CA Final Project Report
Liquid Interaction Simulation under Different Viscosity and Contact Force

0316235 Li-Che Chien
0656039 Sheng-Tang Wong
0656136 Huan-Wei Kang

June 25, 2018

1 Introduction

In the field of computer animation, the computation of fluids is very important for the presentation of visual effects, but it is the one that consumes the most computing resources. Fluids are not just liquids, they also contain flames, smoke, gases, etc. The concept of fluids is not only applied to visual effects, in the field of science, the use of computers to assist fluid dynamics research is called **Computational Fluid Dynamics (CFD)**, which was widely used in weather forecasting, ocean currents, and aircraft dynamics in the 1960s.

Liquid collisions with solid wall surfaces are widespread in nature, its collision behavior and dynamics draw our attention. Viscosity of fluids represents the magnitude of internal resistance when the fluid flows, the greater the viscosity, the greater the friction between the fluid and walls.

Our final project is to study the behavior of fluids in contact with solids at different viscosities.

2 Fundamentals

2.1 Navier-Stokes Equations

Fluid simulation is mainly to solve the **Navier-Stokes equations**. This set of equations has been several hundred years old and can be used to describe different types of fluids.

Basically, this equation is similar to applying **Newton's second law of motion** (that is, $F = ma$) to the fluid. The final solution of the Navier-Stokes equation is not a single simple number, but a set of complex vector fields that represent the velocity field or flow field, which can used to describe the distance and time of fluid movement at a particular point. Once the velocity field is solved, the velocity or resistance of particles can also be derived. The simplified Navier-Stokes equations is as follows:

$$\rho\frac{\partial\vec{u}}{\partial\vec{t}} + (\vec{u}\cdot\nabla)\vec{u} = -\nabla\vec{p} + \mu\Delta\vec{u} + \rho g$$

$$\nabla\cdot\vec{u} = 0$$

Where ρ represents the density of the fluid, $\vec{u}$ represents the velocity of the fluid in the three-dimensional space, $\vec{p}$ is the pressure, μ is the fluid viscosity, g is the gravitational acceleration, ∇ is the gradient, and Δ is the Laplace operator.

The first equation describes Newton's second law of motion for each small unit of fluid obeys, where strength is expressed in terms of density; the second equation is the **Incompressibility condition**. In terms of physical conditions, incompressibility means that the density is constant over time.

2.2 Smoothed-Particle Hydrodynamics

Two kinds of methods commonly used in fluid simulation today represent the Navier-Stokes equations from two different aspects:

1. The **Eulerian specification** randomly selects a fixed point in the space, and then observe changes in the fluid variables at the fixed point.

2. The **Lagrangian specification** treats the flow of a liquid as an infinite number of particles.

Nowadays, real-time fluid simulations commonly used in games often use the Lagrangian method, and most of them are based on the **Smoothed-particle hydrodynamics (SPH)** method. In the early years, SPH was used to simulate the collision of galaxies and the formation of celestial bodies in space physics. It has recently been applied to the study of fluids, heat, and other phenomena.

Imagine a point $\vec{r}$ (which does not necessarily have particle here) in the fluid and there are several particles in the kernel radius h of the point. Let their positions be $\vec{r_0}, \vec{r_1}, \vec{r_2}, ..., \vec{r_j}$, then the equation for any quantity A at this point is given by:

$$A(\vec{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\vec{r} - \vec{r_j}, h)$$

Where m is the particle mass, ρ is the particle density, and W is the kernel function, the effective radius of the kernel function is h, and the value outside the effective radius is zero. The kernel function must satisfy two important properties, first it must be an even function, that is, $W(-r) = W(r)$, and second its integral should be one, that is, $\int W(r)dr = 1$.

Using the above equation, the density equation for the fluid can be expressed as:

$$\rho(\vec{r}) = \sum_j m_j \frac{\rho_j}{\rho_j} W(\vec{r} - \vec{r_j}, h) = \sum_j m_j W(\vec{r} - \vec{r_j}, h)$$

According to the kernel function, we can derive the accumulative function of the density, pressure and velocity at a certain point in the fluid one by one, and then derive the acceleration here to simulate the movement trend of the fluid. The SPH method is summarized as follows:

1. For each particle, calculate its density by all nearby particles whose distance radius is less than h, where h is called **Smoothing Length**, which represents the maximum radius of influence for the particle.

2. For each particle, calculate its pressure from the density using the **ideal gas equation of state**.

3. For each particle, calculate its pressure difference by the pressure of all nearby particles whose distance radius is less than h.

4. For each particle, calculate its viscosity by the difference in velocity of all nearby particles whose distance radius is less than h.

3 Implementation

3.1 SPH system

We used the code from **bikush** on GitHub called **simple-sph-simulation** to help us implement SPH fluid simulation [1]. The code was run under the c++ visual studio 2015 and rendered with openGL. We fixed, rewrote and added a lot of things in this project. The implementation of SPH is used to simulate the fluid has a grid with cells the size of the smoothing length. Calculations of SPH require a lot of multiplication with particle mass and division by particle densities. However that part has been simplified to a multiplication with the particle volume that is pre-calculated before every step.

The simulation is implemented in the file called **SPHSystem3d** which include the way we implemented grid creation, force applying, particle adding and animation of whole simulation process (Fig. 1). Moreover, we can interact with users through keyboard, for example, users can

press L as input to add particles or press I to adjust viscosity (Fig. 2). We implemented user interaction in the file called **SPHScene**. We also implemented interaction forces (Fig. 3) and the shader drawer (Fig. 4 and 5).

3.2 Shader

We rewrite shader by applying new texture and fragment caculation mode, the original shader makes particle ambiguous, so I fix it using new texture and disabling the 3D texture. For the marching cube, we resize it and fix some smooth length problem. For the interactor (the ball), we apply new shader as single and independent one, with the 2D texture on it.

3.3 Interactor

There are two types of interactor in our project, first is container and follwed by the soccer ball(new created). When implementing ball as new interactor, I construct a new particle and set its radius as 2, density as 3. I formed a new class inherented from container class, and apply new force to all particle when collide. The new interactor is also constrained by the container, as it falls from sky, it accelerated by gravity, and when it collide with particles, first it apply force with Kp gradient to scatter particles like splash, then it slow down by the force back from liquid particles.

4 Result

We simulated the fluid under different viscosity and different container. For each fluid with different viscosity, we put it in different container and drop each fluid particle from top to see its behavior under gravity and interaction with other fluid particles. We also simulated the interaction between the fluid and a soccer ball.

First, we simulated the fluid with viscosity 0.08, which we expected that the behavior of the fluid would look like water. As our expectation, the fluid moved smoothly and mixed with other particles (Fig. 6 or video).

Second, we simulated the fluid with viscosity 0.33, which we expected the fluid would look like fresh blood, which would be in lower velocity than water under the same gravity. And still, the fluid was indeed with lower speed, not moved as smoothly as water (Fig. 7 or video). Although the blood in our video was blue not common color.

Third, we simulated the fluid with viscosity 1.6, which we expected the fluid would look like a sweet beverage. It moved even slower and the fluid particles are more likely to stick together (Fig. 8 or video).

And the last, we simulated the fluid with viscosity 1.03 and the water in particle mode (Fig. 9 and Fig. 10). The fluid with viscosity 1.03 moved slower than blood but faster than beverage (video). And we can see the interaction between particles clearly in the **Particle Mode** (video).

5 Conclusion

Liquid simulation is a really interesting and difficult topic. The fundamental of liquid, all these formular and equations, is not easy to understand. And there are much more problems when it comes to implementation. However, we got a lot of fun doing the project. We have learned how liquid move, how liquid take force and how liquid interact with other objects including liquid itself.

The liquid we simulated is quite similar to real liquid. The bigger the viscosity is, the slower the speed is. However, the simulation program would crash if the viscosity is higher than specific value. So, we failed to simulate high viscosity liquid, like honey. Maybe we will solve it in the future.

6 Reference

[1] FLUID SIMULATION SIGGRAPH 2007 Course Notes
[2] Fluid Simulation Using Implicit Particles
[3] SPH survival kit

[4] Matthias Müller, David Charypar, Markus Gross, Particle-based fluid simulation for interactive applications, Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation, July 26-27, 2003, San Diego, California
[5] F. Colin, R. Egli and F.Y. Lin, Computing a null divergence velocity field using smoothed particle hydrodynamics, Journal of Computational Physics, In Press, Corrected Proof, Available online 24 February 2006.

SPHSystem3d.cpp

```
void SPHSystem3d::createGrid()
{
    int newGridHeight = (int)ceil( dHeight / smoothingLength );
    int newGridWidth = (int)ceil( dWidth / smoothingLength );
    int newGridDepth = (int)ceil( dDepth / smoothingLength );
    // If the grid dimensions change, rebuild the grid vectors
    if( gridWidth != newGridWidth || gridHeight!= newGridHeight || gridDepth!=newGridDepth)
    {
        /*gridWidth = newGridWidth;
        gridHeight = newGridHeight;
        gridDepth = newGridDepth;/**/
        gridWidth = 3;
        gridHeight = 3;
        gridDepth = 3;/**/
        grid.clear();
        for(int i=0; i< gridHeight*gridWidth*gridDepth; i++)
        {
            grid.push_back( vector< int >() );
        }
    }else   // else just clear the existing grid vectors
    {
        clearGrid();
    }
    fillGrid();

/* Build index offsets, C current cell, (A,R,F) cell to visit, X ignored cell
   dx          -1              0              +1
   ------------------------------------------------------------------------------
    dz\dy  -1   0  +1      -1   0  +1      -1   0  +1    dy/dz
    +1      X | X | X       A | A | A       F | F | F      +1
           -----------     -----------     -----------
     0      X | X | X       X | C | R       F | F | F       0
           -----------     -----------     -----------
    -1      X | X | X       X | X | X       F | F | F      -1

   */
```

Figure 1: The function to create grids

SPHScene.cpp

```
void SPHScene::eventKeyboardUp(sf::Keyboard::Key keyPressed)
{
    switch (keyPressed)
    {
        case sf::Keyboard::Num9:
            if (!interactored){
                interactor->addInteractor(sph3->addInteractor(glm::vec3(7, 7, 5), glm::vec3(0, 5, 0)));
                interactored = true;
            }
            break;

        case sf::Keyboard::Add:
                    treshold = contain<float>( treshold+0.01f, 0, 2 );
                    marchingCubes->setTreshold( treshold );
                    break;

        case sf::Keyboard::Subtract:
                    treshold = contain<float>( treshold-0.01f, 0, 2 );
                    marchingCubes->setTreshold( treshold );
                    break;

        case sf::Keyboard::P:
                    paused = !paused;
                    break;

        case sf::Keyboard::O:
                    {
                        float o = 0.0f;
                        float d = 1.0f;
                        sph3->addParticle( glm::vec3( 2, 5, 2 ), glm::vec3( 2,0,2 ) );
                    }
                    break;

        case sf::Keyboard::L:
                    {
                        for(int i = 0; i< 2;i++){
                            float o =0;
                            float d = 1;
```

Figure 2: The function implement user interaction

```
void SPHSystem3d::applyInteractorForces(SPHParticle3d& particle)
{
    if (iteractorID == -1);
    else {
        SPHParticle3d interactor_ = particles[iteractorID];
        glm::vec3 rvec;

        rvec = particle.position - interactor_.position;  // + glm::vec3(1.161f, 1.161f, 1.161f)

        float rSq;
        glm::vec3 oldForce;

        rSq = glm::length2(rvec);
        if (rSq < 16)
        {
            oldForce = particle.force;
            if (_isnan(particle.force.x) == 1)
            {
                particle.force = oldForce;
            }
            // pressure
            float pressure = particle.pressure;
            particle.force += (ksgradient(rvec) * pressure * particle.volume)*0.5f;
            if (_isnan(particle.force.x) == 1)
            {
                particle.force = oldForce;
            }
            // viscosity
            //particle.force -= ( particle.velocity ) * (kvlaplacian( sqrtf(rSq) ) * viscosityConstant * particleMass
            particle.force += (particle.velocity) * (kvlaplacian(sqrtf(rSq)) * viscosityConstant * particle.volume);
            if (_isnan(particle.force.x) == 1)
            {
                particle.force = oldForce;
            }
        }
```

Figure 3: The function implement interaction force

```
void Interactor::drawShaded(const Camera& camera)
{
    glDisable(GL_CULL_FACE);
    glEnable(GL_CULL_FACE);
    auto projection = camera.getProjection();
    auto modelView = camera.getView() * transform.getTransformMatrix();
    auto MVP = projection * modelView;

    auto up = camera.getUp();

    shader->turnOn();
    shader->setUniformM4("ModelViewMatrix", glm::value_ptr(modelView));
    shader->setUniformM4("ProjectionMatrix", glm::value_ptr(projection));
    shader->setUniformM4("mvp", glm::value_ptr(MVP));
    shader->setUniformV3("up", up.x, up.y, up.z);
    shader->setUniformF("Size2", pointSize);
    shader->setUniformI("SpriteTex", 0);
    shader->setUniformV3("Color", color.x, color.y, color.z);

    drawArray();
    shader->turnOff();

    glEnable(GL_CULL_FACE);
}

void Interactor::draw(const Camera& camera)
{
    ShaderProgram::turnOff();
    glEnable(GL_TEXTURE_2D);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, textureID);

    GLboolean blendEnabled = glIsEnabled(GL_BLEND);
    glEnable(GL_BLEND);
    glBlendFunc(GL_ONE, 0);
    glDepthMask(GL_FALSE);
    drawShaded(camera);

    glDepthMask(GL_TRUE);
    if (!blendEnabled) glDisable(GL_BLEND);

    glDisable(GL_TEXTURE_2D);
}
```

Figure 4: The function implement drawer

```
void MarchingCubesShaded::draw(const Camera& camera)
{
    glm::mat4 mvp = camera.getViewProjection() * transform.getTransformMatrix();
    glm::vec3 eye = camera.getPosition();

    glDisable( GL_CULL_FACE );
    mcShader->turnOn();
        mcShader->setUniformF( "Treshold", this->treshold );
        mcShader->setUniformV3( "Eye", eye.x, eye.y, eye.z );
        mcShader->setUniformM4( "MVP", glm::value_ptr(mvp) );

        MarchingCubesFactory::setTexture( GL_TEXTURE1 );    //33985

        glEnable( GL_TEXTURE_3D );
        glActiveTexture( GL_TEXTURE0 );
        glBindTexture( GL_TEXTURE_3D, dataTexID );
        if (dataChanged)
        {
            glTexImage3D(GL_TEXTURE_3D, 0, GL_ALPHA32F_ARB, dataWidth, dataHeight, dataDepth, 0, GL_ALPHA, GL_FLOAT, dataField);
            dataChanged = false;
        }

        // !
        GLboolean blendEnabled = glIsEnabled( GL_BLEND );
        glEnable( GL_BLEND );
        glBlendFunc(GL_ONE, GL_ONE);
        glDepthMask(GL_FALSE);
            glBindVertexArray(gridVao);
            glDrawArrays(GL_POINTS, 0, gridElementCount );
            glBindVertexArray(0);
        glDepthMask(GL_TRUE);
        if (!blendEnabled) {
            glDisable(GL_BLEND);
        }
```

Figure 5: The function implement drawer

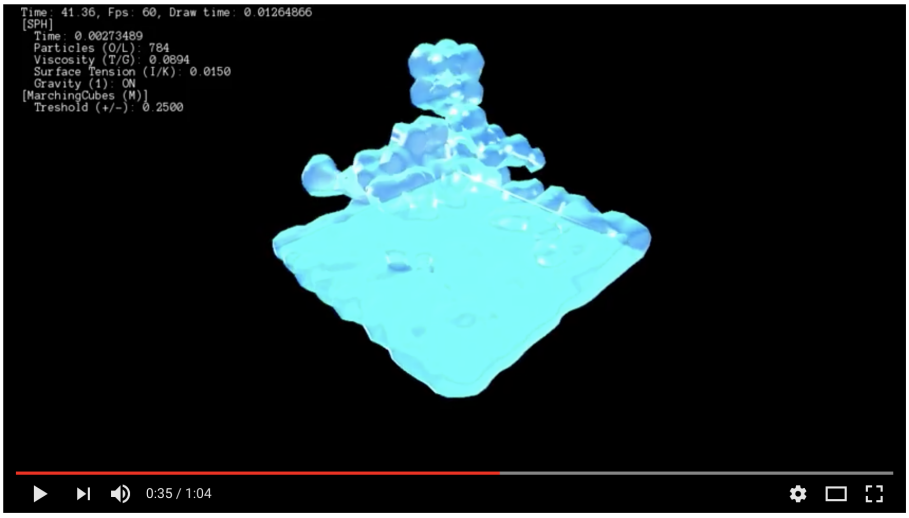

Figure 6: The fluid with 0.08 viscosity - water

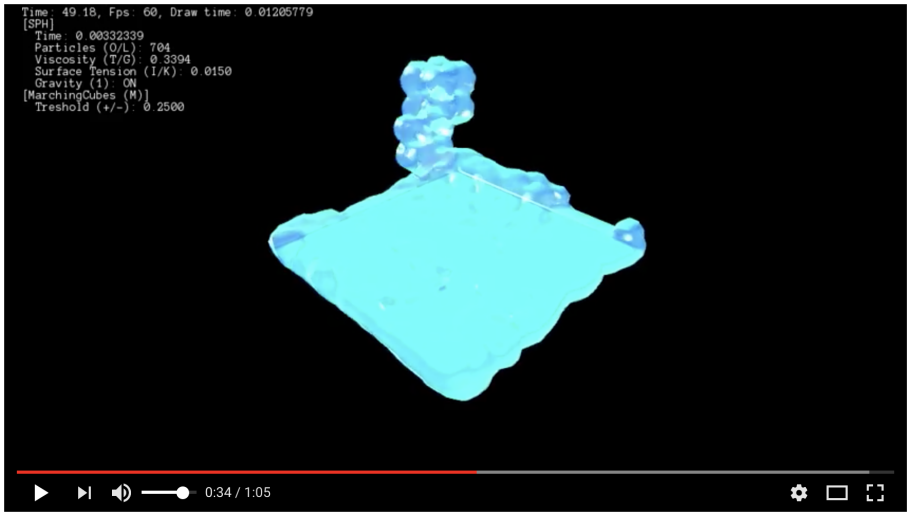

Figure 7: The fluid with 0.33 viscosity - blood

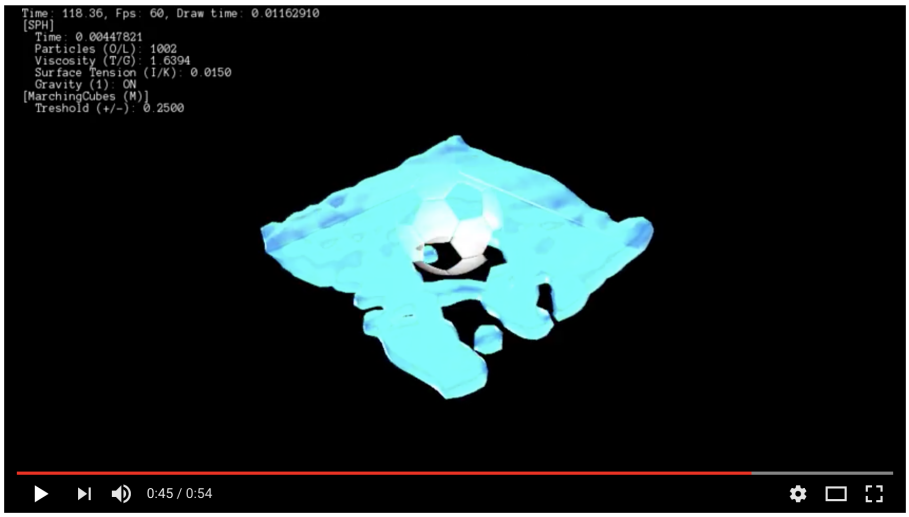

Figure 8: The fluid with 1.6 viscosity

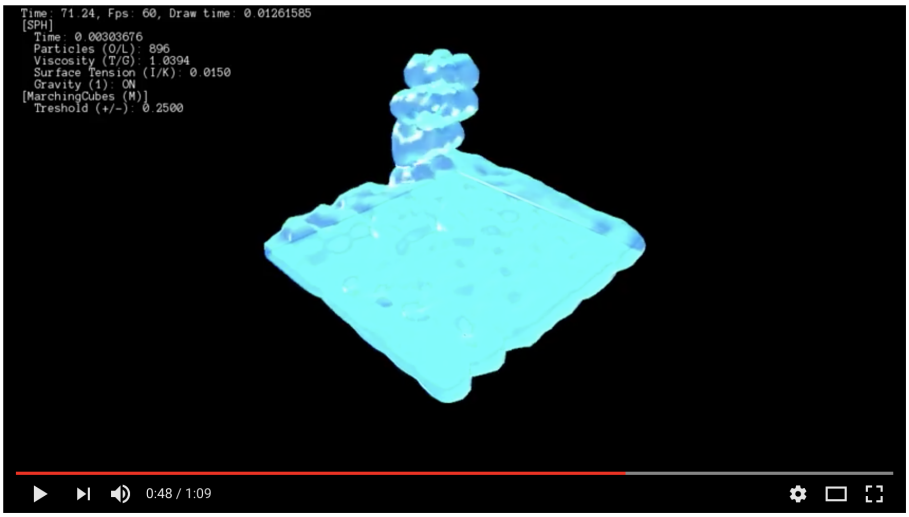

Figure 9: The fluid with 1.03 viscosity

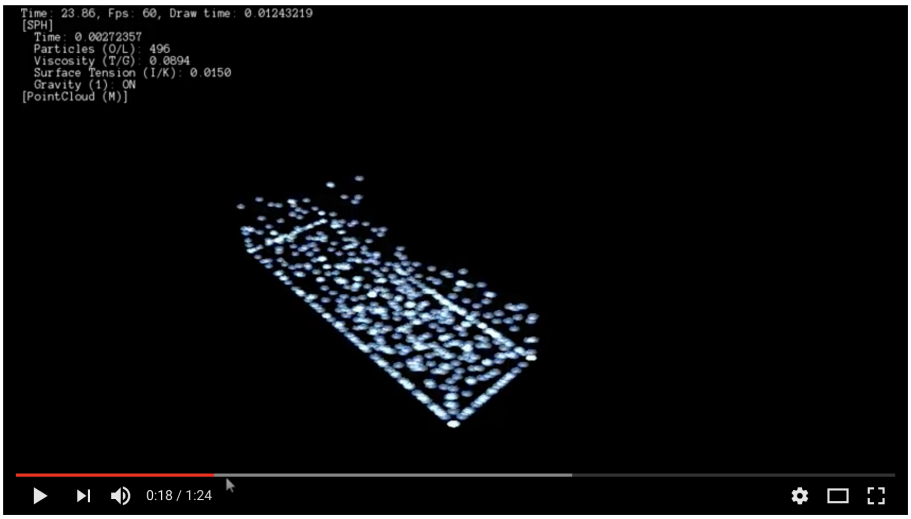

Figure 10: The fluid with 0.08 viscosity