax) {
        this->pos.y = yMax - epsilon;
        fixed = true;
        norm = glm::vec3(0.0, -1.0, 0.0);
    }
    if(this->pos.z < zMin) {
        this->pos.z = zMin + epsilon;
        fixed = true;
        norm = glm::vec3(0.0, 0.0, 1.0);
    }
    if(this->pos.z > zMax) {
        this->pos.z = zMax - epsilon;
        fixed = true;
        norm = glm::vec3(0.0, 0.0, -1.0);
    }

    if(fixed) {
        this->velocity = glm::reflect(this->velocity, norm) * float(0.9);
    }
}
```

```
//
//  FluidMath.hpp
//  SimpleRayTracer
//
//  Created by Eric Knapik on 3/10/16.
//  Copyright © 2016 EKnapik. All rights reserved.
//

#ifndef FluidMath_hpp
#define FluidMath_hpp

#include <stdio.h>
#include "../glm/vec3.hpp"
#include "../glm/glm.hpp"
#include <math.h>

// #define fluid constants
#define pi 3.14159265

// The fluid smoothing kernal functions

float W(glm::vec3 r, glm::vec3 rb, float h);

glm::vec3 gradientW(glm::vec3 r, glm::vec3 rb, float h);

float gradientSquaredW(glm::vec3 r, glm::vec3 rb, float h);

#endif /* FluidMath_hpp */


//
//  FluidMath.cpp
//  SimpleRayTracer
//
//  Created by Eric Knapik on 3/10/16.
//  Copyright © 2016 EKnapik. All rights reserved.
//

#include "FluidMath.hpp"


// W smoothing kernal quite arbitrary
float W(glm::vec3 r, glm::vec3 rb, float h) {
```

```
    float radius = glm::length(r-rb);
    float scale = 315 / (64 * pi * pow(h,9));
    float inner = (h*h) - (radius*radius);
    return scale * pow(inner, 3);
}

// The gradient of W smoothing kernal
glm::vec3 gradientW(glm::vec3 r, glm::vec3 rb, float h) {
    float radius = glm::length(r-rb);
    float scale = -45 / (pi * pow(h, 6));
    glm::vec3 vector = glm::normalize(r-rb);
    return scale * float(pow(h-radius,2)) * vector;
}

// the graident squared of W smoothing kernal
float gradientSquaredW(glm::vec3 r, glm::vec3 rb, float h) {
    float radius = glm::length(r-rb);
    float scale = 45 / (pi * pow(h, 6));
    return scale * (h-radius);
}

/*
 // check for 0 <= radius <=
 float radius = glm::length(r-rb);
 if(0 <= radius || radius <= h) {
 return 0;
 }
*/
```

****

Other files have been omitted because they deal with rendering
To the screen however they can all be found at:
https://github.com/EKnapik/Simple_Fluid_Sim

****