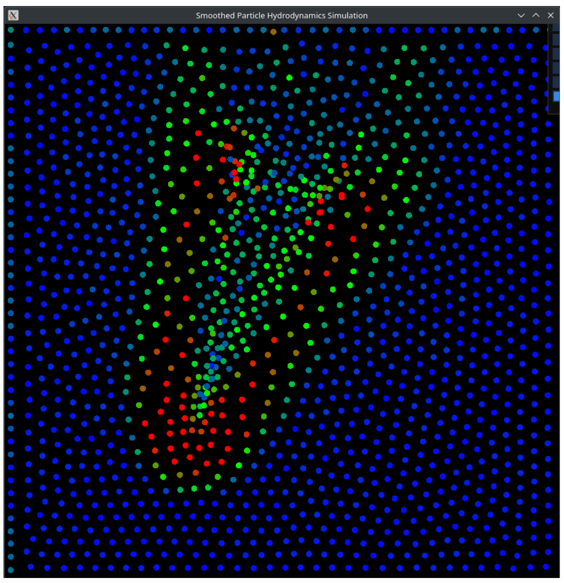
Simulating Fluid Motion using Smoothed Particle Hydrodynamics

Aayush Joshi

Abstract

This paper covers a method of computational fluid dynamics known as Smoothed Particle Hydrodynamics. Within the introduction, the paper presents the motivations for this project, a brief introduction to SPH and outlines the success crtieria for the artefact. In the research review, the paper provides a history of the SPH techniques as well as alternative CFD approaches considered in industrial applications such as Aerospace. The theoretical model outlines and elaborates on a mathematical brief for each section of SPH and Development shows the implementation of each section of the theoretical model in C++. Development also outlines the added interactive elements using SFML and ImGui. Finally, the evaluation measures the success of the overall project to the criteria outlined within the introduction.



Final interactive SPH implementation.

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 graphics 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 its implementation in industry. 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.

CPU. The Central Processing Unit (CPU) is the primary component of a computer that acts as its “control center”.

GPU. The Graphics Processing Unit (GPU) is a specialized electronic circuit designed for digital image processing and to accelerate computer graphics calculations.

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.

Debug.	The process of finding and fixing errors (bugs) in source code.
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.
Kinetic energy.	
Intermolecular Forces.	
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.
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 simulation 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:

- Research SPH and popular alternative methods of fluid simulation.
- Develop a simulation which solves the Navier Stokes Equations and therefore can be classed as a fluid.
- Have an animation that runs with minimal lag and resource wastage.
- Add window resizing, mouse forces and sliders for user interaction.
- Ensure the simulation can respond to user interaction.

1.6 Skills

- Read and understand academic publishings and research papers in a Mathematics and/or Computer Science background.
- Be able to take key information and synthesize it to create my own model.
- Typeset my written work in LaTeX, a document formatting tool.
- Learn to use a new programming language, C++.
- Learn to add interactive elements to projects in C++ using SFML and ImGui.
- Test and debug code in C++.
- Manage my source code using version control software, Git/Hub.
- Manage my time and changes in timescales.

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]. It was hypothesised at the time that fission caused formation of close binary star systems [2].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 an 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,

[1]Splitting of atomic nuclei causing a release of energy
[2]A system of 2 stars orbiting the other

this paper acts as the ultimate guide by placing all the information needed in one 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].

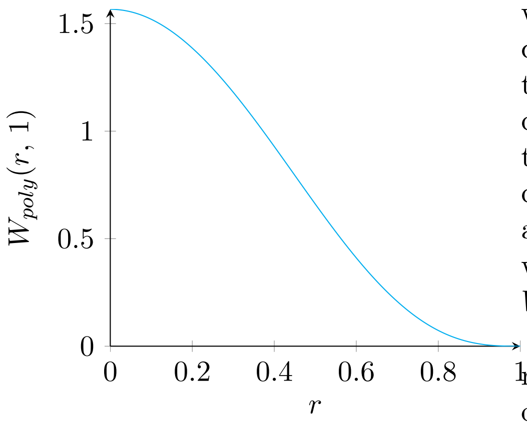
For my artefact, I aim to create a semi-realistic animations of fluids suggesting SPH is a viable technique as I do not aim for accuracy like some critical real-time systems in industry require, for example modelling airflow around rocket fuselages. Furthermore, I aim to create an element of interaction 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

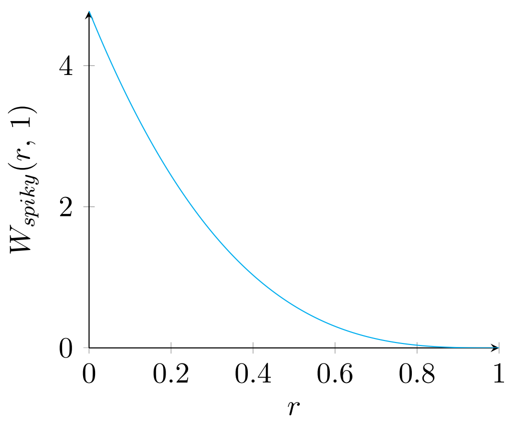
Each SPH particle has a circle of influence determined by the Smoothing Kernel $W(r,h)$ where r is the distance from the sample point to the particle center and h is the smoothing radius. The influence of a particle on a sample point increases as the sample point gets closer to the particle center. The 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

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 for now. Refer to Figure 1 in Appendix B.

```
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);
```

Appendix B Figure 2 (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. Refer to Appendix B Figure 3 (video).

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 computed 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 casued 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. Finally, using SFML's in-built support for mouse features, I have coded a mouse force function as shown below. The program adds the appropriate acceleration within the main simulation loop in the next section.

```
sf::Vector2f
    interactiveForce(sf::Vector2i
    position, int i, float repulsive){
    sf::Vector2f force;
    float dst =
        (particles[i].position.x -
        position.x) *
        (particles[i].position.x -
        position.x) +
        (particles[i].position.y -
        position.y) *
        (particles[i].position.y -
        position.y);
    if (dst<=10000){
        force +=
            (((sf::Vector2f)position)
            -
            particles[i].predicted_position)
            * (100-sqrt(dst))/4.f *
            repulsive;
    }
    return force;
}
```

The program looks for a left or right click within the events loop.

```
if (event.type ==
    sf::Event::MouseButtonPressed){
    interactive = true;
    //Determine whether it is an
        attractive or repulsive force
    if (event.mouseButton.button ==
        sf::Mouse::Left){
        repulsive = 1.f;
    }
    else{
        repulsive = -1.f;
    }
}
//Stop the interactive forces is the
    mouse button is released
if (event.type ==
    sf::Event::MouseButtonReleased){
    interactive = false;
}
```

The intention behind this is to be able to better explain fluids moving down the pressre gradient 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);
   // Add the mouse forces
   if (interactive){
       pressure_force +=
           interactiveForce(position,
           i, repulsive);
   }
   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

The simulation clearly solves each section of the Navier-Stokes equation through SPH. Figure 4 in Appendix B shows the fluid moving down the pressure gradient from an area of high pressure at the start of the simulation, to areas of low pressure as the simulation border is extended. By sucessfully attempting to move down this pressure gradient, the fluid constantly attempts to achieve constant density and therefore incompressibility. This is visible in Figure 5 where the simulation eventually reaches a stable state with particles having no/minimal velocities (shown by their blue colour) and arranging themselves in an orderly manner. Viscosity is harder to show but as the particles move from high to low pressure, red particles with unusually high velocities bounce back from the border and match the velocities of the particles around them, turning green and eventually blue as seen in Figure 6. This also helps the fluid reach a constant density throughout the simulation space.

As for interactivity, the window resizing has worked sucessfully since the start of this project and can be noticed throughout development. Figure 7 in Appendix B shows the sliders on screen which control different aspects of the simulation as discussed, available for the user to control the behaviour of the fluid. The ultimate test of interactivity is the mouse forces, which can be seen in Figure 8. Attractive and repulsive forces are working with left and right click respectively with dragging. The particles around them adjust for change in density appropriately.

5 Evaluation and Final Remarks

5.1 Final artefact outcome

The first success criteria was to research SPH and popular alternative methods of fluid simulation. From my research review, I think it is evident that this criteria was fulfilled. My initial introduction to Lague [1] and his sources led me to discover SPH and how CFD is handled within industry. I initially used Youtube to gain a high-level understanding of SPH but quickly realised the amount of detail which videos glossed over when explaining SPH. Sebastian Lague's sources on his github page prompted me to use Google Scholar to search for Müller's [4] introductory work. This was naturally the most useful source in terms of theoretical understanding of SPH and therefore extremely useful when developing the theoretical model. Koschier [6] and Clavet [5] provided more algorithmic approaches and were more useful in the development part of the project. Per my Research review, I have also considered the alternative Eulerian approach to simulating fluids. Although more precise, Bridson [7] stated it is less suitable for interactive implementations as the resizing the window would reshuffle the grids which this grid-based approach relies upon.

The second objective on my sucess criteria is to develop a simulation which solves the Navier-Stokes equations and can be classed as a fluid. Referring to the figures in Appendix B, it is evident the simulation solves every term of the Navier-Stokes equation. Namely, moving down the pressure gradient, constant density, viscosity and gravity. Interestingly, the pressure multipler ends up controlling how gas-like or fluid-like the simulation is. Clavet [5] mention that the pressure multiplier acts as a stiffness constant and this is obvious in my implementation as a higher pressure multiplier makes the simulation more stiff, portraying more liquid-like behaviour in my SPH implementation as particles exert more intermolecular forces. A lower pressure multiplier has smaller intermolecular forces and therefore the particles are more gasseous. Interestingly, an extremely high pressure multiplier leads to the simulation behaving more like a soft bodied solid, like jelly. This is similar to the outcome mentioned by Clavet [5]. The simulation is highly sucessful at striving to obtain a constant density. This was best evident upon adding mouse forces to the simulation because a repulsive force creates additional space, which the particles rush to fill once the force has been removed. This is similar to the particles moving down the pressure gradient when the window is resized to add additional space. The particles end up in a stable position with no velocity in an orderly fashion.

Viscosity and gravity has a lower level of success than initially anticipated. Although viscosity does contribute to the particles velocity ending up being the same as its neighbours, this is better achieved by a larger pressure multiplier. Gravity is also not well incorporated within the simulation. The best results are achieved by imagining the simulation is looking at a sample of submerged fluid or a top-down view of a container containing this fluid with gravity set to zero. If I attempt to simulate the fluid as water within a glass with gravity, for the same settings, the fluid ends up being very chaotic as seen in the Appendix B Figure 9. Although still semi-realistic, the problem is the particles at the bottom seem to gain kinetic energy, as if they are boiling or are representing a heavy-gas, while particles at the top seem unphased. One way around this is to reduce the particle number. This introduces its own problem though. The simulation becomes less accurate for a lower particle number. The aim is to be able to render as many particles as possible and in order to have fluid in a glass simulation, I have to half my particle number to 800 particles. The issue created here is the surface at the top does not end up smooth, but ends up becoming jagged as seen in Figure 10.

It is fair to say the animation runs at with minimal lag and resource wastage. During early stages of development, I did have to ensure that all my calculations were being done in as few loops as possible. Currently, my programs has 2 loops, one nested within another. This, as I found during my development, was the limit for an interactive simulation. The addition of further loops caused the program to start slowing down and become unresponsive. With this in mind, I limited my calculations to this structure. Upon adding mouse forces, the program also started to slow down and this was due to how SFML handles mouse button down events. I did find that interactive elements worked best implemented within the events loop rather than the main simulation loop, which retrospectively is expected as those events are designed to be handled within that loop.

As for interactive elements, it is evident throughout Appendix B that window resizing has worked flawlessly since the start. As for mouse forces and the linear colour algorithm, these were elements I was able to add over the summer due to the extension of the project deadline. Their outcome has been extremely sucessful as the mouse forces have helped me test the simulation response to user interaction. SPH is designed for interaction and this was evident upon implementing the mouse forces. Before the extension of the deadline, I struggled to change fluid settings due to the lack of sliders with key fluid variables. Over the summer, I made sure to implement sliders seen in Figure 7. These allowed me to change variables to values whilst the simulation is running, saving me time spent debugging or recompiling otherwise. They also help demonstrate concepts such as constant density and forcing fluids to move down pressure gradients. I found this useful, especially during my presentation, as they helped visualise concepts to the audience. The colour also adds a layer of aesthetic investigation and intrigue to the user. Any additional time over the summer was spent perfecting the interactive elements to ensure the simulation responds well to any user interaction, hitting the last two points of my sucess criteria.

5.2 Improvements

A major improvement which I would make to my artefact would be adding optimisations to render more particles. This would solve most issues with my project so far. As stated in the last section, the effects of viscosity are not immediately obvious but a larger particle number would mean the smoothing radius of more particles overlap with each other, increasing the influence experienced by each particle. As per my code, this would increase the effect of viscosity overall leading to a noticeable effect on screen. When it comes to gravity, seen in Figure 10, a larger particle number would even out the jagged surface which is unrealistic in practice. This would increase the realism of the simulation as a liquid in a glass. Attempting to improve viscosity further would add a surface tension-like effect as well, as seen in Lague's [1] implementation, leading to a layer of particles attracted to each other. Interestingly, container pressure or surface tension effect is observed within the animation, refer to Figure 8. On the simulation border, we see an unusual adhesion the particles have to the window border, causing a repelling force. In reality, this property arises from a known disadvantage of SPH when handling boundary cases. This effect will appear to be smoothed out with a larger number of particles.

One method of optimising my code to be able to add more particles would be to develop the simulation in an existing physics engine like Unity or Unreal Engine 5. These game engines are designed to be able to run simulation physics to a very high standard and contain proprietary optimisation techniques. Moreover they would help with compute-heavy calculations such as distance calculations and standardise units into kilograms or meters. Currently, my program runs on a pixel-to-pixel basis, where one unit of distance is a pixel. This means the dimension analysis [3] of my units for pressure or density are not correct. Physics engines often ease such aspects of computer graphics to allow developers to focus on more complex simulation mechanics.

One final improvement I would make to my artefact would be to turn the simulation into a compute shader. Currently, my program runs on the CPU. Turning it into a compute shader would involve major changes to the code such that it can be run on the GPU. Although this would involve its own research and would be the equivalent of undertaking another EPQ, it is certainly an improvement I would consider making to understand more about GPU mechanics and how it can be utilised more effectively for simulation. Parallel processing power of the GPU would be put to test when using the fluid simulation for modelling fluid flow around or within objects for more accurate and realistic results, an extension worth considering for aerospace applications.

5.3 Skills

Reading and understanding academic publishings and research papers in a Mathematics and/or Computer Science background was initially challenging. Although the formal language posed a small barrier, the mathematical notation was more nuanced and took longer to parse. Most papers included a key which set out the notation and I found myself referring to it constantly when first researching SPH. This inspired me to create a definitions section within my write-up to help everybody understand the language used in this paper more thoroughly with minimal ambiguity. Reading such published work is a skill I feel much more confident with now and am proud to have learned as my work at university will undoubtedly use this to a great extent.

Being able to create my theoretical model based upon existing research is another skill I have clearly accomplished. I am able to justify every section within my theoretical model and have sucessfully applied the research to customise certain sections of SPH, such as the proposed compute-power heavy smoothing kernel versus my implemented gentle alternative.

Typing up my write-up in LaTeX, a

[3] Analysis of base units of a physical quantity

document formatting tool which I have used to compile my pdf, has eased the process of integrating mathematical formulae into published work. From the theoretical model and development sections of this write-up, it is evident this had been very sucessful. I can say with experience this would have been arduous on alternative document formatting tools such as Word or Onenote. Furthermore, LaTeX is preferred at university for mathematics and is used by students to take digital notes as well as submit assignments with instead of easily misplaced or misunderstood handwritten work. I have extended my knowledge of LaTeX by being able to add plots, seen in the theoretical model, code, as seen in development and images, as seen in Appendix B. This skill will be invaluable for me at university.

A big motive of taking this project was also to learn C++, a new language with which I was not initially familiar. C++ is often the preferred language due to the vast amount of support it has in many software disciplines but especially due to its fast execution time which is crucial with simulation. Learning C++ was time consuming at the start, I found myself constantly referring to W3schools [10], a website with tutorials to learn new concepts within programming languages. This would be for simple tasks such as outputting values to the screen or to understand the cause of simple syntax errors. I also spent a lot of time on the SFML documentation site [11] to understand how to setup and use SFML and ImGui. Due to my experience with Python [4], I was perhaps able to draw parallels quicker than someone learning the language from scratch. My familiarity with the testing and debugging process fastracked error resolving overall. Now after finishing my artefact, I feel more confident using C++ on a day-to-day basis in my Computing A-level at school or eventually at university for projects.

In order to manage my code and allow data synchonisation between my laptop and PC, I used Github. Github is a cloud based service which allows developers to store their source code to ease with development version control. It is a framework built on top of git, a version control software. Github provides a graphical interface allowing me to more easily manage coding over multiple devices as well as publish code to the wider audience for everybody to use. It thoroughly streamlined synchonisation and allowed me to save time when trying to work on the project at school and work. It is a tool widely used in the IT industry and a tool I can confidently use in a corporate environment.

One of the biggest skills I have managed to develop is my time management. At the start of the EPQ, I consistently spent 2 hours weekly on Thursdays and Fridays. I found it was easier researching SPH at school and developing at home where I had access to my PC. Adhering to this schedule meant the development of density and pressure forces was very quick due to the well conducted research and well planned-out theoretical model which outlined the main mathematical elements to be coded. Interestingly, the half of my timescale is extremely accurate but the second half is not very accurate at all. This is likely due to additional commitments during the latter parts of Year 12 such as mocks as well as the extension of the deadline. This allowed me to refine the artefact to include viscosity and add interactive elements other than

[4]Another programming language

window resizing, resulting in a more polished implementation overall. Learning to adapt to sudden changes in timescales and change the theoretical model appropriately is a skill undertaking an EPQ has taught me.

5.4 Presentation

For my presentation, I decided on displaying my artefact by setting up my PC in school. I felt that having the simulation on display helped attract attention especially due to its aesthetics with the linear colour algorithm and interactions with the mouse. Having the artefact present also helped illustrate my points on the inner workings of SPH when presenting with the slideshow on my laptop. Showing how different sections of SPH solve the Navier-Stokes equations, such as resizing the window to show fluids moving down the pressure gradient, provided a better intuitive sense of my artefact and many complemented me for this visualisation. Generally, I felt my pace when explaining sections of the simulation was good and I was able to field questions when asked. The most common included “Why did you choose this project”, to which I responded with how CFD perfectly overlaps with my A level subjects and how it is a field I intend to explore further. Some SPH and artefact specific questions included “Why does the pressure multipler control gas or liquid behaviour?”, as this slider seemed to interest most people. I responded by explaining how the pressure multiplier controls the stiffness of the fluid and quantifies intermolecular forces between these particles. Another question was “Why do the particles stick to the border and repel particles within the window” to which I honestly answered by elaborating on the known disadvantage of SPH when handling boundary and edge cases. The biggest success was observed when people interacted with the fluid using the mouse forces or sliders themselves. This was anticipated by me when the project deadline was extended and I decided on adding interactive elements that it would be put to most use during the presentation.

6 Bibliography

[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] W3Schools. [Online]. Available: https://www.w3schools.com/cpp/default.asp

[11] S. documentation. [Online]. Available: https://www.sfml-dev.org/

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

[13] 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

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

Appendices

A C++ code

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

using namespace std;

static int particle_num = 1600;
static float particle_mass = 30.f;
const int particle_radius = 5;
const float collision_damping = 0.8f;
const float pi = 3.141f;
static float target_density = 100.f;
static float pressure_multiplier = 75.f;
static float smoothing_radius = 70.f;
const float dt = 1.f/60.f;
static float gravity = 0.f;
static int framerate = 60;
static int viscosity_strength = 10.f;
static bool interactive = false;

struct particle{
    sf::CircleShape droplet{particle_radius};
    sf::Vector2f position{0.f, 0.f};
    sf::Vector2f velocity{0.f, 0.f};
    float local_density = 1.f;
    float local_pressure = 1.f;
    sf::Vector2f predicted_position{0.f, 0.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;
    }
}

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

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;
}

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);
    return value/volume;
}
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;
}
float calculateDensity(int i){
    float density = 0.f;
    for (int j =0; j < particle_num; j++){
        float dst = sqrt((particles[j].predicted_position.x -
            particles[i].predicted_position.x) *
            (particles[j].predicted_position.x -
            particles[i].predicted_position.x) +
            (particles[j].predicted_position.y -
            particles[i].predicted_position.y) *
            (particles[j].predicted_position.y -
            particles[i].predicted_position.y));
        float influence = smoothingKernel(dst);

        density += particle_mass * influence;
    }
    return density;
```

```
}

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

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);
}

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;
        // Newton's 3rd law implementation below
        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;
}

sf::Vector2f calculateViscosityAcceleration(int i){
    sf::Vector2f viscosity_acceleration;
    float dst;
    for (int j; j<particle_num; j++){
        if (i==j) continue;
        dst = sqrt((particles[i].predicted_position.x -
            particles[j].predicted_position.x) *
            (particles[i].predicted_position.x -
            particles[j].predicted_position.x) +
            (particles[i].predicted_position.y -
            particles[j].predicted_position.y) *
            (particles[i].predicted_position.y -
            particles[j].predicted_position.y));
        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;
}

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;
    }
    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;
    }
}

void resolveColour(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));
}

sf::Vector2f interactiveForce(sf::Vector2i position, int i, float repulsive){
    sf::Vector2f force;
    float dst = (particles[i].position.x - position.x) *
        (particles[i].position.x - position.x) + (particles[i].position.y -
        position.y) * (particles[i].position.y - position.y);
    if (dst<=10000){
        force += (((sf::Vector2f)position) - particles[i].predicted_position) *
            (100-sqrt(dst))/4.f * repulsive;
    }
    return force;
}

int main()
{
    //Initialize SFML
    sf::RenderWindow window(sf::VideoMode(900, 900), "Smoothed Particle
        Hydrodynamics Simulation");
    window.setFramerateLimit(framerate);
    sf::View view = window.getDefaultView();
    sf::Vector2u window_size = window.getSize();
    placeParticles();
    ImGui::SFML::Init(window);
    sf::Vector2i position;
    float repulsive = 1;
    sf::Clock deltaClock;
    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            ImGui::SFML::ProcessEvent(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));
            }
            if (event.type == sf::Event::MouseButtonPressed){
                interactive = true;
                if (event.mouseButton.button == sf::Mouse::Left){
                    repulsive = 1.f;
                }
                else{
                    repulsive = -1.f;
                }

            }
            if (event.type == sf::Event::MouseButtonReleased){
                interactive = false;
            }
        }
        sf::Vector2u window_size = window.getSize();
        ImGui::SFML::Update(window, deltaClock.restart());
```

```
window.clear();

//ImGui Menu
ImGui::Begin("Menu");
ImGui::SliderInt("Particle Num", &particle_num, 1, 1600);
ImGui::SliderFloat("Particle Mass", &particle_mass, 10.f, 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::End();
sf::CircleShape circle;
position = sf::Mouse::getPosition(window);
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);
    if (interactive){
        pressure_force += interactiveForce(position, i, repulsive);
    }
    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);
    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);// add at the end
    //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);
        }
        ImGui::SFML::Render(window);
        window.display();

    }
     ImGui::SFML::Shutdown();

    return 0;
}
```

B Media

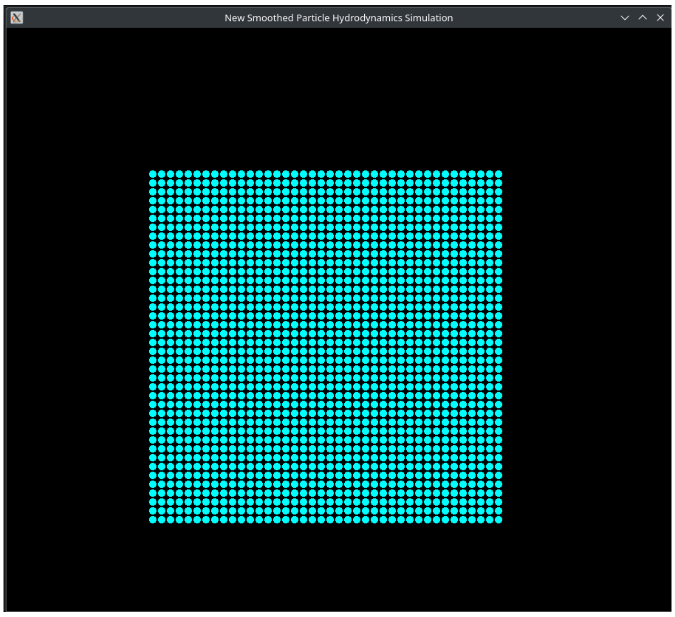


Figure 1: Cyan particles rendered on screen.

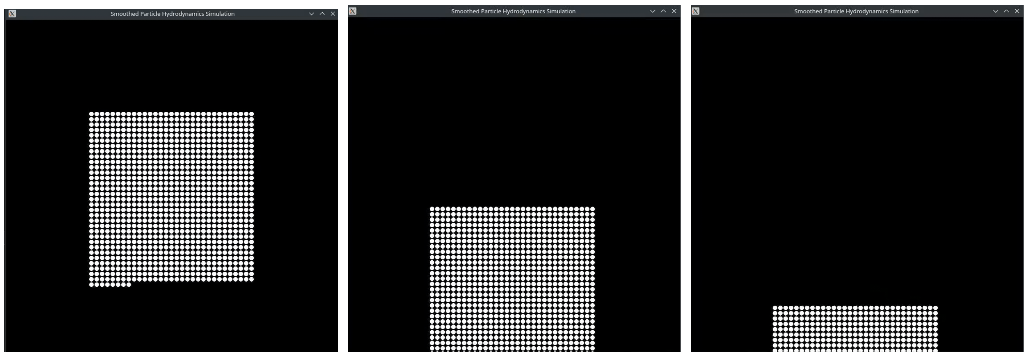

Smoothed Particle Hydrodynamics Simulation
Smoothed Particle Hydrodynamics Simulation
Smoothed Particle Hydrodynamics Simulation

Figure 2: Gravity with no collision with screen border.

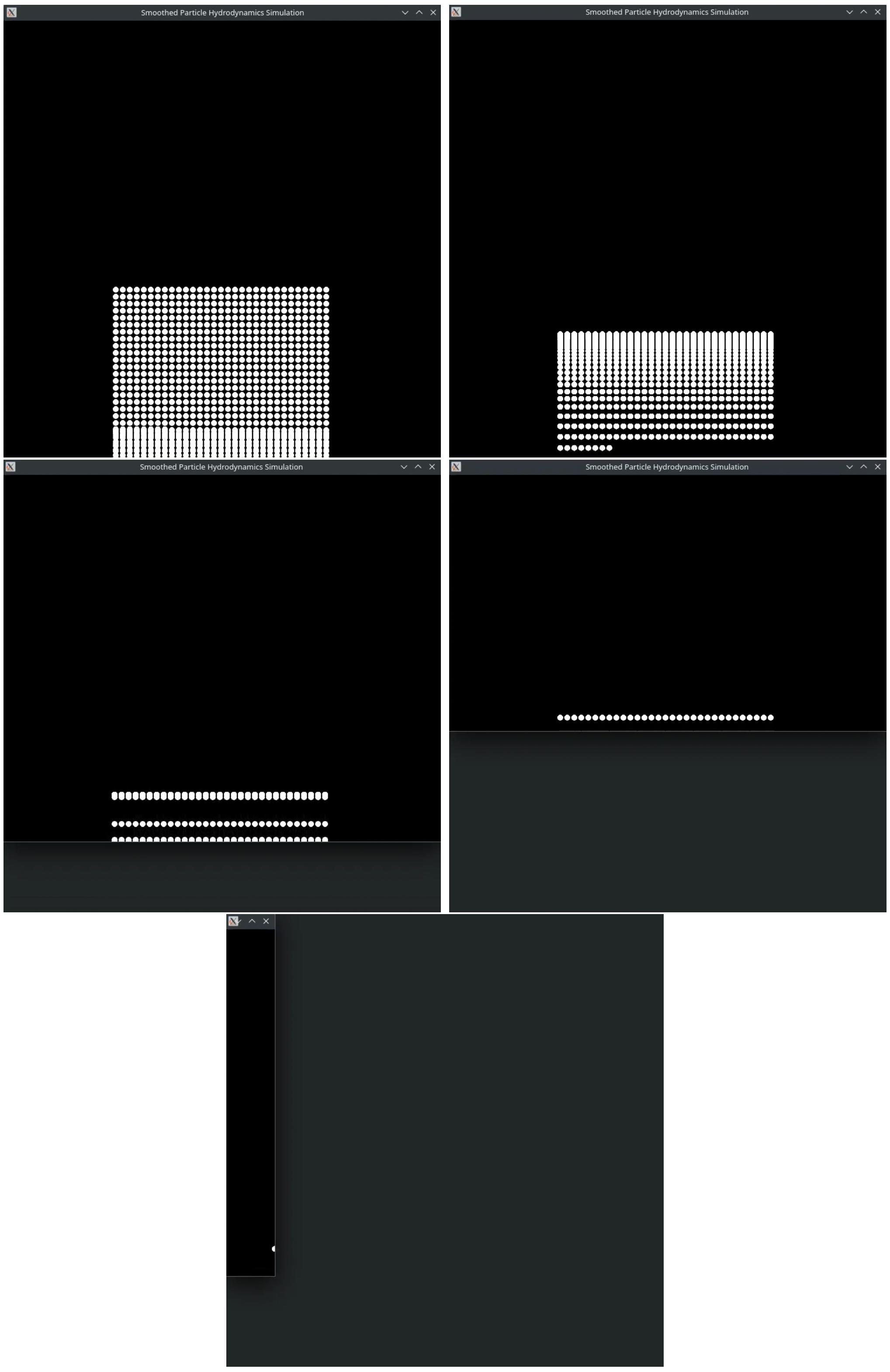
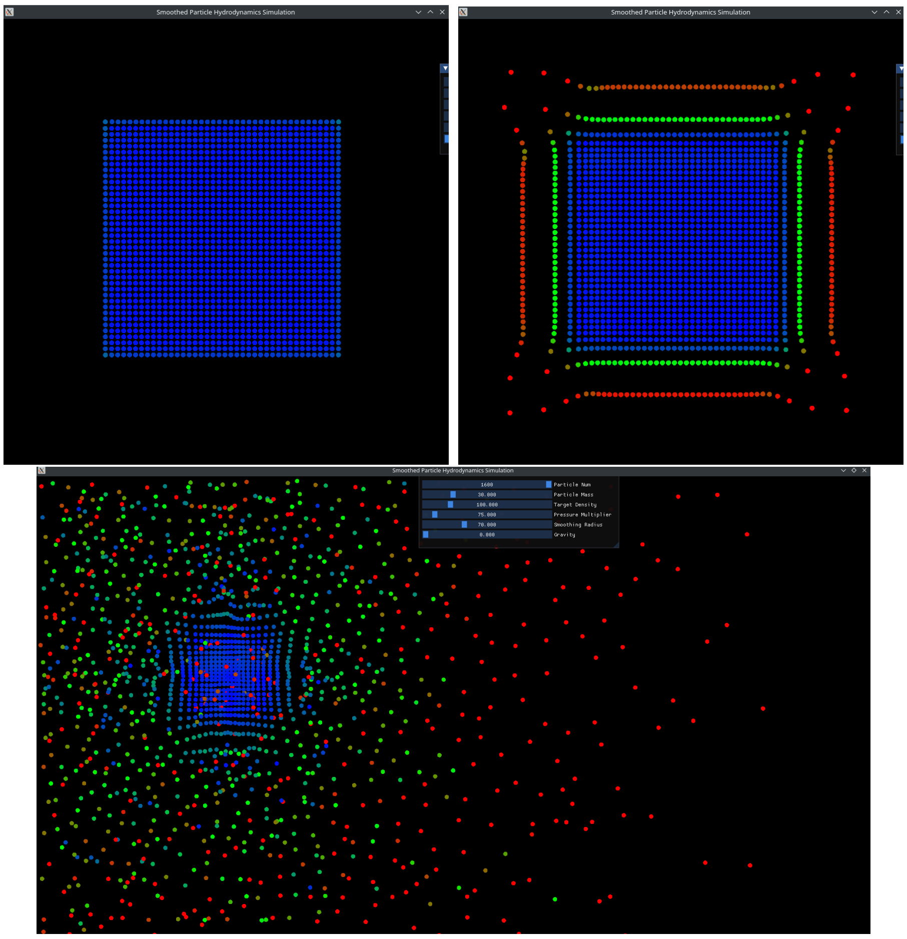
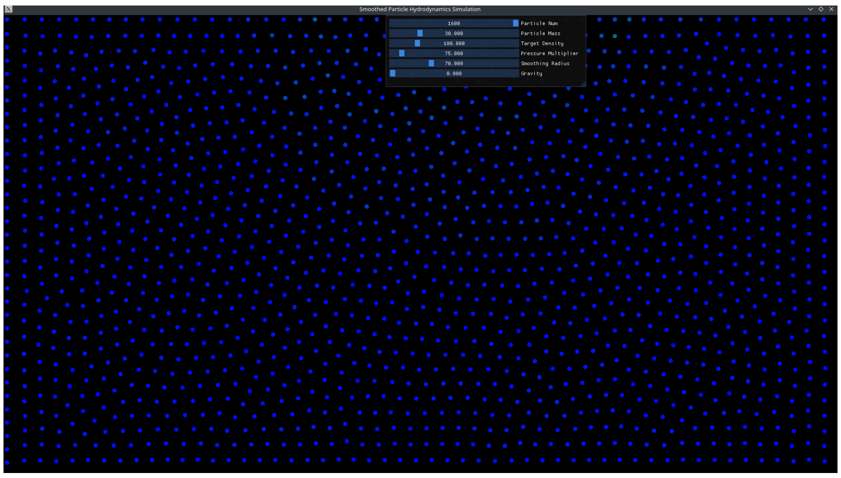

Smoothed Particle Hydrodynamics Simulation
Smoothed Particle Hydrodynamics Simulation
Smoothed Particle Hydrodynamics Simulation
Smoothed Particle Hydrodynamics Simulation

Figure 3: Gravity with collision and window resizing.


Smoothed Particle Hydrodynamics Simulation
Smoothed Particle Hydrodynamics Simulation
Smoothed Particle Hydrodynamics Simulation

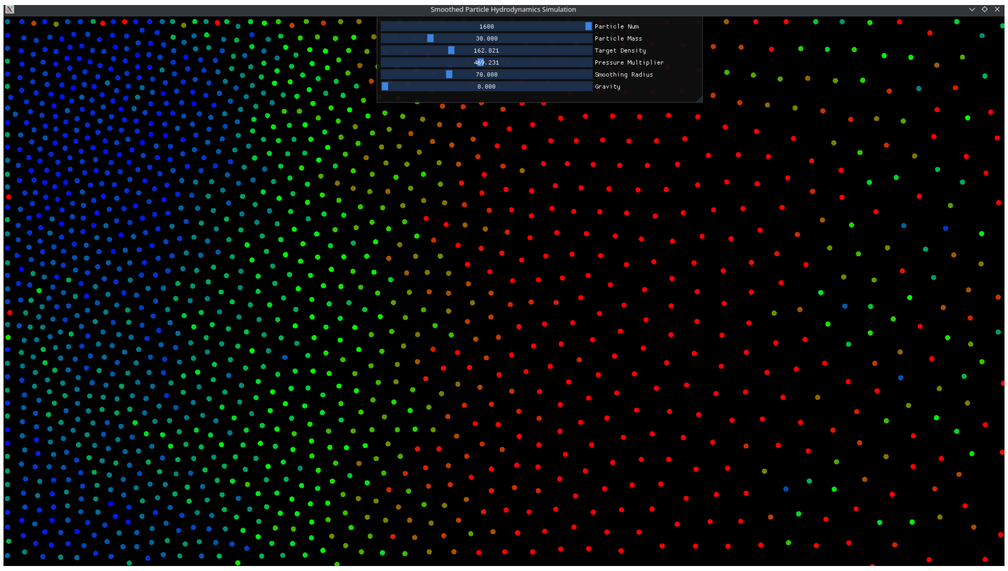
```
1600      Particle Num
30.000    Particle Mass
100.000   Target Density
75.000    Pressure Multiplier
70.000    Smoothing Radius
0.000     Gravity
```

Figure 4: Fluid particles moving down the pressure gradient.



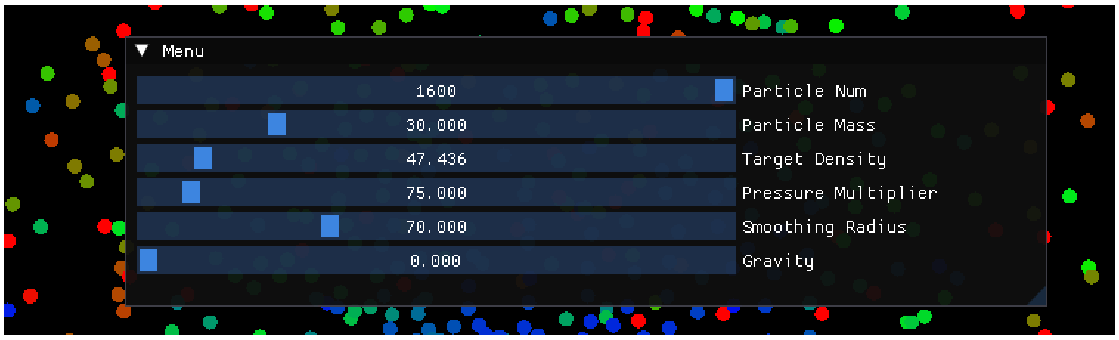
```
1600      Particle Num
30.000    Particle Mass
100.000   Target Density
75.000    Pressure Multiplier
70.000    Smoothing Radius
0.000     Gravity
```

Figure 5: Fluid particles reaching a constant density in a stable state.



```
1600      Particle Num
30.000    Particle Mass
162.821   Target Density
469.231   Pressure Multiplier
70.000    Smoothing Radius
0.000     Gravity
```

Figure 6: Red particles with high velocities becoming green at the edge, displaying viscosity



```
1600      Particle Num
30.000    Particle Mass
47.436    Target Density
75.000    Pressure Multiplier
70.000    Smoothing Radius
0.000     Gravity
```

Figure 7: ImGui sliders on screen.

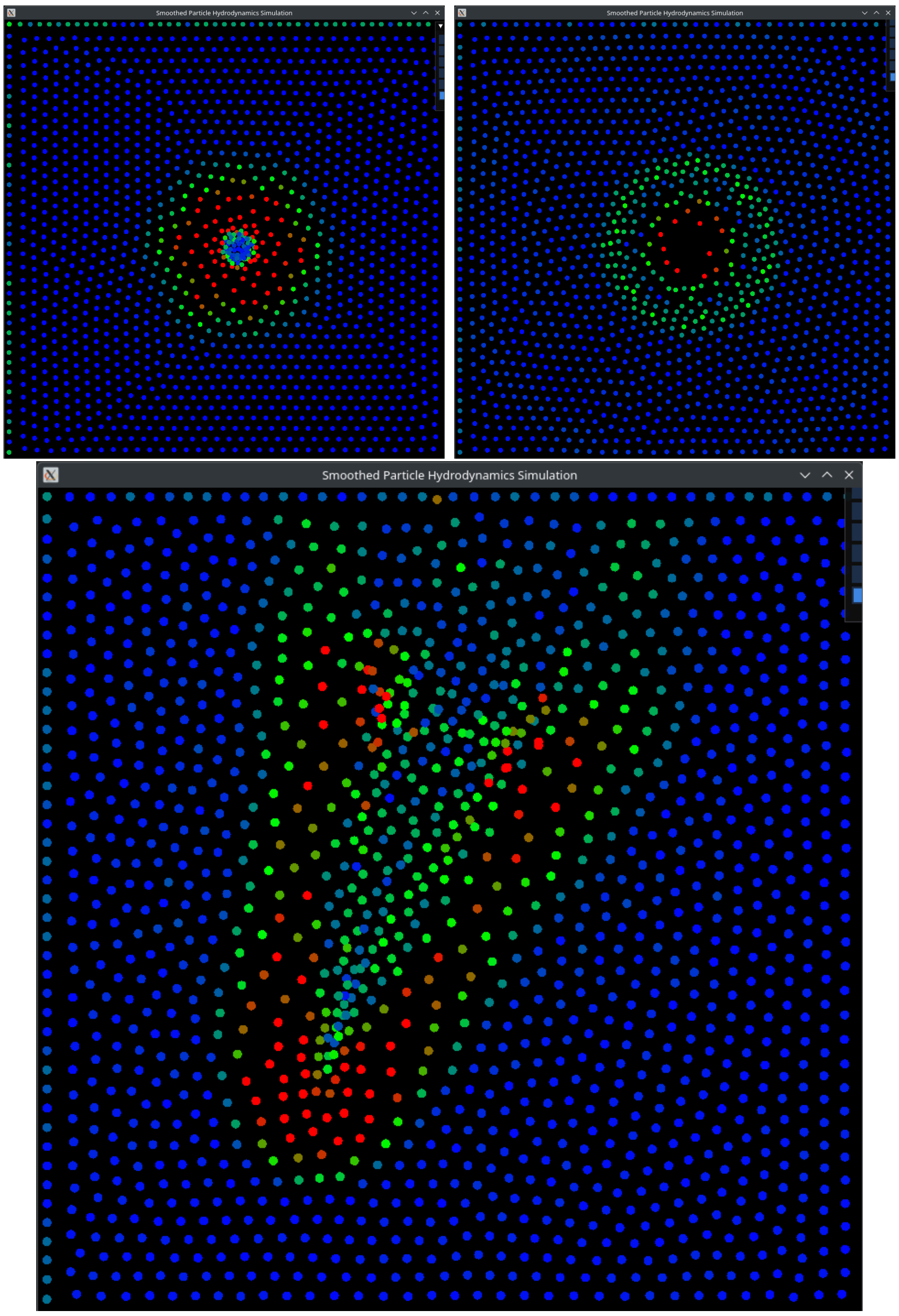

Smoothed Particle Hydrodynamics Simulation
Smoothed Particle Hydrodynamics Simulation
Smoothed Particle Hydrodynamics Simulation

Figure 8: Attractive and repulsive mouse forces with mouse drag effect.

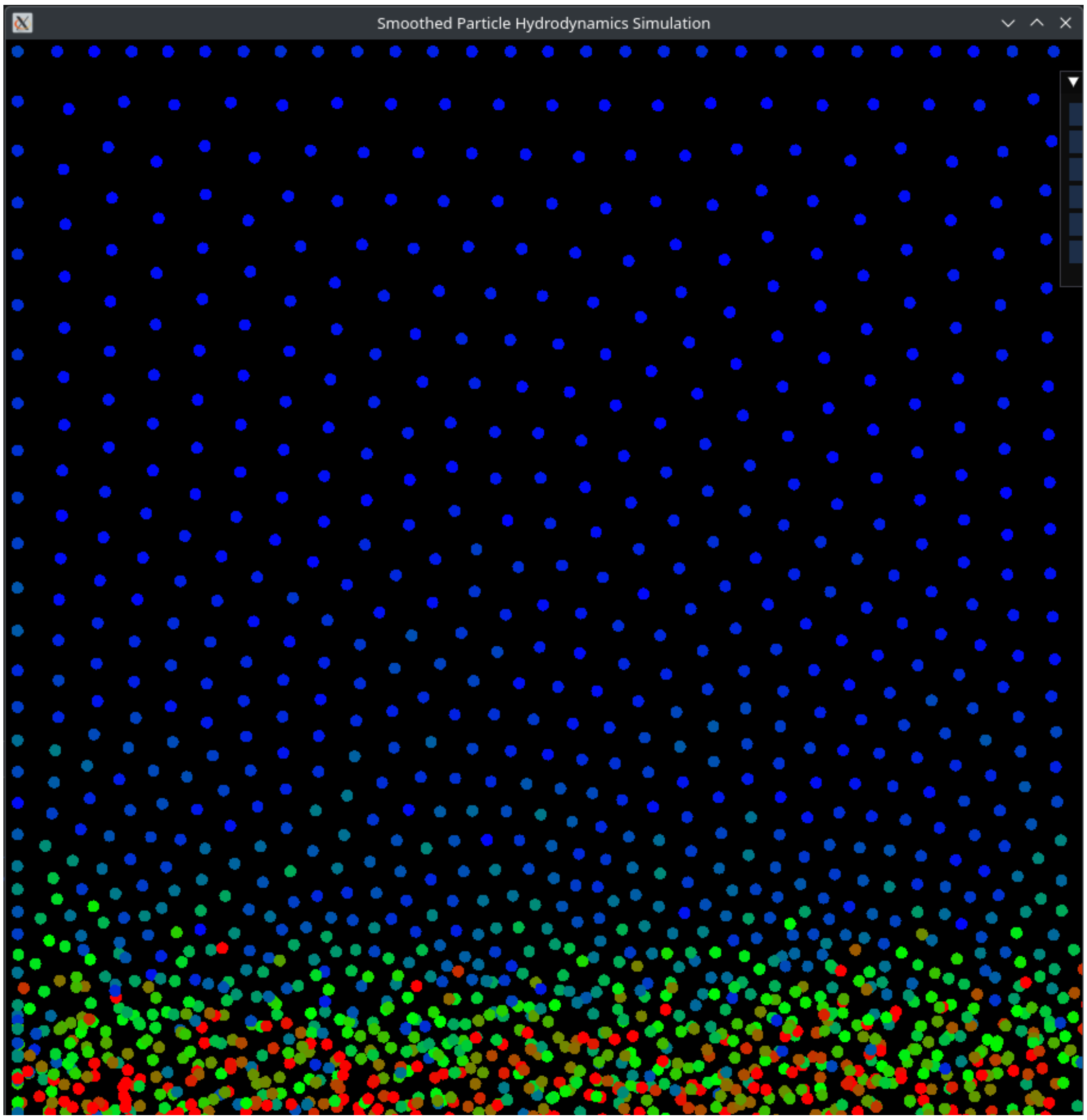
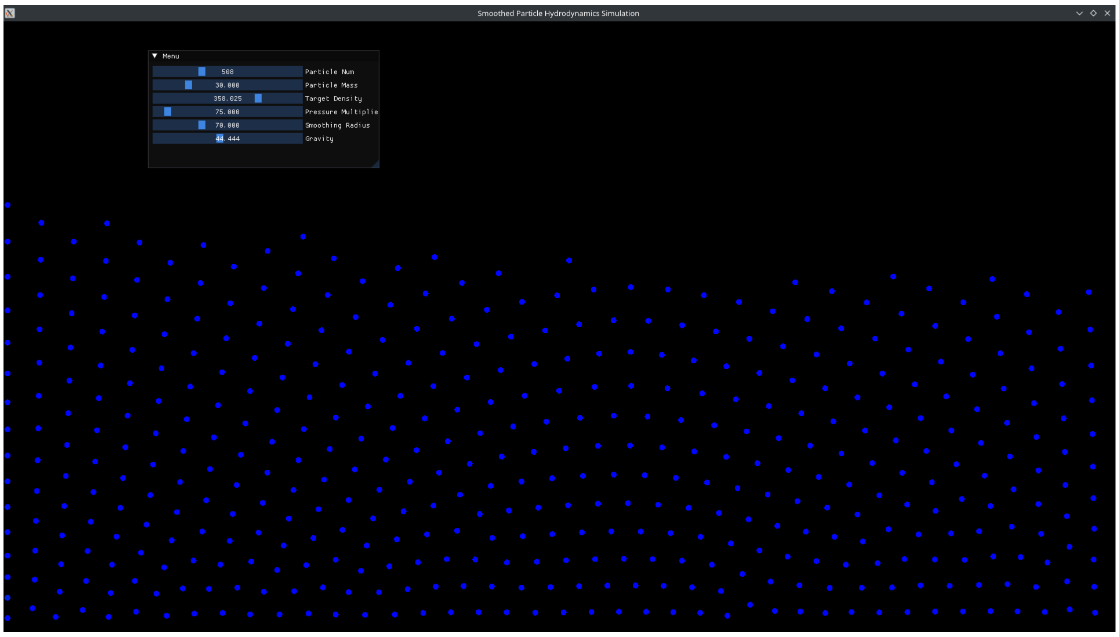


Figure 9: Fluid with gravity behaving like a boiling liquid or heavy gas.



508	Particle Num
30.000	Particle Mass
358.025	Target Density
75.000	Pressure Multiplie
70.000	Smoothing Radius
44.444	Gravity

Figure 10: Fluid with gravity giving an unsmooth surface.