Real Time Fluid Simulation
using Smoothed-Particle Hydrodynamics and OpenGL
Computer Graphics CS 488

Michael Berg
Matthias Untergassmair

December 11, 2014

This paper explores the field of Smooth Particle Hydrodynamics (SPH), starting at its beginnings as a tool to simulate astrophysical phenomena and following its evolution and implementation as a way to simulate fluids such as water. We will then give a brief mathematical background and ensuing algorithm of our SPH simulation followed by a detailed explanation of it.

1 Introduction

Smoothed Particle Hydrodynamics (SPH) successfully simulates fluids by breaking up a fluid body into individual parts, or particles. These particles together form a particle system that simulates various gravitational forces. Fluid movement is simulated in the system by moving particles around any particle moved, simulating a rippling, wave effect.

The ripple effect is created by first calculating which particles are surrounding a particle moved, and these surrounding particles are moved according to the movement of the first particle moved. But SPH wasn't originally intended to simulate liquid substances, but astrophysical phenomena.

2 History

Given all the different applications for Smoothed Particle Hydrodynamics (SPH), it was first used to simulate interstellar phenomena. Conceived in 1977 by Gingold and Monaghan was an improvement to the Standard Finite Difference Method, which until their breakthrough, was the method to use to simulate astrophysical phenomena. They improved on this method by making "use of Lagrangian description of fluid flow which automatically focuses attention on fluid elements" **sphastrophysics** In this implementation, particles "move according to the Newtonian equations with forces due to the pressure gradient and other body forces: gravity, rotation and magnetic" **sphastrophysics**

3 Mathematical Background

$$a_i^n = \frac{F_i^n}{m_i} = \dots \tag{1}$$

4 The Algorithm

In the following, we denote the position for the particle i at time t as x_i^t, its velocity as v_i^t and its acceleration as a_i^t. We omit the vector notation $(\boldsymbol{x}, \boldsymbol{v}, \boldsymbol{a})$ for these quantities, since the following equations are valid for the vectors as well as for each component individually.

As suggested in S. Adami 2012, we use the *Velocity-Verlet* time stepping scheme as follows:

Algorithm 1: Single Timestep with Velocity Verlet Algorithm

Data: x_i^t, $v_i^{t-\frac{\Delta t}{2}}$, a_i^t, Δt

Result: $x_i^{t+\Delta t}$, $v_i^{t+\frac{\Delta t}{2}}$, $a_i^{t+\Delta t}$

$v_i^{t+\frac{\Delta t}{2}} = v_i^{t-\frac{\Delta t}{2}} + \Delta t a_i^t$;

$x_i^{t+\Delta t} = x_i^t + \Delta t v_i^{t+\frac{\Delta t}{2}}$;

$a_i^{t+\Delta t} = a_i^{t+\Delta t}(x_i^{t+\Delta t}, m_i)$ from equation 1 ;

Appendix: Code

Listing 1: sphModel.hpp

```
#ifndef SPH_HPP
#define SPH_HPP

#define _USE_MATH_DEFINES // make M_PI available

#include <iostream>
#include <stdio.h>
#include <cmath>
#include <random>
#include <unistd.h>

using std::ostream;

class SPH {

  public:

    static const unsigned _ghostDepth = 3;

    // Constructor
    SPH(unsigned);

    // Destructor
    ~SPH();

    // Time Propagation of model
    void timestep(float);

    // Update forces on particles based on SPH
    void updateForces();

    // Overloading Output Operator
    friend ostream& operator<<(ostream&, const SPH&);

    // Applying elastic boundary conditions
    void applyBoundary();

    // Get Radius of Particle i
    unsigned getTotalParticles() const;

    // Write position to the 3-array x
    inline void getPosition(unsigned index, float* x) {
      if(index >= _nTotal) {
        std::cout << "ERROR: Invalid index";
        return;
      }
      // Take into account switching of axes for OpenGL
      x[0] = _x2[index];
      x[1] = _x3[index];
      x[2] = _x1[index];
```

```
    }

    // Write velocity to the 3-array v
    inline void getVelocity(unsigned index, float* v) {
      if(index >= _nTotal) {
        std::cout << "ERROR: Invalid index";
        return;
      }
      // Take into account switching of axes for OpenGL
      v[0] = _v2[index];
      v[1] = _v3[index];
      v[2] = _v1[index];
    }

    // Return Kinetic Energy
    float getEkin() const;

    // Return Potential Energy
    float getEpot() const;

    // Get Radius of Particle i
    float getRadius(unsigned) const;

    // Setting Gravity
    void setGravity(float);

    // Functions for changing Box position
    // void moveBoxX(float);
    // void moveBoxY(float);

  private:

    unsigned _nParticles; // Number of fluid particles
    unsigned _nGhostWall; // Number of ghost particles in the
       walls
    unsigned _nGhostObject; // Number of ghost particles in the
       object
    unsigned _nTotal; // Total number of particles

    // Array of particle coordinates & velocities &
       accelerations
    float* _x1;
    float* _x2;
    float* _x3;
    float* _v1;
    float* _v2;
    float* _v3;
    float* _a1;
    float* _a2;
    float* _a3;

    // Array of particle masses
    float* _m;

```

```
    // Array of particle radii
    float* _r;

    // Wall Coordinates
    float _x2MinWall;
    float _x2MaxWall;
    float _x3MinWall;
    float _x3MaxWall;

    // Box Coordinates
    float _x2MinBox;
    float _x2MaxBox;
    float _x3MinBox;
    float _x3MaxBox;

    // Velocity component introduced by Box movement
    float _v2Box;
    float _v3Box;


    // Gravity
    float _g;

    // Damping factor for elastic bounding on walls
    float _damping;

    // Total time
    float _T;
    unsigned _tStep;

    // Size of current timestep
    float _dt;

    // Was the Box moved?
    bool _boxMoved;

};


#endif // SPH_HPP
```

Listing 2: sphModel.cpp

```
#include "sphModel.hpp"

// TODO: implement SPH interaction
// TODO: introduce ghost particles on boundary

SPH::SPH(unsigned N)
  : _nParticles(N),
    _nGhostWall(100),
    _nGhostObject(0),
    _nTotal(_nParticles+_nGhostWall+_nGhostObject),
    _x1(new float[_nTotal]),
```

```
    _x2(new float[_nTotal]),
    _x3(new float[_nTotal]),
    _v1(new float[_nTotal]),
    _v2(new float[_nTotal]),
    _v3(new float[_nTotal]),
    _a1(new float[_nTotal]),
    _a2(new float[_nTotal]),
    _a3(new float[_nTotal]),
    _m(new float[_nTotal]),
    _r(new float[_nTotal]),
    _x2MinWall(-100),
    _x2MaxWall(+100),
    _x3MinWall(-100),
    _x3MaxWall(+100),
    _x2MinBox(-40),
    _x2MaxBox(+40),
    _x3MinBox(-100),
    _x3MaxBox(+100),
    _g(0),
    _damping(.8),
    _T(0.0),
    _tStep(0),
    _dt(0),
    _boxMoved(false)
{
  std::cout << "\nInitializing Model...";

  // Seeding random number generator and set parameters for
      normal distribution
  // std::random_device rd; // Uncomment to make it even more
      random ;)
  // std::mt19937 e2(rd());
  std::mt19937 e2(42);
  float mean = 0; // mean velocity
  float stddev = 50; // standard deviation of velocity
  std::normal_distribution<> dist(mean,stddev);

  // Initialize Fluid Particles
  for(unsigned i=0; i<_nParticles; ++i) {

    // Position
    _x1[i] = 0;
    _x2[i] = (i%2 ? -50 : 50) + .1*dist(e2);
    _x3[i] = fmod(rand(),200)-100;

    // Masses (assume all particles have the same mass)
    _m[i] = 1; // 1e-6 * (1+i%3);

    // Radius / Support of particles
    _r[i] = 1; // 1+i%3;

    // Compute Forces acting on particles based on positions
    updateForces();
```

```
    // Velocities (sampled from random normal distribution)
    _v1[i] = dist(e2);
    _v2[i] = dist(e2);
    _v3[i] = 0;

    // _v1[i] = 0; // TODO: remove, v_x = 0 only for debugging
    // _v2[i] = 40.f; // TODO: remove, v_y = 40 only for
        debugging

  }

  // Initialize Ghost Particles in Wall
  for(unsigned i=_nParticles; i<_nParticles+_nGhostWall; ++i) {

    int side = (i-_nParticles)/(.25*_nGhostWall);
    float ratio = 0;

    switch(side) {

      // Position
      case 0: // Bottom
        ratio = float(i-_nParticles)/(.25*_nGhostWall);
        _x2[i] = -100 + 200*ratio;
        _x3[i] = -100;
        break;
      case 1: // Top
        ratio = float(i-_nParticles)/(.25*_nGhostWall)-1;
        _x2[i] = -100 + 200*ratio;
        _x3[i] = +100;
        break;
      case 2: // Left
        ratio = float(i-_nParticles)/(.25*_nGhostWall)-2;
        _x2[i] = -100;
        _x3[i] = -100 + 200*ratio;
        break;
      case 3: // Right
        ratio = float(i-_nParticles)/(.25*_nGhostWall)-3;
        _x2[i] = +100;
        _x3[i] = -100 + 200*ratio;
        break;
    }

    _x1[i] = 0;

    // Velocities = 0 in boundary
    _v1[i] = 0;
    _v2[i] = 0;
    _v3[i] = 0;

    // Masses (assume all particles have the same mass)
    _m[i] = 1e10;

    // Radius / Support of particles
    _r[i] = .2;
```

```
}

// Initialize Ghost Particles in Object
for(unsigned i=_nParticles+_nGhostWall; i<_nTotal; ++i) {

  int side = (i-_nParticles-_nGhostWall)/(.25*_nGhostObject);
  float ratio = 0;

  float boxWidth = _x2MaxBox - _x2MinBox;
  float boxHeight = _x3MaxBox - _x3MinBox;

  // TODO: cast float to int
  switch(side) {

    // Position
    case 0: // Bottom
      ratio = float(i-_nParticles-_nGhostWall)/(.25*
          _nGhostObject);
      _x2[i] = _x3MinBox + boxWidth*ratio;
      _x3[i] = _x3MinBox;
      break;
    case 1: // Top
      ratio = float(i-_nParticles-_nGhostWall)/(.25*
          _nGhostObject)-1;
      _x2[i] = _x3MinBox + boxWidth*ratio;
      _x3[i] = _x3MaxBox;
      break;
    case 2: // Left
      ratio = float(i-_nParticles-_nGhostWall)/(.25*
          _nGhostObject)-2;
      _x2[i] = _x3MinBox;
      _x3[i] = _x3MinBox + boxHeight*ratio;
      break;
    case 3: // Right
      ratio = float(i-_nParticles-_nGhostWall)/(.25*
          _nGhostObject)-3;
      _x2[i] = _x3MaxBox;
      _x3[i] = _x3MinBox + boxHeight*ratio;
      break;
  }


  // Velocities = 0 in Object
  _v1[i] = 0;
  _v2[i] = 0;
  _v3[i] = 0;

  // Masses (assume all particles have the same mass)
  _m[i] = 1e10;

  // Radius / Support of particles
  _r[i] = 3;
```

```
  }

}

SPH::~SPH() {
  // Free memory
  if(_r) { delete[] _r; }
  if(_m) { delete[] _m; }
  if(_a3) { delete[] _a3; }
  if(_a2) { delete[] _a2; }
  if(_a1) { delete[] _a1; }
  if(_v3) { delete[] _v3; }
  if(_v2) { delete[] _v2; }
  if(_v1) { delete[] _v1; }
  if(_x3) { delete[] _x3; }
  if(_x2) { delete[] _x2; }
  if(_x1) { delete[] _x1; }
  std::cout << "\nMemory freed";
}

void SPH::timestep(float dt) {

  // Update Time counters
  _dt = dt;
  _T += _dt;
  ++_tStep;

  // Update Forces
  updateForces();

  for(unsigned i=0; i<_nParticles; ++i) {

    // Update Velocities
    _v1[i] += _dt*_a1[i];
    _v2[i] += _dt*_a2[i];
    _v3[i] += _dt*_a3[i];

    // Update Positions
    _x1[i] += _dt*_v1[i];
    _x2[i] += _dt*_v2[i];
    _x3[i] += _dt*_v3[i];

  }

  // applyBoundary();

  // TODO: remove - sleeping only for debugging, simulates
      longer execution time
  unsigned microseconds = 20000;
  usleep(microseconds);

}

void SPH::updateForces() {
```

```
  float d1, d2, d3; // Particle Distance in each space direction
  float R; // Particle Distance in 3D space
  float theta, phi; // Angles for orientation in 3D space
  float F; // Force between two particles

  for(unsigned i=0; i<_nParticles; ++i) {

    _a1[i] = 0;
    _a2[i] = 0;
    _a3[i] = 0;

    for(unsigned a=0; a<_nParticles /*_nTotal*/; ++a) {

      if(a == i) continue; // Particles don't interact with
          themselves

      d1 = _x1[a] - _x1[i];
      d2 = _x2[a] - _x2[i];
      d3 = _x3[a] - _x3[i];

      R = sqrt(d1*d1+d2*d2+d3*d3);

      if (R == 0) continue;

      phi = atan2(d2,d1); // d2 or d3
      theta = acos(d3/R);

      // F = (std::abs(d1) > 70 ? -d1 : d1);
      F = (R>50 ? R : -10000/R);
      // R-50; // (R > 70 ? -R : R); // (R!=0 ? 1/R : 0); //
          Only Temporary force computation: Hooke's law

      _a1[i] += F*sin(theta)*cos(phi);
      _a2[i] += F*sin(theta)*sin(phi);
      _a3[i] += F*cos(theta);

      // No Interaction
      /*
      _a1[i] = 0;
      _a2[i] = 0;
      _a3[i] = 0;
      */

    }
    _a3[i] += _g; // add gravity

  }
  std::cout << "Acceleration 1: " << _a1[0];
  std::cout << "Acceleration 10: " << _a1[9];
  std::cout << "Acceleration 100: " << _a1[99];
}
```

```
void SPH::applyBoundary() {

  float center1Box = .5*(_x2MinBox+_x2MaxBox);
  float center2Box = .5*(_x3MinBox+_x3MaxBox);

  for(unsigned i=0; i<_nParticles; ++i) {

    // Additional velocitiy componentes introduced by box movement
    float v2Box = 0;
    float v3Box = 0;

    // Check if the box was moved within the last time interval
    if(_boxMoved) {
      v2Box = _v2Box;
      v3Box = _v3Box;
      _boxMoved = false; // movement of box has been considered, set to false now
      _v2Box = 0; // reset velocity components of Box to zero
      _v3Box = 0; // reset velocity components of Box to zero
    }

    // Elastic reflection on wall
    if(_x1[i] <= _x2MinWall) _v1[i] = +_damping*std::abs(_v1[i]);
    if(_x1[i] >= _x2MaxWall) _v1[i] = -_damping*std::abs(_v1[i]);
    if(_x2[i] <= _x3MinWall) _v2[i] = +_damping*std::abs(_v2[i]);
    if(_x2[i] >= _x3MaxWall) _v2[i] = -_damping*std::abs(_v2[i]);

    // Elastic reflection on box
    // if(_x1[i] >= _x2MinBox && _x1[i] < center1Box /*&& _x2[i] >= _x3MinBox && _x2[i] < center2Box*/) _v1[i] = -_damping*std::abs(_v1[i]) + v1Box;
    // if(_x1[i] <= _x2MaxBox && _x1[i] > center1Box /*&& _x2[i] <= _x3MaxBox && _x2[i] > center2Box*/) _v1[i] = +_damping*std::abs(_v1[i]) + v1Box;
    // if(_x2[i] >= _x3MinBox && _x2[i] < center2Box /*&& _x2[i] >= _x3MinBox && _x2[i] < center2Box*/) _v2[i] = +_damping*std::abs(_v2[i]) + v2Box;
    // if(_x2[i] <= _x3MaxBox && _x2[i] > center2Box /*&& _x2[i] <= _x3MaxBox && _x2[i] > center2Box*/) _v2[i] = +_damping*std::abs(_v2[i]) + v2Box;

    /*
    if(_x2[i] >= _x3MinBox && _x2[i] < center2Box) _v2[i] = -std::abs(_v2[i]);
    if(_x2[i] <= _x3MaxBox && _x2[i] > center2Box) _v2[i] = +std::abs(_v2[i]);
*/
  }

```

```
}

/*
void SPH::moveBoxX(float dx) {
  float tmpMinX = _x2MinBox + dx;
  float tmpMaxX = _x2MaxBox + dx;
  if(tmpMinX > _x2MinWall && tmpMaxX < _x2MaxWall) {
    _x2MinBox = tmpMinX;
    _x2MaxBox = tmpMaxX;

    // Move Ghost particles
    for(unsigned i=_nParticles+_nGhostWall; i<_nTotal; ++i) {
      _x2[i] += dx;
    }

  } else {
    std::cout << "You hit the wall";
  }
  _v2Box = dx/_dt;
  _boxMoved = true;

}

void SPH::moveBoxY(float dy) {
  float tmpMinY = _x3MinBox + dy;
  float tmpMaxY = _x3MaxBox + dy;
  if(tmpMinY > _x3MinWall && tmpMaxY < _x3MaxWall) {
    _x3MinBox = tmpMinY;
    _x3MaxBox = tmpMaxY;

    // Move Ghost particles
    for(unsigned i=_nParticles+_nGhostWall; i<_nTotal; ++i) {
      _x3[i] += dy;
    }

  } else {
    std::cout << "You hit the wall";
  }
  _v3Box = dy/_dt;
  _boxMoved = true;

}
*/

float SPH::getRadius(unsigned i) const {
  return _r[i];
}

float SPH::getEkin() const {
  float Ekin;
  for(unsigned i=0; i<_nParticles; ++i) {
    Ekin += _m[i] * (_v1[i]*_v1[i] + _v2[i]*_v2[i] + _v3[i]*_v3[
        i]);
  }
```

```
  return .5*Ekin;
}

float SPH::getEpot() const {
  float Epot;
  for(unsigned i=0; i<_nParticles; ++i) {
    Epot += _m[i] * _x2[i];
  }
  return _g*Epot;
}

void SPH::setGravity(float g) {
  _g = g;
}

unsigned SPH::getTotalParticles() const {
  return _nTotal;
}


// Overloaded output operator
ostream& operator<<(ostream& os, const SPH& s) {

  os << "\n===========================================================";
  os << "\nTime:   " << s._T << "\tTimestep:   " << s._tStep;
  os << "\nGravity:\t" << s._g;
  os << "\nKinetic Energy:   \t" << s.getEkin();
  os << "\nPotential Energy: \t" << s.getEpot();
  os << "\nBox:\t[ " << s._x2MinBox << " , " << s._x2MaxBox << " ] x [ " << s._x3MinBox << " , " << s._x3MaxBox << " ]";

  unsigned nOutput = 2; // Only output first particle
  // unsigned nOutput = s._nParticles; // All particles

  os << "\nPosition:\t| ";
  for(unsigned i=0; i<nOutput; ++i) {
    printf("%3.4f %3.4f %3.4f | ", s._x1[i], s._x2[i], s._x3[i]);
  }

  os << "\nVelocity:\t| ";
  for(unsigned i=0; i<nOutput; ++i) {
    printf("%3.4f %3.4f %3.4f | ", s._v1[i], s._v2[i], s._v3[i]);
  }

  os << "\nAcceleration:\t| ";
  for(unsigned i=0; i<nOutput; ++i) {
    printf("%3.4f %3.4f %3.4f | ", s._a1[i], s._a2[i], s._a3[i]);
  }
```

```
  os << "\n
    ==========================================================
    ";
  os << "\n";

  return os;
}
```

Listing 3: Simulation.cpp

```
#define GLM_FORCE_RADIANS
#define BUFFER_OFFSET(i) (reinterpret_cast<void*>(i))

#include <string>


#ifdef TARGET_OS_MAC // MAC
  std::string platform = "MAC";
  // TODO: Include Mac Headers here
#elif defined __linux__ // LINUX
  std::string platform = "LINUX";
  #include "Aluminum/Includes.hpp"
  #include "Aluminum/Program.hpp"
  #include "Aluminum/MeshBuffer.hpp"
  #include "Aluminum/MeshData.hpp"
  #include "Aluminum/Shapes.hpp"
  #include "Aluminum/Camera.hpp"
  #include "Aluminum/Utils.hpp"
  #include "Aluminum/MeshUtils.hpp"
  #include "Aluminum/FBO.hpp"
  #include "Aluminum/Behavior.hpp"
  #include "Aluminum/ResourceHandler.hpp"
  #include "Aluminum/Texture.hpp"
  #include "Aluminum/RendererLinux.hpp"
#elif defined _WIN32 || defined _WIN64
  std::string platform = "WINDOWS";
#else
#error "unknown platform"
#endif



#include "sphModel.hpp"
#include "extendedShapes.hpp"




using glm::vec3;
using glm::mat4;

float pi = glm::pi<float>();
```

```
using namespace aluminum;


// TODO: improve performance by only adding one single sphere instead of N spheres (just use different model matrices).
// TODO: make liquid flow in from top
// TODO: pass in only points to shader and use geometry shader to create 3d particles
// TODO: see 3.5.1: flowing water and particle effects, stream output

class Simulation : public RendererLinux {

public:

  static const unsigned N = 0; // 40;
  unsigned M = 0;

  ResourceHandler rh;
  Camera camera;
  Program program;

  GLint posLoc = 0;
  GLint normalLoc = 1;
  GLint colLoc = 2;

  MeshBuffer* mb;

  mat4 view, proj;
  Behavior rotateBehavior;

  bool gravityOn;



  SPH fluidsimulation = SPH(N); // Initialize Fluid simulation model with N particles


  unsigned stepCounter = 0; // TODO: remove - step counter that keeps track of how many timesteps have been done - model stops after certain number of steps



  void onCreate() {

    // Output Simulation state
    std::cout << "\nModel Parameters after Initialization:\n" << fluidsimulation;

    rh.loadProgram(program, "resources/simulation", posLoc, normalLoc, -1, colLoc);
```

```
  M = fluidsimulation.getTotalParticles(); // Render all
     particles
  mb = new MeshBuffer[M];

  for(int i=0; i<M; ++i) {
    MeshData md;
    // addCube(md,fluidsimulation.getRadius(i),vec3(0,0,0));
    // addRect(md,4.f,4.f,100.f,vec3(0,0,0));
    addSphere(md,5*fluidsimulation.getRadius(i),8,8);
    mb[i].init(md,posLoc,normalLoc,-1,colLoc);
  }

  glEnable(GL_DEPTH_TEST);
  glViewport(0, 0, width, height);

  rotateBehavior = Behavior(now()).delay(1000).length(5000).
     range(vec3(3.14, 3.14, 3.14)).reversing(true).repeats
     (-1).linear();

  camera = Camera(glm::radians(60.0),1.,0.01,1000.0);
  camera.translateZ(-400);


  gravityOn = false;

}

  void loadProgram(Program &p, const std::string& name) {

  p.create();
  p.attach(p.loadText(name + ".vsh"), GL_VERTEX_SHADER);

  glBindAttribLocation(p.id(), posLoc, "vertexPosition");
  // glBindAttribLocation(p.id(), colLoc, "vertexColor");
  glBindAttribLocation(p.id(), normalLoc, "vertexNormal");

  p.attach(p.loadText(name + ".fsh"), GL_FRAGMENT_SHADER);
  p.link();

  }


void onFrame(){

  /////////////////////////////////////////////////////////
  // PROPAGATE MODEL
  /////////////////////////////////////////////////////////

  if(stepCounter < 0 /*5000*/) {
    ++stepCounter;
    fluidsimulation.timestep(.05); // Propagate
       fluidsimulation in time
    std::cout << fluidsimulation; // Output current status of
```

```
        Fluid particles
  }

  // Getting position data for rendering
  /*

    unsigned M = 5;

    float* X = new float[3*M];
    float* V = new float[3*M];

    for(unsigned i=0; i<M; ++i) {
      fluidsimulation.getPosition(i,(X+3*i));
      fluidsimulation.getVelocity(i,(V+3*i));
    }

    // TODO: position = position, velocity = colorcoded
    // TODO: opengl: allow switching from particle view to
        grid view

    delete[] V;
    delete[] X;
  */


  ///////////////////////////////////////////////////////////



  // Start displaying

  glViewport(0, 0, width, height);
  // glClearColor(0.1,0.1,0.1,1.0);
  glClearColor(1.0,1.0,1.0,1.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  if (camera.isTransformed) {
    camera.transform();
  }

  vec3 totals = vec3(.0f,.0f,.0f); // rotateBehavior.tick(now
      ()).totals(); // TODO: uncomment for rotation

  // Draw Cubes
  for(int i=0; i<M; ++i) {
    program.bind(); {
  /*
  proj = glm::perspective(45.0, 1.0, 0.1, 100.0);
  view = glm::lookAt(vec3(0.0,0.0,100), vec3(0,0,0), vec3
      (0,1,0) );
*/

      mat4 model = mat4(1.0);

```

```
        float position[3];
        float velocity[3];
        fluidsimulation.getPosition(i,position);
        fluidsimulation.getVelocity(i,velocity);

        model = glm::translate(model,vec3(position[0],position
            [1],position[2]));

        // For Rotation of Cubes
        /*
        model = glm::rotate(model, -totals.x, vec3(1.0f,0.0f,0.0
            f));
        model = glm::rotate(model, -totals.y, vec3(0.0f,1.0f,0.0
            f));
        model = glm::rotate(model, -totals.z, vec3(0.0f,0.0f,1.0
            f));
        */

        glUniformMatrix4fv(program.uniform("model"), 1, 0, ptr(
            model));
        glUniformMatrix4fv(program.uniform("view"), 1, 0, ptr(
            camera.view));
        glUniformMatrix4fv(program.uniform("proj"), 1, 0, ptr(
            camera.projection));

        glUniform3f(program.uniform("velocity"), std::abs(
            velocity[0])/100,velocity[1]/100,.25);

        mb[i].draw();
      } program.unbind();

    }

  }


  // Keyboard Interaction

  void specialkeys(int key, int x, int y) {

    // FreeGlutGLView::specialkeys(key,x,y);

    // Switch Cross Compatible with Linux/MacOS

    float dxBox = 1;

    if(key == GLUT_KEY_UP || false) {
      // camera.rotateX(glm::radians(-2.));
      // fluidsimulation.moveBoxY(dxBox);
    } else if(key == GLUT_KEY_DOWN || false) {
      // camera.rotateX(glm::radians(2.));
      // fluidsimulation.moveBoxY(-dxBox);
    } else if(key == GLUT_KEY_RIGHT || false) {
      // camera.rotateY(glm::radians(2.));
```

```
      // fluidsimulation.moveBoxX(+dxBox);
    } else if(key == GLUT_KEY_LEFT || false) {
      // fluidsimulation.moveBoxX(-dxBox);
      // camera.rotateY(glm::radians(-2.));
    }

  }


  void keyboard(unsigned char key, int x, int y) {

    float dxCamera = 5;

    if(key == ' ' || false) {
      camera.resetVectors();
    } else if(key == 'a' || false) {
      camera.rotateY(glm::radians(-2.));
    } else if(key == 's' || false) {
      camera.rotateY(glm::radians(+2.));
    } else if(key == 'n' || false) {
      camera.translateZ(-dxCamera);
    } else if(key == 'u' || false) {
      camera.translateZ(+dxCamera);
    } else if(key == 'h' || false) {
      camera.translateX(+dxCamera);
    } else if(key == 'l' || false) {
      camera.translateX(-dxCamera);
    } else if(key == 'k' || false) {
      camera.translateY(-dxCamera);
    } else if(key == 'j' || false) {
      camera.translateY(+dxCamera);
    } else if(key == 'g' || false) {
      if(gravityOn) {
        fluidsimulation.setGravity(0);
        gravityOn = false;
      } else {
        fluidsimulation.setGravity(-20);
        gravityOn = true;
      }
    }
  }

};


int main(){
  std::cout << "\n\nRunning on Platform: " << platform << "\n\n"
      ;
  Simulation().start();
  return 0;
}
```

Listing 4: simulation.vsh

```
#version 150
uniform mat4 proj, view, model;
uniform vec3 velocity;

in vec4 vertexPosition, vertexNormal, vertexColor;

out vec3 color;

void main() {

  vec4 position = view * model * vertexPosition;

  color = velocity; // vertexColor.xyz;

  gl_Position = proj * position;

}
```

Listing 5: simulation.fsh

```
#version 150

in vec3 color;

out vec4 frag;

void main(){

  frag = vec4(vec3(color),1.0);

}
```

References

Akenine-Möller, Tomas, Eric Haines, and Natty Hoffman (2008). *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., p. 1045. ISBN: 987-1-56881-424-7.

S. Adami X.Y. Hu, N.A. Adams (2012). "A generalized wall boundary condition for smoothed particle hydrodynamics". In: *Journal of Computational Physics* 231.