Simulating Fluid Motion using Smoothed Particle Hydrodynamics

Aayush Joshi

Abstract

Realistic simulation of fluids is an important tool with a wide variety of applications such as within the Aerospace industry to model fluid based phenomena of spacecraft parts and within the computer games industry for authentic graphics. In this paper, I explore a method for simulating fluids known as Smoothed Particle Hydrodynamics (SPH) in order to better understand the mathematical theory behind Computational Fluid Dynamic methods and their implementation in an appropriate programming language, C++.

Contents

1 Introduction

The field of simulation is one with many applications in all industries, with much overlap between Mathematics, Physics and Computer Science due to its predictable behaviour. One such application is Computational Fluid Dynamics (CFD), or in other words predicting the movement of fluids, which will be the focus for this project.

Simulating fluids involves observation of fluid phenomena such as wind, weather, ocean waves, waves induced by ships or simply pouring a glass of water. Such phenomena may seem extremely trivial at first glance, but in reality involve a deeper understanding of physical, mathematical and algorithmic methods.

1.1 Motivation

My motivation for this project stems from the work of Sebastian Lague [1], a games developer who shares his exemplar work on Github and through digital media on YouTube. Through his work, I was introduced to the concept of Smoothed Particle Hydrodynamics in the Computer Graphics community and was given great insight into the expectations from a project such as this. Further reading, especially into the sources of Lague, piqued my interest and only reinforced the idea of undertaking this concept because it provided the overlap between Mathematics, Physics and Computer Science, it was far beyond the scope of the A level curriculum but most importantly it provided a means to challenge, extend and implement new knowledge in a field which I plan to undertake in the future.

1.2 Definitions

In this section, I provide clear and concise definitions for the key terms essential to understanding the context and methods presented in this paper.

Computer Graphics. A technology that generates images and videos on a computer screen, also referred to as CG.

Simulation. Imitation of a situation or process.

Frame. A single image which makes up a collection of images for an animation.

Render. The process of generating a photorealistic or non-photorealistic image from a 2D or 3D model

Algorithm. A set of instructions used to solve a particular problem or perform a specific task.

Pseudocode. Writing an algorithm in plain English for design purposes.

Compute time. The time required for a computer system to perform a specific task or calculation.

Optimisation. Modifying an algorithm or software to reduce the usage of computer resources or compute time.

Lagrangian. A particle based approach to simulation.

Eulerian. A grid based approach to simulation.

Velocity. Speed of an entity associated with a direction.

Acceleration. The rate of change of velocity.

Force. An influence which causes an object to accelerate.

Friction. A force resisting the relative motion of an object.

Fluid. Any substance which flows due to applied forces, namely Liquids and Gasses.

Liquid. A type of fluid which takes the shape of any container or vessel it is stored within.

Advection. The horizontal movement of a mass of fluid.

Incompressible.

Mass. The measure of the amount of matter in a system.

Density. The compactness of a substance, or the mass per unit volume.

Pressure. The physical force exerted on an object by something in contact with it, or the force per unit area.

Viscosity. A quantity defining the magnitude of the internal friction in a fluid, or the Pressure resisting uniform flow.

Surface Tension. The tension on the surface of a liquid caused by the attraction of particles in the surface layer, tends to minimise surface area.

Brownian Motion. The random motion of particles suspended in a medium.

1.3 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics (SPH) stands out as a Lagrangian approach to fluid simulation, offering a dynamic method for modeling complex fluid behavior. Developed in 1977 from the work of Lucy [2] and Gingold and Monaghan [3] in astrophysics, it posed as a strong alternative to existing methods at the time. Its transformative potential was further realized in interactive liquid simulation, thanks to the efforts of Müller *et al.* [4] in 2003.

In SPH, the spatial domain is approximated into particles, each embodying various fluid properties like mass, density, and velocity. Throughout the simulation, these particles dynamically interact, forming a fluid-like continuum. Notably, the field quantities characterizing the fluid, such as pressure or velocity, can be precisely evaluated at any point in space by observing the overlapping influence spheres of individual particles. Adaptability and precision makes SPH a compelling choice for simulating fluid phenomena across a spectrum of scales and applications.

1.4 Outline and Structure

I plan to code a semi-realistic 2-D animation of a fluid in the programming language C++. This will involve describing fluid phenomena mathematically to come up with a theoretical model. I will then implement each section of the theoretical model, test its efficacy and possibly look into optimisation techniques as required. Finally to evaluate the success of my sim-

ulation I will check against the success criteria, reverting to previous methods of development if necessary.

1.5 Success Criteria

The success criteria is as follows:

- Implement all aspects of the Theoretical model within the animation where every section behaves as intended in C++.
- Have an animation of a semi-realistic 2-D incompressible fluid where the window can be resized to interact with the simulation.
- Have an animation that runs at a satisfactory speed with minimal time lag and resource wastage.

1.6 Skills

2 Research Review

Much of my research comes from GitHub, a Microsoft owned cloud-based platform for developers to store their personal or professional projects and publish them for wider use by the community. From the GitHub page of Sebastian Lague, I was introduced to articles on SPH written by many reputable institutes such as by *ETH Zurich*, *Université de Montréal* and *University College London, UK*. The majority of these papers had affiliation with *SIGGRAPH*, the international Association for Computing Machinery's Special Interest Group on Computer Graphics and Interactive Techniques. SPH related techniques are researched and published most for showcase at *SIGGRAPH* events. Through the use of google scholar and *SIGGRAPH*, I have been able to narrow my search for related documents for simulating liquids.

2.1 History and Relevant Literature

Lucy [2] introduced Smoothed Particle Hydrodynamics as a numerical testing tool for astrophysical calculations involving fission[1] within stars. This idea of quantity interpolation or "approximation" of fluid quantities was furthered by Gingold and Monaghan [3] and applied to non-spherical stars. Although both sources provide appropriate applications of this technique, the obvious limitation is that the majority is within the context of Astrophysics and not CFD. Additionally, both sources were released in 1977 with major development in the simulation field, such as the use of more modern optimisation techniques which utilise the powerful hardware now widely available, leading to the source being obsolete for present-day applicational use.

The work of Müller *et al.* [4] adapted SPH for interactive fluid applications, the first of its kind, putting forward an alternative Lagrangian method than the more common Eulerian method used for CG and modelling purposes. The paper provides a gentle introduction to SPH with a mathematical brief to the most important phenomena observed within fluids for simulation, including Pressure, Viscosity and Surface Tension. There is a distinct lack of algorithms, which leaves implementation up to the reader but the paper fulfills its purpose as an excellent introduction to SPH.

After the foundational work in 2003, Clavet *et al.* [5] release their work two years after with the primary focus on implementation, introducing key algorithms such as the Simulation step which covers the pseudocode for every frame of the animation and how the quantities of individual particles change frame by frame. Problems specific to implementation are also acknowledged, for example the near-density and near-pressure tricks are also introduced which prevent an issue that causes liquid particles to cluster.

An example of a more recent publication is Koschier *et al.* [6] in 2019. This tutorial summarises the state of SPH in its entirety by covering the theory and implementation rigorously, but also with a focus on optimisation methods to lessen compute time utilising modern hardware. The tutorial is diagrammatic and visual helping reinforce the ideas being expressed. Compared to earlier iterations covering SPH, this paper acts as the ultimate guide by placing all the information needed in one

[1] Splitting of atomic nuclei causing a release of energy

document. The paper dives much deeper into the niche complexities involved with simulating any fluids or even soft-bodied solids, but are beyond the scope of this project.

2.2 Alternative Approaches

An alternative approach for simulating fluids I have mentioned across this write-up is the Eulerian approach. Robert Bridson, in his book *Fluid Simulation for Computer Graphics* [7], perfectly encapsulates the Eulerian Viewpoint. In his words, *"The Eulerian approach, named after the Swiss mathematician Euler, takes a different tactic that's usually used for fluids. Instead of tracking each particle, we instead look at fixed points in space and see how measurements of fluid quantities, such as density, velocity, temperature, etc., at those points change in time. "*. The non-particle centric approach means the fluid is treated like a continuous medium and the simulation solves Partial Differential Equations (PDEs) to model its behaviour. Spatial domain is split up into equal sized grids and the fluid is modelled as being incompressible which would mean the total inflow within a grid must equal the total outflow. This process leads to advection within the simulation and then is rendered on screen.

Grid-based approaches have the advantage of having higher numerical accuracy and efficiency because solving PDEs can be optimised using techniques like finite difference or finite volume methods. Exactly enforcing incompressibility is important for accurate production of turbulence, and SPH methods have a hard time enforcing incompressibility efficiently. They also can have difficulty allocating computational elements throughout space efficiently. For these reasons, they have not been demonstrated to be effective for calculating flows such as air around a car, as stated per this paper by NVIDIA employees in proceedings of the 2010 ACM SIGGRAPH symposium [8].

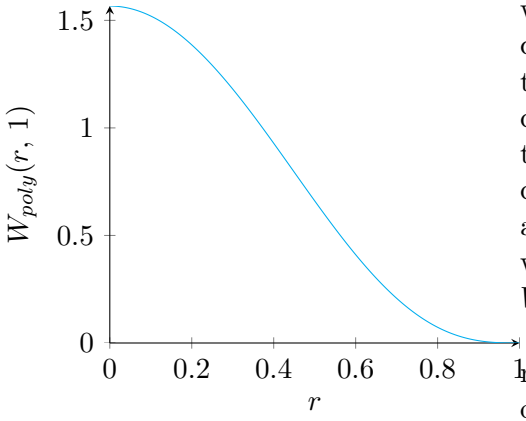
For my artefact, I aim to create a semi-realistic animations of fluids suggesting SPH is a viable technique as I do not aim for accuracy like some critical real-time systems in industry require, for example modelling airflow around rocket fuselages. Furthermore, I aim to create an element of interaction by resizing windows to show some kind of advection. "Particle-based methods like Smoothed Particle Hydrodynamics (SPH) are attractive because they do not suffer from the limitation to be inside a box" [8], also implying that resizing is impractical to implement with an Eulerian approach as resizing the window would restructure the grids that an Eulerian approach relies upon.

3 Theoretical model

3.1 Smoothing Kernel

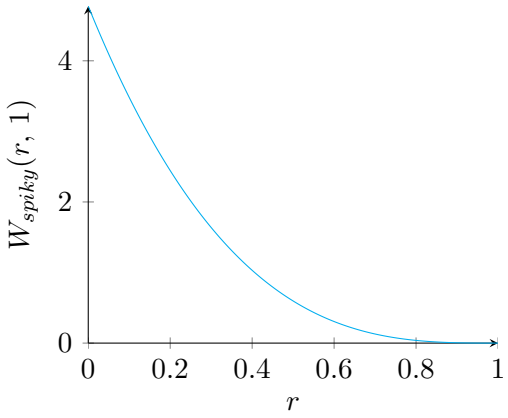
Each SPH particle has a circle of influence determined by the Smoothing Kernel $W(r,h)$ where r is the distance from the sample point to the particle center and h is the smoothing radius. The influence of a particle on a sample point increases as the sample point gets closer to the particle center. The choice of smoothing kernel is crucial as it is used by the **Interpolation Equation** to calculate scalar quantities, such as density. Müller *et al.*[4] describe two popular smoothing kernels, each with different properties.

$$W_{\text{poly}}(r,h) = \frac{315}{64\pi h^9}\begin{cases}(h^2-r^2)^3 & 0 \le r \le h\\ 0 & \text{otherwise,}\end{cases}$$



$W_{\text{poly}}(r,h)$ is a versatile kernel designed to simplify distance computations by squaring the distance term, eliminating the need for square root calculations when using the Pythagorean Theorem. However, Müller *et al.*[4] note that when $W_{\text{poly}}(r,h)$ is used for pressure computations, as required for enforcing incompressibility in this project, particles tend to cluster due to the gradient's effect on pressure calculation. The gradient of $W_{\text{poly}}(r,h)$ approaches zero for small distances, resulting in a diminishing repulsion force.

$$W_{\text{spiky}}(r,h) = \frac{15}{\pi h^6}\begin{cases}(h-r)^3 & 0 \le r \le h\\ 0 & \text{otherwise,}\end{cases}$$



$W_{\text{spiky}}(r,h)$ is used specifically for pressure computations. Its gradient is high when r is close to 0, generating the required repulsion forces for pressure calculations and therefore making it the superior choice for my simulation. However, the h^6 term is computationally challenging to calculate for every particle, therefore a more appropriate smoothing kernel for our uses would be a variation of $W_{\text{spiky}}(r,h)$ where $W(r,h) = (h-r)^3$ if $0 \le r \le h$, 0 otherwise.

For the simulation, the influence value returned will also be divided by the volume of our kernel, this keeps values consistent regardless of our values of h.

3.2 Interpolation Equation

The Interpolation Equation is responsible for calculating a scalar quantity A at a location r by a weighted sum of contributions from all the particles within our simulation. It sits at the heart of this project. This equation will be used to calculate density

and viscosity which will indicate the pressure and therefore the net force a particle observes.

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

A_s is the property we want to calculate,
m_j is the mass of the particle,
A_J is the value of that property of particle with index j,
ρ_j is the local density,
$W(r - r_j, h)$ is the value of the smoothing kernel with the distance between the 2 particles being $r - r_j$. [4]

3.3 SPH gradient optimisation

The simulation step updates the position vectors of the particles according to the rate of change of the properties every frame. We can calculate this rate of change by taking the derivative of our smoothing kernel with respect to r:

$$W(r, h) = (h - r)^3$$
$$\frac{\partial W}{\partial r} = -3(h - r)^2$$

3.4 Pressure Forces and Newton's Third Law

Clavet *et al.* [5] mention calculating a pseudo-pressure P_i, proportional to the difference between the local density ρ_i and the current density ρ_0, governed by the equation $P_i = k(\rho_i - \rho_0)$, where k is a constant, which will be referred to as the pressure multipler, governing the stiffness of the fluid. We then apply the SPG gradient optimisation to get a value for the pressure force. By Newton's second law, $F = ma$, we can find the acceleration of the particle and apply this to the particles within the simulation step. Furthermore, Newton's Third Law of Motion must also be applied in this context, where if a particle exerts a pressure force, it must experience an equal and opposite reaction force.

3.5 Viscosity

The nature of this project would lead you to assume that viscosity is calculated using the interpolation equation. However, Koschier *et al.*[6] mention there are major issues with this approach. The most significant issue is that this approach is sensitive to particle disorder. An proposed alternative which is viable for the scope of this project is to use "one derivative using SPH and the second one using finite differences" [6]. Following from Sebastian Lague's implementation, the viscosity force is calculated by taking the difference in velocities, multiplying by the influence from a viscosity kernel and then multiplying by an arbitary scalar value. For my implementation, I will settle for the smoothing kernel for viscosity for simplicity instead of the suggested sharper viscosity kernel as I am cautious of overall compute time.

3.6 Predicted Position optimisation

An interesting paper by B. Solenthaler and R. Pajarola [9] uses the current velocities of particles to determine a predicted position and uses predicted positions in the density and pressure calculations instead of current particle positions. This Predictive-Corrective Incompressible SPH (PCISPH) model allows for greater enforcement on incompressibility whilst having low computational cost per update with a large timestep, which is useful as other methods of enforcing incompressibility rely on smaller timesteps that are computationally heavy.

4 Development and Testing

4.1 Boilerplate code

The beginning of my implementation involved the boilerplate minimal code required for my graphics library SFML to be setup as well as including key libraries I will need throughout the project.

```
#include <SFML/System/Vector2.hpp>
#include <SFML/Graphics.hpp>
#include<iostream>
#include <algorithm>
#include<vector>
#include<cmath>
#include <omp.h>

int main()
{
    //Initialize SFML
    sf::RenderWindow
        window(sf::VideoMode(900,
        900), "Smoothed Particle
        Hydrodynamics Simulation");
    window.setFramerateLimit(120);
    sf::View view =
        window.getDefaultView();
    sf::Vector2u window_size =
        window.getSize();

    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type ==
                sf::Event::Closed)
                window.close();
            if (event.type ==
                sf::Event::Resized){
                sf::FloatRect
                    visibleArea(0.f,
                    0.f,
                    event.size.width,
                    event.size.height);
                window.setView(sf::View(
                visibleArea));
            }
        }
        sf::Vector2u window_size =
            window.getSize();
        window.clear();
        window.display();
    }
    return 0;
}
```

4.2 Particle initialization

We can begin adding particles using instances of a Particle struct which stores the information each particle needs. This includes position and velocity vectors, the circular shape, local densities and pressures and their predicted position.

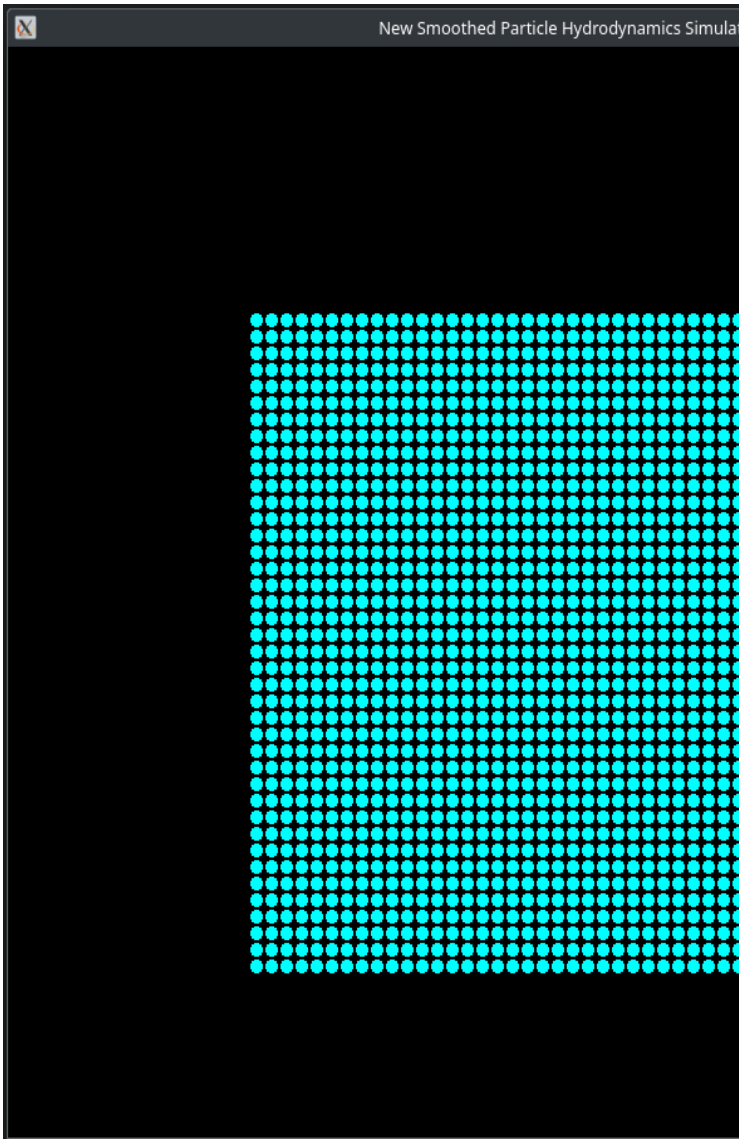
```
struct particle{
    sf::CircleShape
        droplet{particle_radius};
    sf::Vector2f position{0.f, 0.f};
    sf::Vector2f velocity{0.f, 0.f};
    float local_density = 0.f;
    float local_pressure = 0.f;
    sf::Vector2f
        predicted_position{0.f, 0.f};
};
```

We initialize a dynamic array of these particles and place them in an orderly fashion at the start of the simulation, as well as set their colour to cyan.

```
const int particle_num = 1600;
const float particle_mass = 1.f;
vector<particle>
    particles(particle_num);

void placeParticles(){
    int particlesPerRow =
        sqrt(particle_num);
    int particlesPerColumn =
        (particle_num -
        1)/particlesPerRow + 1;
    int spacing = 2.5*particle_radius;
```

```
    for (int i = 0; i < particle_num;
        i++){
        particles[i].position.x = (i %
            particlesPerRow +
            particlesPerRow / 2.5f
            +0.5f) * spacing;
        particles[i].position.y = (i /
            particlesPerRow +
            particlesPerColumn / 2.5f
            +0.5f) * spacing;
        particles[i].droplet
        .setFillColor(sf::Color::Cyan);
    }
}
```



4.3 Simulation loop

In order to calculate new positions of particles, we need to loop over them every step of the simulation to calculate its new position vector using its current velocity and acceleration vectors and multiplying by dt, our change in time per simulation step and then draw them. We can begin updating these by implementing gravity first.

```
const float gravity = 9.81f;
const float dt = 1.f/60.f;

 void resolveGravity(int i){
    particles[i].velocity.y += gravity
        * dt
}
```

And within the simulation loop:

```
for (int i = 0; i < particle_num;
    i++){
    resolveGravity(i);
    particles[i].position.x +=
        particles[i].velocity.x * dt;
    particles[i].position.y +=
        particles[i].velocity.y * dt;
    particles[i].droplet
    .setPosition(particles[i]
    .position.x-particle_radius,
    particles[i].position.y-particle_radius);

    //Draw particles
    window.draw(particles[i].droplet);
```

This video shows the implementation of gravity but with no simulation border. This can be implmented with ease in the following manner:

```
void resolveCollisions(int i,
    sf::Vector2u window_size){
    if (particles[i].position.x >
        window_size.x ||
        particles[i].position.x < 0){
        particles[i].position.x =
            clamp((int)particles[i].position.x,
            0, (int)window_size.x);
        particles[i].velocity.x *= -1
            * collision_damping;
    }
    if (particles[i].position.y >=
        window_size.y ||
        particles[i].position.y <= 0){
        particles[i].position.y =
            clamp((int)particles[i].position.y,
            0, (int)window_size.y);
        particles[i].velocity.y *= -1
            * collision_damping;
    }
}
```

I've also implemented a collision damping factor here to simulate the loss the energy upon collision with the simulation border. We now see that we can also successfully interact with the simulation by resizing our window, hitting one of the sucess criteria for this project. The video can be watched here.

4.4 Predicted Position Optimisation

The predicted position of the particles, which will be used throughout the development, need to be calculated first. This is done quite easily in this manner:

```
void predictPositions(int i){
    particles[i].predicted_position.x
        = particles[i].position.x +
        particles[i].velocity.x * dt;
    particles[i].predicted_position.y
        = particles[i].position.y +
        particles[i].velocity.y * dt;
}
```

The function simply predicts the position of the particle depending on the current velocity.

4.5 Smoothing Kernel and derivative

As per the theoretical model, my preferred choice of smoothing kernel is the customised version of $W_{\text{spiky}}(r, h)$, allowing for large repulsion forces at small distances and less computationally strenuous powers. In C++, this looks like the following:

```
float smoothingKernel(double dst){
    float volume = pi *
        (float)(pow(smoothing_radius,
```

```
        4.0)/2);
    if (dst >=smoothing_radius){
        return 0.f;
    }
    float value =
        (float)pow(smoothing_radius -
        dst, 3.0);
    return value/volume;
}
```

Notice how we return 0 if the distance is further than the smoothing radius, this saves time as unnecessary values are not computed. The influence value is also divided by the volume of the function, giving us similar influence values regardless of the smoothing radius.

Below is the implementation of the smoothing kernel derivative used for optimisation discussed in the theoretical model.

```
float
    smoothingKernelDerivative(double
    dst){
    if (dst >= smoothing_radius)
        return 0.0;
    float value = -6.f *
        (smoothing_radius - dst) *
        (smoothing_radius - dst) / (pi
        * pow(smoothing_radius, 4));
    return value;
}
```

4.6 Density Calculations

Calculating the local densities for each point requires us to use the SPH Interpolation equation, $A_s(r) = \sum_j m_j \frac{A_j}{\rho_j} W(r - r_j, h)$ [4]. Interestingly, if we want to calculate the density at a certian point, substituting for ρ gives us:

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

Where the densities cancel out, giving us a method of calculating the local density without being dependant on it.
Coding this in C++ gives us:

```
float calculateDensity(int i){
    float density = 0.f;
    for (int j =0; j < particle_num;
        j++){
        float dst = sqrt();
        //distance is calculated using
            predicted positions
        float influence =
            smoothingKernel(dst);
        density += particle_mass *
            influence;
    }
    return density;
}
```

4.7 Pressure Calculations

Quantifying the local density into a pressure value, per my theoretical model, would involve taking the local density and finding a pseudo-pressure value, $P_i = k(\rho_i - \rho_0)$ [5], and applying the gradient optimisation to turn this pseudo-pressure into pressure.

The code below is the implementation of the equation above, where k is the pressure multiplier:

```
float densityToPressure(int j){
    float density_error =
        particles[j].local_density -
        target_density;
    float local_pressure =
        density_error *
        pressure_multiplier;
    return local_pressure;
}
```

Then, in order to apply Newton's 3rd law, the index of the 2 particles which are being computing is taken and the average of their pseudo-pressures is returned:

```
float sharedPressure(int i, int j){
```

```
    float pressurei =
        densityToPressure(particles[i]
    .local_density);
    float pressurej =
        densityToPressure(particles[j]
    .local_density);
    return (-(pressurei + pressurej) /
        2.f);
}
```

Notice how a negative pressure value is returned, this ensures pressure is a purely repulsive force which makes intuitive sense as the particles must repel each other.

Finally, the pseudo-pressure is converted to a pressure force using the SPH gradient optimisation which takes into account the rate of change of this property as well as the x and y direction of the particle to turn the scalar pressure force into a vector. Including the nested loop, the implementation of this is below:

```
sf::Vector2f
    calculatePressureForce(int i){
    sf::Vector2f pressure_force;
    sf::Vector2f viscosity_force;
    for (int j = 0; j<particle_num;
        j++){
        if (i == j) continue;
        float x_offset;
        float y_offset;
        if (particles[i].
        predicted_position ==
            particles[j].
        predicted_position){
            x_offset = (float)1+
                (rand() % 20);
            y_offset = (float)1+
                (rand() % 20);
        }
        else{
            x_offset =
                particles[j].predicted_position.x
                -
                particles[i].predicted_position.x;
            y_offset =
                particles[j].predicted_position.y
                -
                particles[i].predicted_position.y;
        }
        double dst = sqrt(abs(x_offset
            * x_offset + y_offset *
            y_offset));
        float gradient =
            smoothingKernelDerivative(dst);
        float x_dir = x_offset/dst;
        float y_dir = y_offset/dst;
        float shared_pressure =
            sharedPressure(i, j);
        pressure_force.x +=
            shared_pressure * gradient
            *
            particle_mass/particles[j].local_density
            * x_dir;
        pressure_force.y +=
            shared_pressure * gradient
            *
            particle_mass/particles[j]
        .local_density * y_dir;
    }
    return pressure_force;
}
```

Notice how a random direction is picked to apply the force if two particles have the same position, this contributes to the particles' Brownian motion.

The returned pressure force must now be converted to an acceleration. This can be done in the main simulation loop by calling the calculatePressureForce function and dividing the value by the density, as per Sebastian Lague's implementation [1] of an SPH solver. He mentions this is due to the particles being spheres of volume instead of mass. The implementation within the simulation loop looks like this:

```
sf::Vector2f pressure_force =
    calculatePressureForce(i);
sf::Vector2f pressure_acceleration;
pressure_acceleration.x =
    pressure_force.x/particles[i]
```

```
.local_density;
pressure_acceleration.y =
    pressure_force.y/particles[i]
.local_density;
```

4.8 Viscosity

As calculating viscosity requires a weighted sum from every other particle within the smoothing radius, just like pressure, a nested loop can be used to achieve this. From my theoretical model, implementation in C++ looks like the following:

```
sf::Vector2f
    calculateViscosityAcceleration(int
    i){
    sf::Vector2f
        viscosity_acceleration;
    float dst;
    for (int j; j<particle_num; j++){
        if (i==j) continue;
        float dst;
        //dst is calculated using
            predicted positions
        viscosity_acceleration.x -=
            (particles[i].velocity.x -
            particles[j].velocity.x) *
            (smoothingKernel(dst)) *
            viscosity_strength;
        viscosity_acceleration.y -=
            (particles[i].velocity.y -
            particles[j].velocity.y) *
            (smoothingKernel(dst)) *
            viscosity_strength;
        if (viscosity_acceleration.x >
            0 ||
            viscosity_acceleration.y >
            0){
            viscosity_acceleration.x =
                0;
            viscosity_acceleration.y =
                0;
        }
    }
    return viscosity_acceleration;
}
```

The acceleration caused by the viscosity within a fluid slows the particle down, this is because viscosity is caused by friction within a fluid, a force which opposes motion. This is why the acceleration caused by viscosity is not added, but rather subtracted from the overall acceleration of the particle.

4.9 Interactivity and visuals

One of the criterias for this project is interactivity. Using SFML, I have allowed the user to resize the window but ImGui will allow the user to use sliders and change key SPH properties discussed in this paper such as the Target density, Pressure multiplier and the smoothing radius. Additionally, I will also allow for gravity, number of particles and particle mass to also be changed to allow the user freedom and to explore different types of fluid which are possible to simulate using this technique. ImGui interactive variable initialisation code is below:

```
ImGui::Begin("Menu");
ImGui::SliderInt("Particle Num",
    &particle_num, 1, 1600);
ImGui::SliderFloat("Particle Mass",
    &particle_mass, 0.1f, 100.f);
ImGui::SliderFloat("Target Density",
    &target_density, 0.f, 500.f);
ImGui::SliderFloat("Pressure
    Multiplier", &pressure_multiplier,
    0.f, 1000.f);
ImGui::SliderFloat("Smoothing
    Radius", &smoothing_radius, 10,
    200);
ImGui::SliderFloat("Gravity",
    &gravity, 0.f, 100.f);
ImGui::SliderInt("Framerate",
    &framerate, 60, 1200);
ImGui::End();
```

In terms of the visuals of the project, cyan particles were being rendered earlier. Instead, a more aesthetically pleasing and useful algorithm to implement would be a linear interpolation colour algorithm. Simply put, the algorithm would output a colour for the particle depending on it's velocity. A larger velocity would output a more red-shifted colour whereas a slower velocity would output a blue-shifted colour. In C++, we would have:

```
void resolveColor(int i, float vel){
    int b = clamp((int)(-255/50 * vel
        + 255), 0, 255);
    int r = clamp((int)(255/50 * vel
        -255), 0, 255);
    int g = clamp((int)(-abs(255/50 *
        (vel-50))+255), 0, 255);
    particles[i].droplet.setFillColor
    (sf::Color(r, g, b));
}
```

This will be run in the simulation loop after the velocities have been calculated for all particles in that frame. The intention behind this is to be able to better explain densities by visualising them, as well as evaluate any anomalies within the simulation.

4.10 Updating Simulation loop

Finally, we can accumulate update our simulation loop by putting together everything within our development so far. This is the updated simulation loop:

```
for (int i = 0; i < particle_num;
     i++){
    resolveCollisions(i, window_size);
    //gravity step
    resolveGravity(i);
    //Predict next positions
    predictPositions(i);
    // window bounding box
    //calculate densities
    particles[i].local_density =
        calculateDensity(i);
    //convert density to pressure
    particles[i].local_pressure =
        densityToPressure(i);
    //Calculate pressure forces and
        acceleration
    sf::Vector2f pressure_force =
        calculatePressureForce(i);
    sf::Vector2f pressure_acceleration;
    pressure_acceleration.x =
        pressure_force.x/particles[i].local_density;
    pressure_acceleration.y =
        pressure_force.y/particles[i].local_density;
    //Calculate acceleration due to
        viscosity
    pressure_acceleration +=
        calculateViscosityAcceleration(i);
    //calculate velocity
    particles[i].velocity.x +=
        pressure_acceleration.x * dt;
    particles[i].velocity.y +=
        pressure_acceleration.y * dt;
    //resolve colour
    float vel =
        sqrt(particles[i].velocity.x *
        particles[i].velocity.x +
        particles[i].velocity.y *
        particles[i].velocity.y);
    resolveColour(i, vel);
    //calculate particle positions
        with radius offset
    particles[i].position.x +=
        particles[i].velocity.x * dt;
    particles[i].position.y +=
        particles[i].velocity.y * dt;
    //set particle position on screen
    particles[i].droplet.setPosition
    (particles[i].position.x+particle_radius,
        particles[i].position.y+particle_radius);
    //render particle
    window.draw(particles[i].droplet);
}
```

4.11 Final Product

5 Evaluation and Final Remarks

References

[1] S. Lague, "Coding adventure: Simulating fluids," 2023. [Online]. Available: https://github.com/SebLague/Fluid-Sim

[2] L. B. Lucy, "A numerical approach to the testing of the fission hypothesis," *Astronomical Journal, vol. 82, Dec. 1977, p. 1013-1024.*, vol. 82, pp. 1013–1024, 1977.

[3] R. A. Gingold and J. J. Monaghan, "Smoothed particle hydrodynamics: theory and application to non-spherical stars," *Monthly notices of the royal astronomical society*, vol. 181, no. 3, pp. 375–389, 1977.

[4] M. Müller, D. Charypar, and M. Gross, "Particle-based fluid simulation for interactive applications," in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Citeseer, 2003, pp. 154–159.

[5] S. Clavet, P. Beaudoin, and P. Poulin, "Particle-based viscoelastic fluid simulation," 2005. [Online]. Available: http://www.ligum.umontreal.ca/Clavet-2005-PVFS/pvfs.pdf

[6] D. Koschier, J. Bender, B. Solenthaler, and M. Teschner, "Smoothed particle hydrodynamics techniques for the physics based simulation of fluids and solids," 2019. [Online]. Available: https://github.com/InteractiveComputerGraphics/SPH-Tutorial/blob/master/pdf/SPH_Tutorial.pdf

[7] R. Bridson, *Fluid simulation for computer graphics*. AK Peters/CRC Press, 2015.

[8] J. M. Cohen, S. Tariq, and S. Green, "Interactive fluid-particle simulation using translating eulerian grids," in *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, 2010, pp. 15–22.

[9] B. Solenthaler and R. Pajarola, "Predictive-corrective incompressible sph," *University of Zurich*, [Accessed May 19, 2024].

[10] C. Braley and A. Sandu, "Fluid simulation for computer graphics: A tutorial in grid based and particle based methods," *Virginia Tech, Blacksburg*, 2010.

[11] N. S. Division, "Mach contours showing plume-induced flow seperation after solid rocket booster separation at high altitude." 2020, [Online; accessed February 14, 2024]. [Online]. Available: https://www.nas.nasa.gov/SC09/Ares_V_backgrounder.html

[12] S. Tavoularis, *Measurement in Fluid Mechanics*. Cambridge University Press, 2005.