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

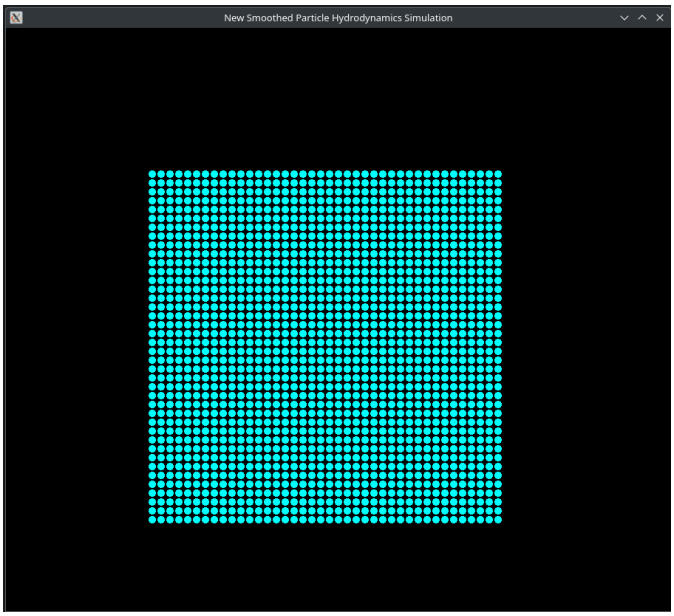


Figure 1: Cyan particles rendered on screen.

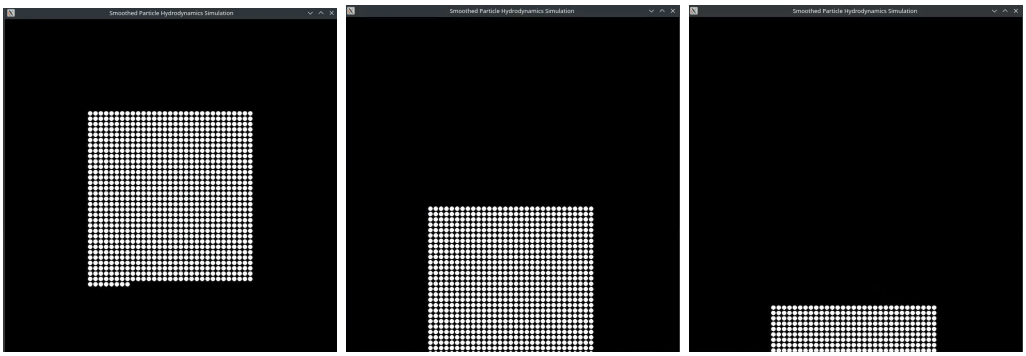


Figure 2: Gravity with no collision with screen border.

C Presentation slides

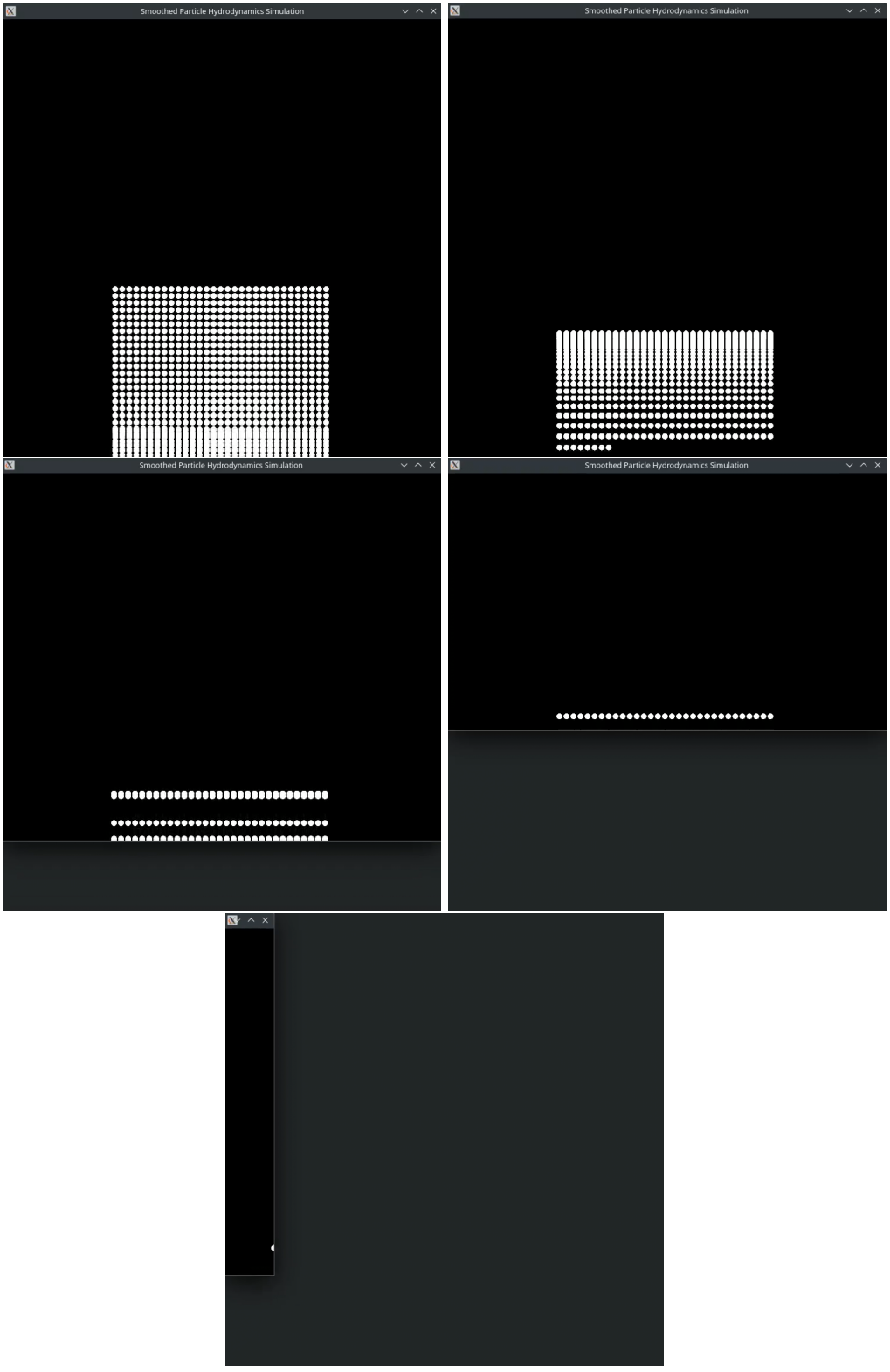


Figure 3: Gravity with collision and window resizing.

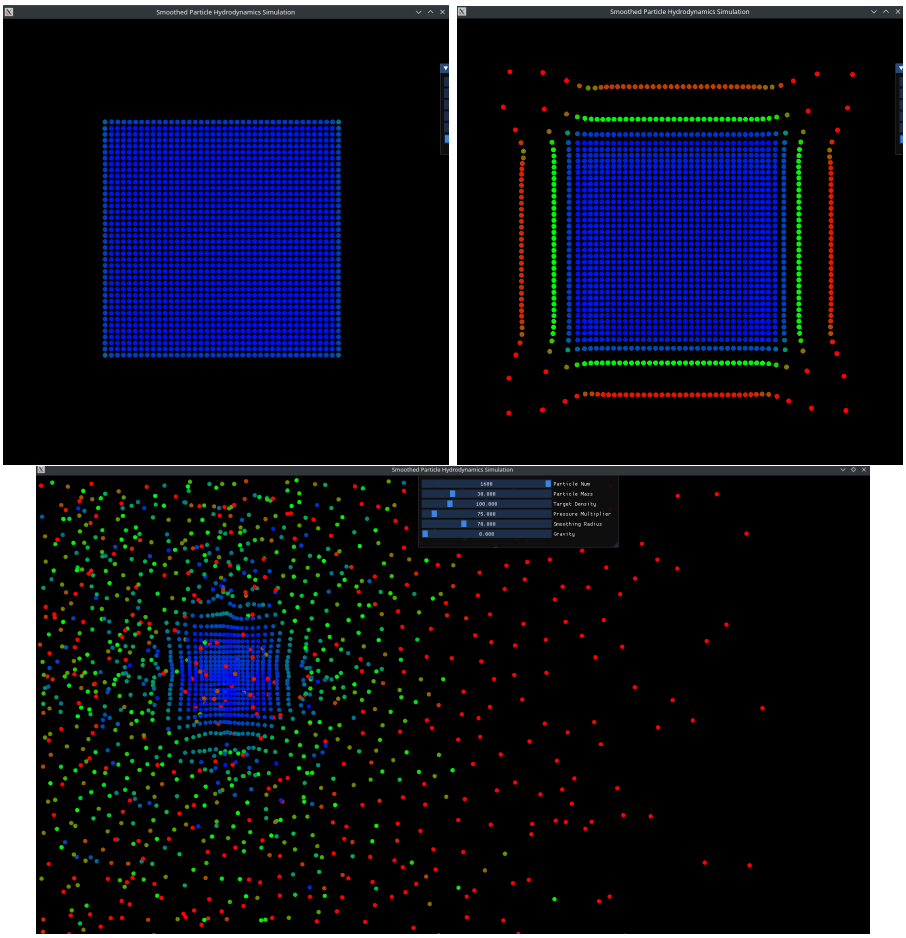


Figure 4: Fluid particles moving down the pressure gradient.

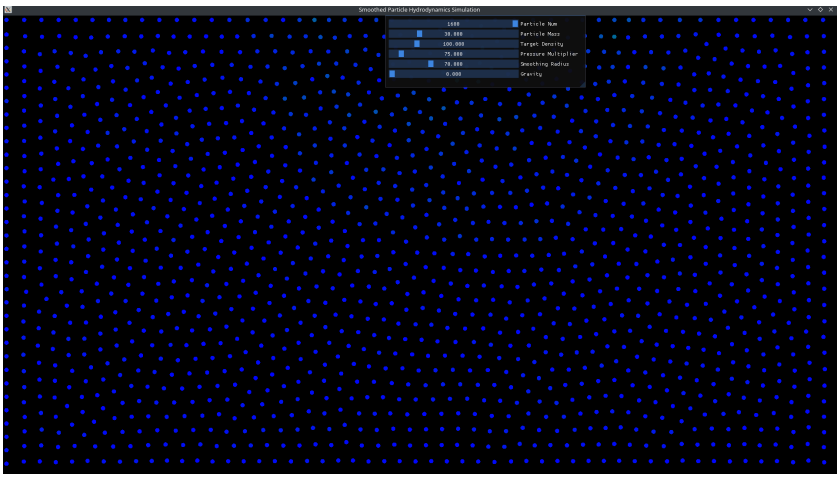


Figure 5: Fluid particles reaching a constant density in a stable state.

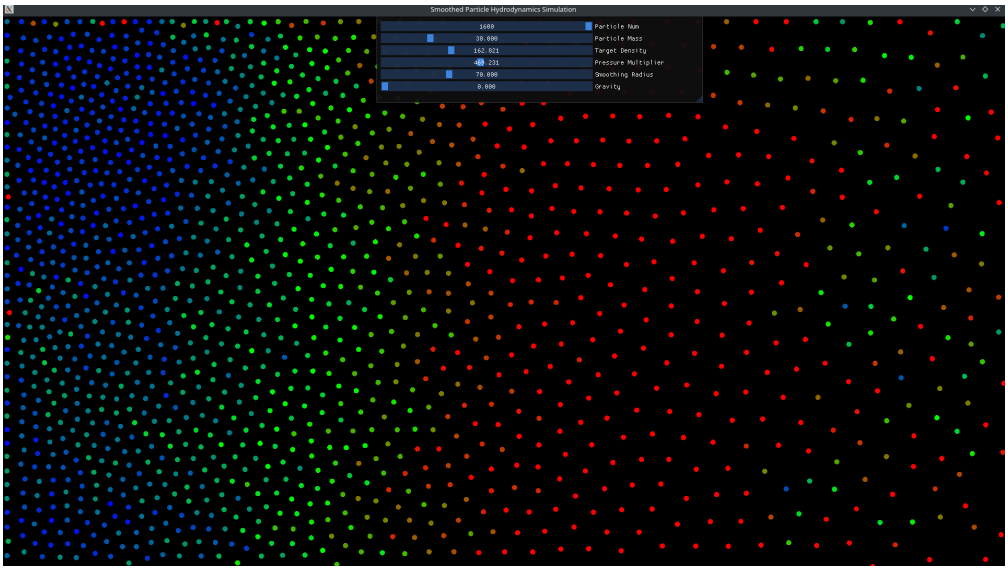


Figure 6: Red particles with high velocities becoming green at the edge, displaying viscosity

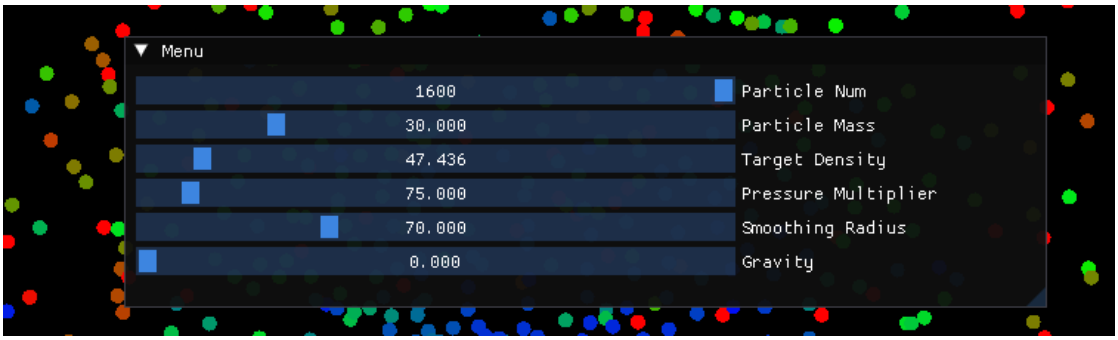


Figure 7: ImGui sliders on screen.

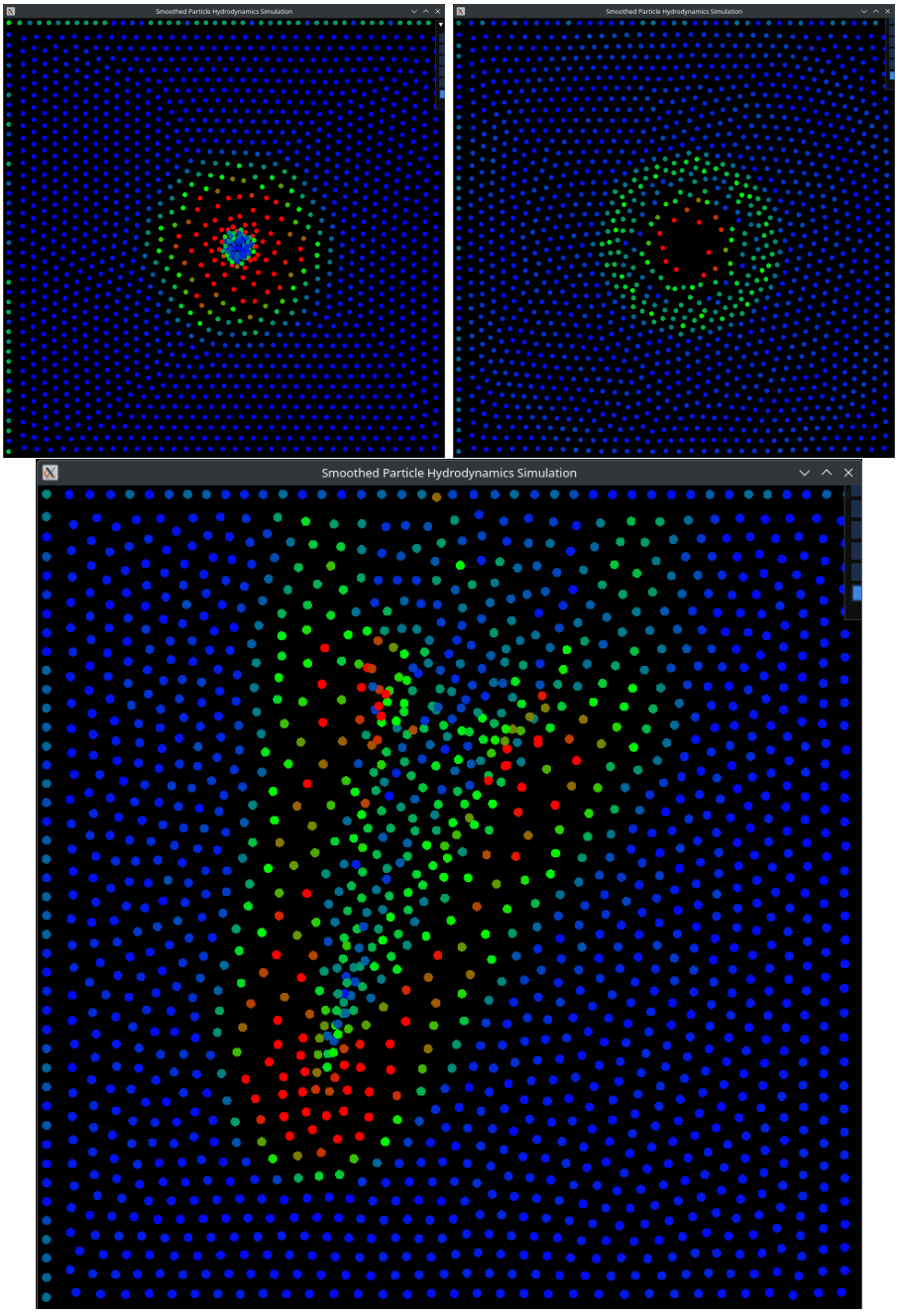


Figure 8: Attractive and repulsive mouse forces with mouse drag effect.

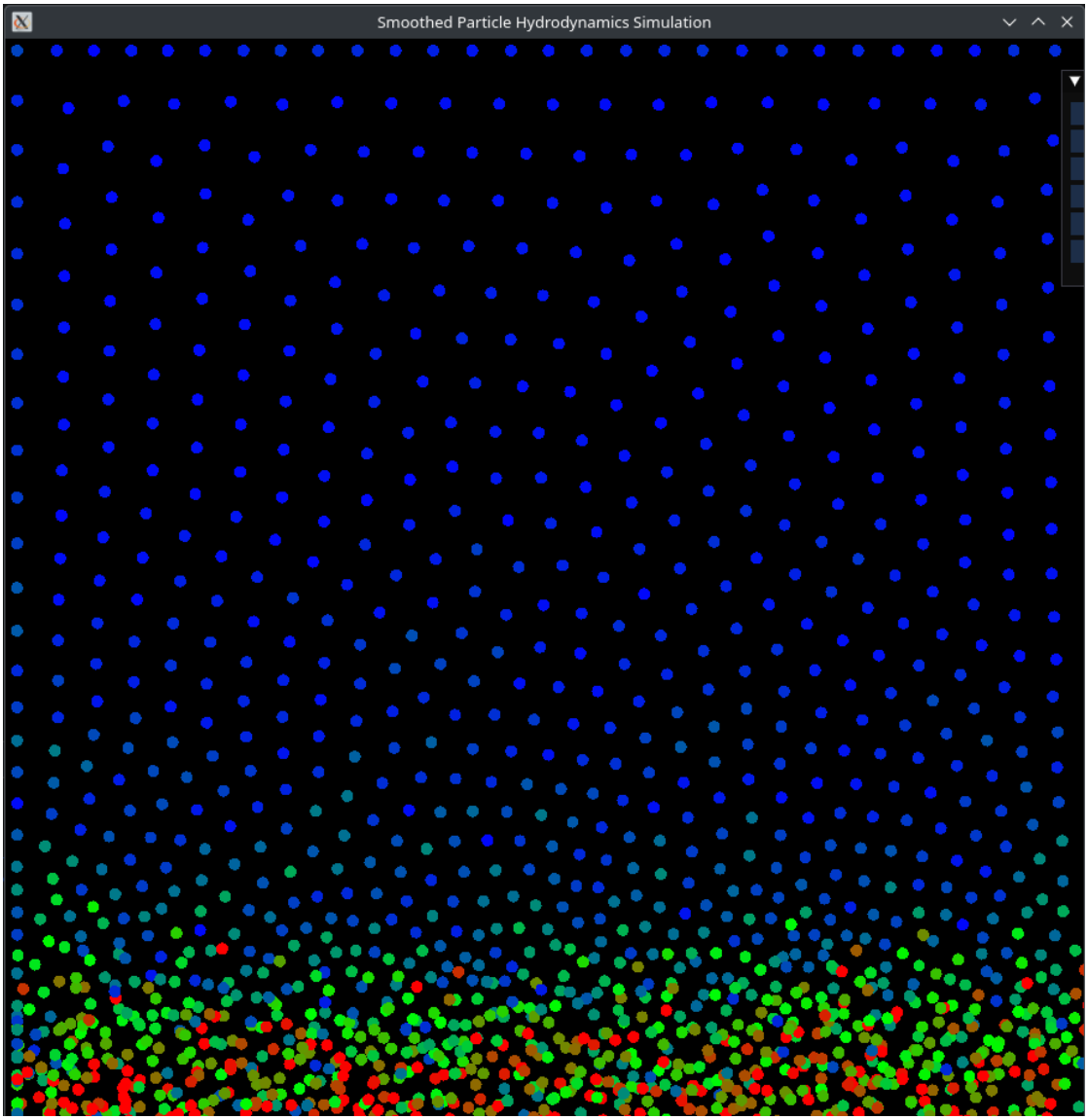


Figure 9: Fluid with gravity behaving like a boiling liquid or heavy gas.

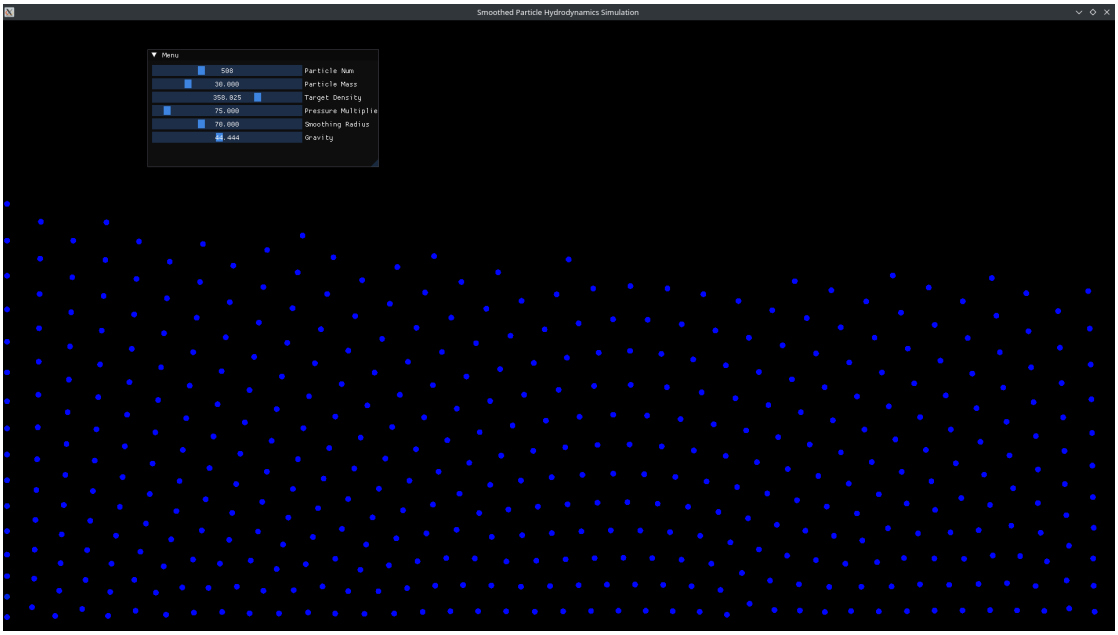


Figure 10: Fluid with gravity giving an unsmooth surface.

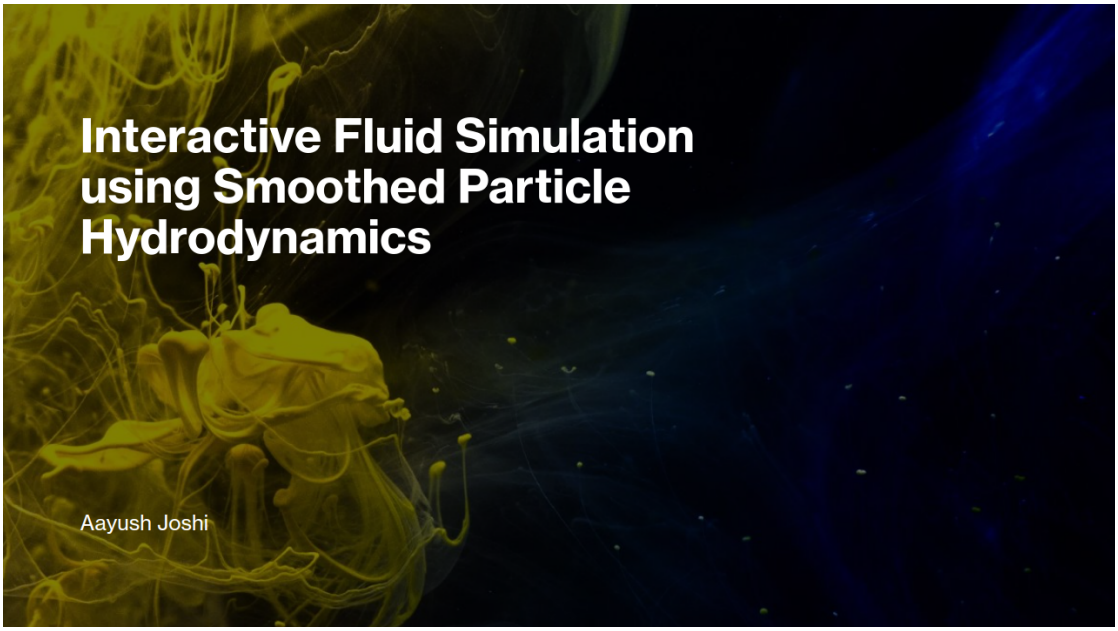


Figure 11: Slide 1.

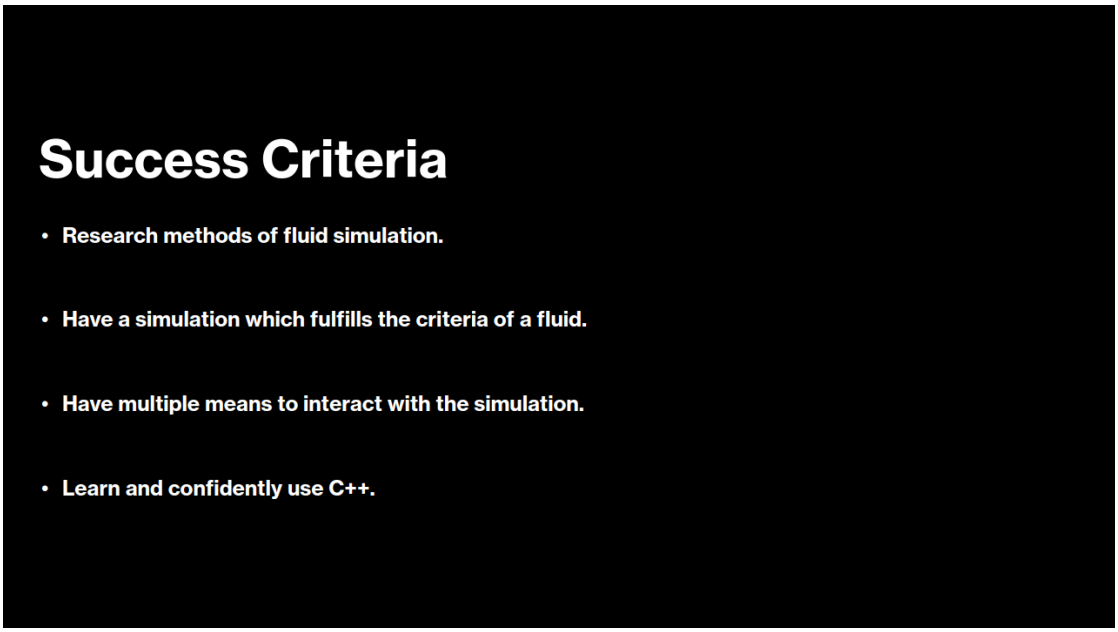


Figure 12: Slide 2.

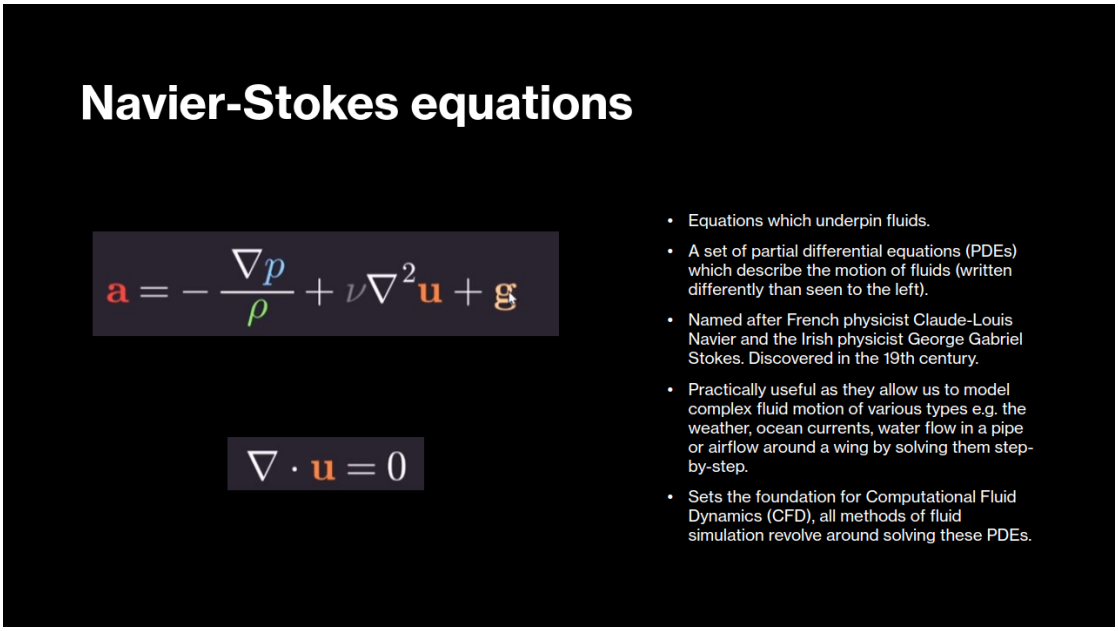
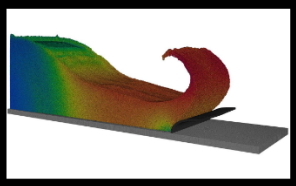
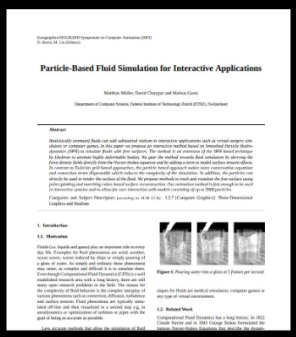


Figure 13: Slide 3.

Introduction to SPH

- Smoothed Particle Hydrodynamics (SPH) is a CFD method.
- Lagrangian – Particle Based
- Was initially developed as a testing tool to help with astrophysical calculations involving fission within stars in 1977.
- SPH was adapted for interactive applications in 2003, as opposed to the more common Eulerian method at the time.
- After this foundational work, many more recent publications have popularised SPH.
- It is now a common choice for small scale, semi-realistic fluid simulations with interactive applications.





Particle-Based Fluid Simulation for Interactive Applications

Figure 14: Slide 4.

SPH: Fundamentals

- In SPH, each particle represents volume of fluid and has fluid properties:
  - Mass
  - Velocity
  - Local density
- Properties interact over a circle of influence– creates the fluid like continuum
- For each particle, this influence it exerts on every other particle within the circle is calculated.
- This influence value is then used to consider fluid properties of the particles at that point.

```
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};
};
```

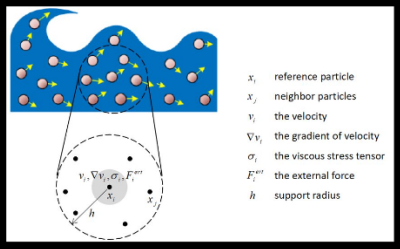
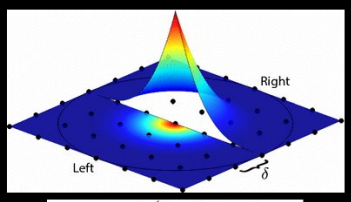


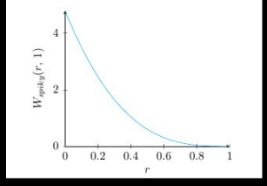
Figure 15: Slide 5.

SPH: Smoothing Kernel and Interpolation equation

- Influence is quantified by a function known as the Smoothing Kernel.
- This kernel takes the distance between 2 particles as an input and outputs an influence value.
- This special value allows us to interpolate (calculate) the numerical value of any fluid property using the SPH Interpolation equation.
- Therefore, we can calculate the density and pressure of the fluid around the particle because we have a value for it.



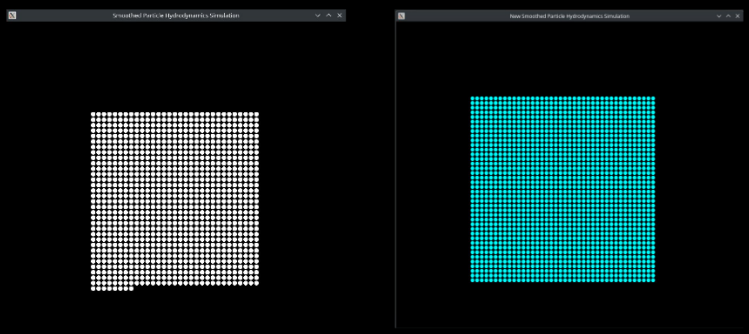
$$W_{\text{spiky}}(r,h) = \frac{15}{\pi h^6}\begin{cases}(h-r)^3 & 0 \le r \le h \\ 0 & \text{otherwise,}\end{cases}$$



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

Figure 16: Slide 6.

Development: Boilerplate



- Before simulating fluids, I first rendered the fluid particles in C++ using SFML.
- Getting particles to be rendered on screen was a huge development milestone.

Figure 17: Slide 7.

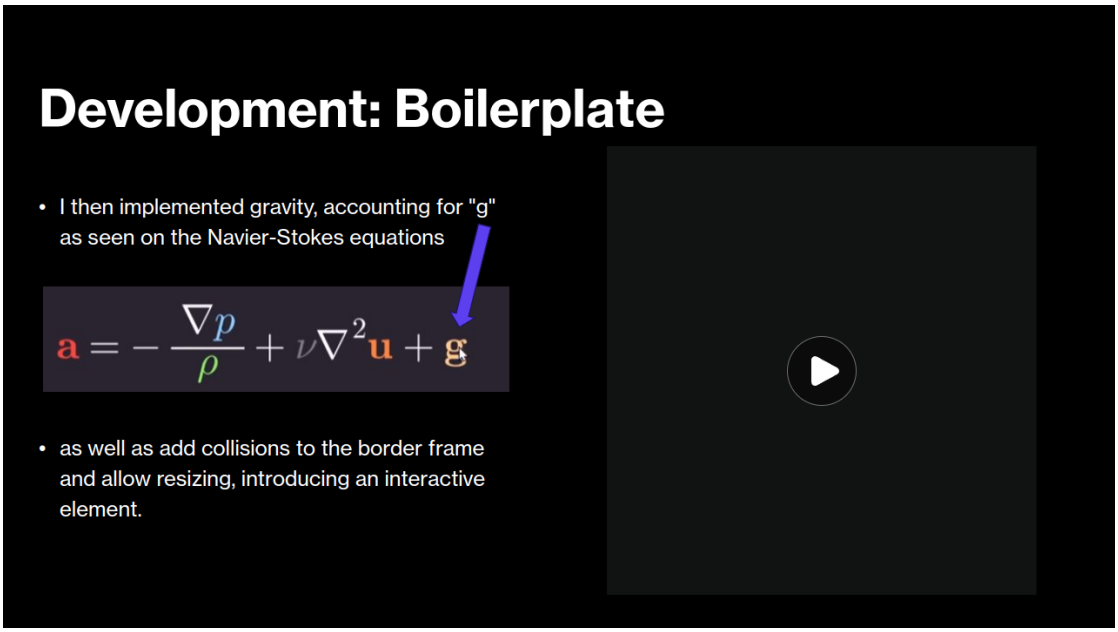


Figure 18: Slide 8 with resizing border video.

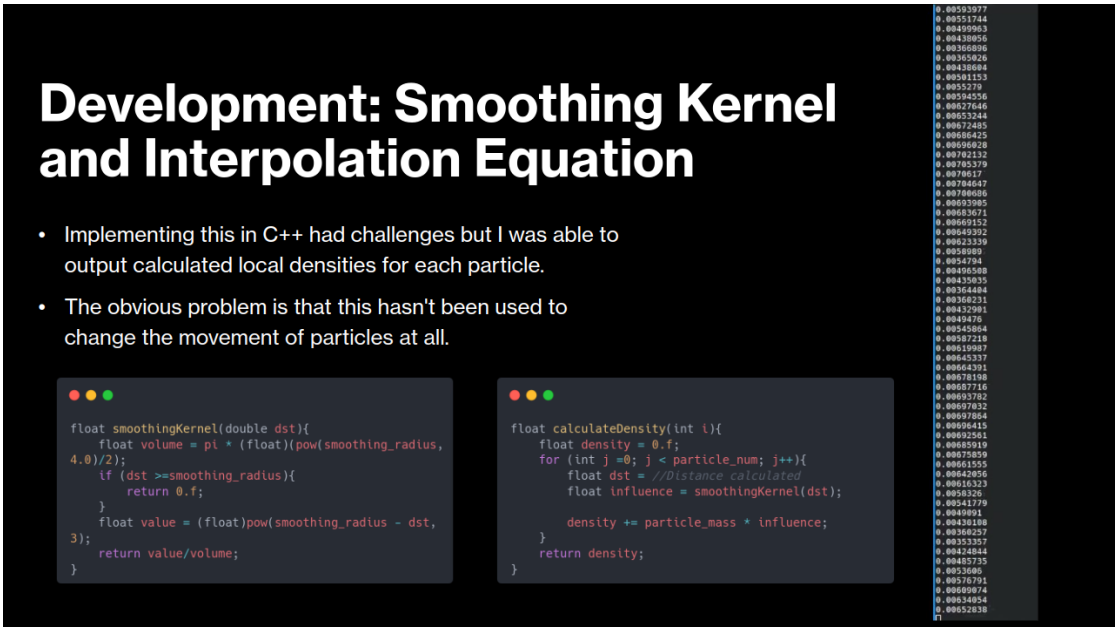


Figure 19: Slide 9.

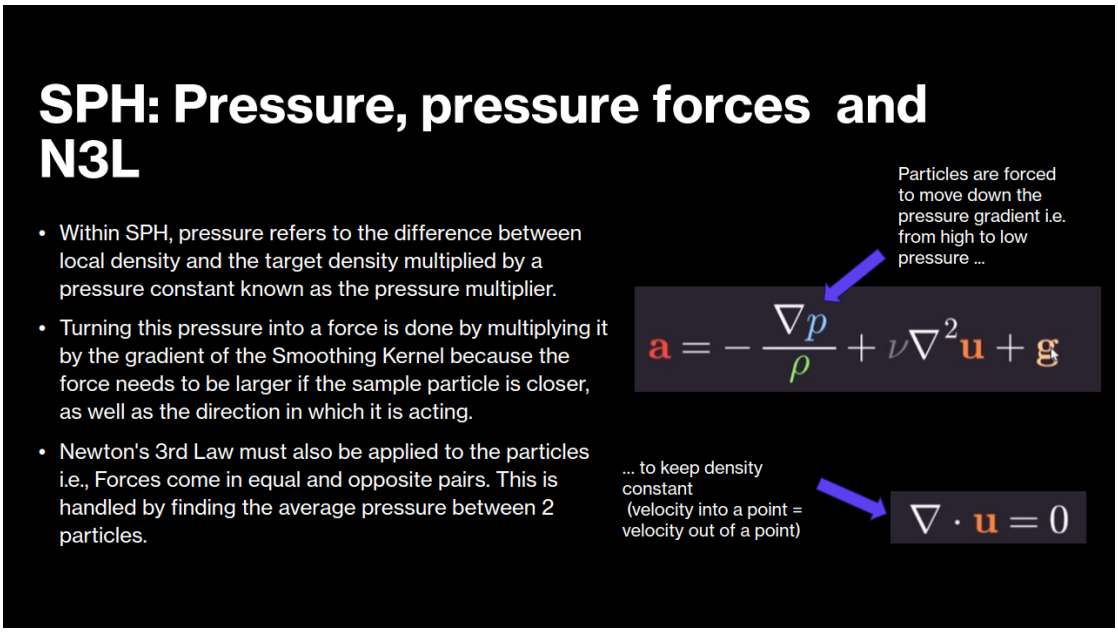


Figure 20: Slide 10.

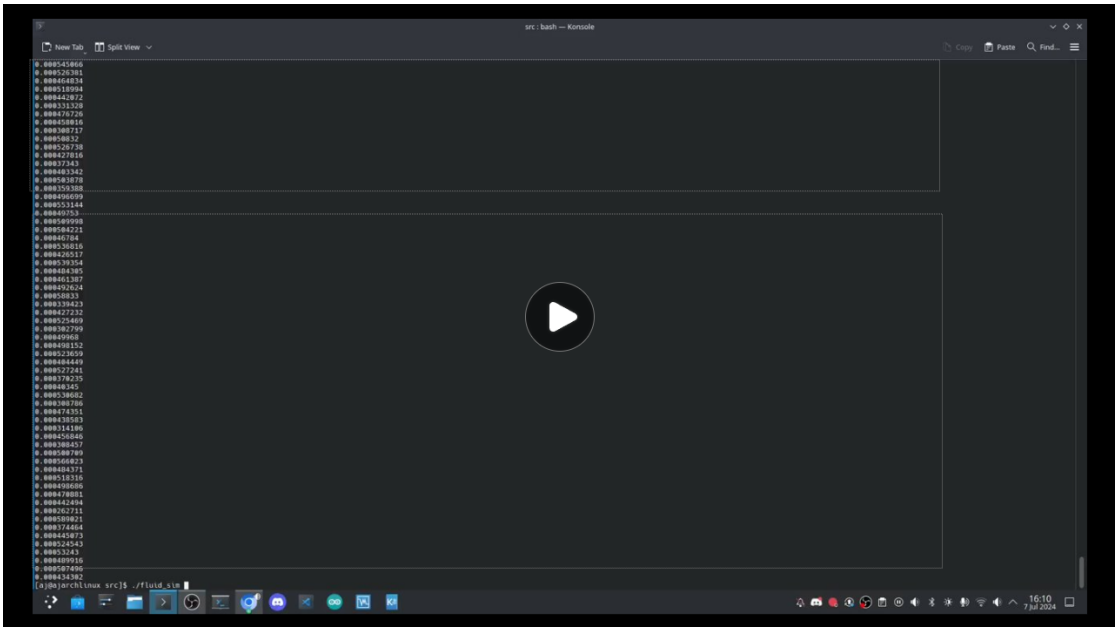


Figure 21: Slide 11 with video of first simulation.

Development: Pressure, pressure forces and N3L

$$\mathbf{a} = -\frac{\nabla p}{\rho} + \nu\nabla^2\mathbf{u} + \mathbf{g}$$

- At this stage, the development was missing off acceleration due to viscosity within the simulation. This term within the Navier-Stokes equation was unaccounted for.
- Pressure forces and densities are accounting for some viscosity clearly visible in the early simulation of the project.
- This was due to a mathematically incorrect version of the model running at the time, although development wasn't challenging, testing and perfecting the simulation was cumbersome as values would have to be manually updated within the code every time for small changes.

Figure 22: Slide 12.

SPH: Viscosity

- Viscosity within a fluid is the friction atoms experience. It isn't different to the day-to-day usage of the word.
- In SPH, this is tackled by taking the average velocity of the particles within the smoothing radius and applying a viscosity force, calculated in a similar manner to the pressure forces.
- This forces particles with unusually high velocities to match the velocity of its neighboring particles, which is visible in the current model of the simulation.

Determines the acceleration due to viscosity within a particle

$$\mathbf{a} = -\frac{\nabla p}{\rho} + \nu\nabla^2\mathbf{u} + \mathbf{g}$$

Figure 23: Slide 13.

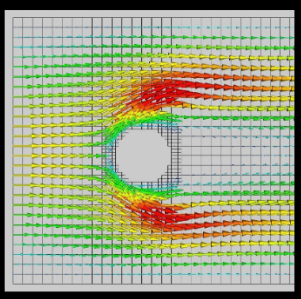
Adding Interactive elements – Sliders, Colour and Mouse forces

- Implemented sliders using ImGui in C++.
  - Allowed me to better test and fine-tune values to see their effects in real time.
  - Helps save time.
  - Helps others understand the effects of changing key variables.
- Implemented a Linear colour interpolation algorithm.
  - Helps visualise the fluid better.
  - Aesthetically pleasing.
- Implemented mouse forces.
  - Helps test the main advantage of SPH - interactivity

Figure 24: Slide 14.

Alternative: Eulerian methods

- Popular alternative to Lagrangian methods of simulating fluids.
- Grid-based instead of particle based:
  - Spatial domain is split into equal sized grids
  - Measurements of fluid quantities e.g. density or velocity are tracked
  - Advection is created by considering the total inflow and outflow of fluid in a box within the grid
- Has higher numerical accuracy and efficiency.
  - Used more in the aerospace industry
- **Not interactive**
  - Resizing the window would restructure the grid which the method relies upon.
  - Lagrangian methods (SPH) are preferred in interactive situations.



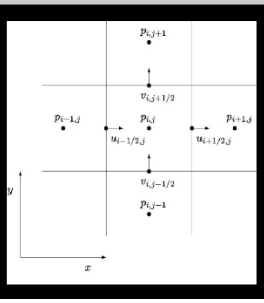


Figure 25: Slide 15.

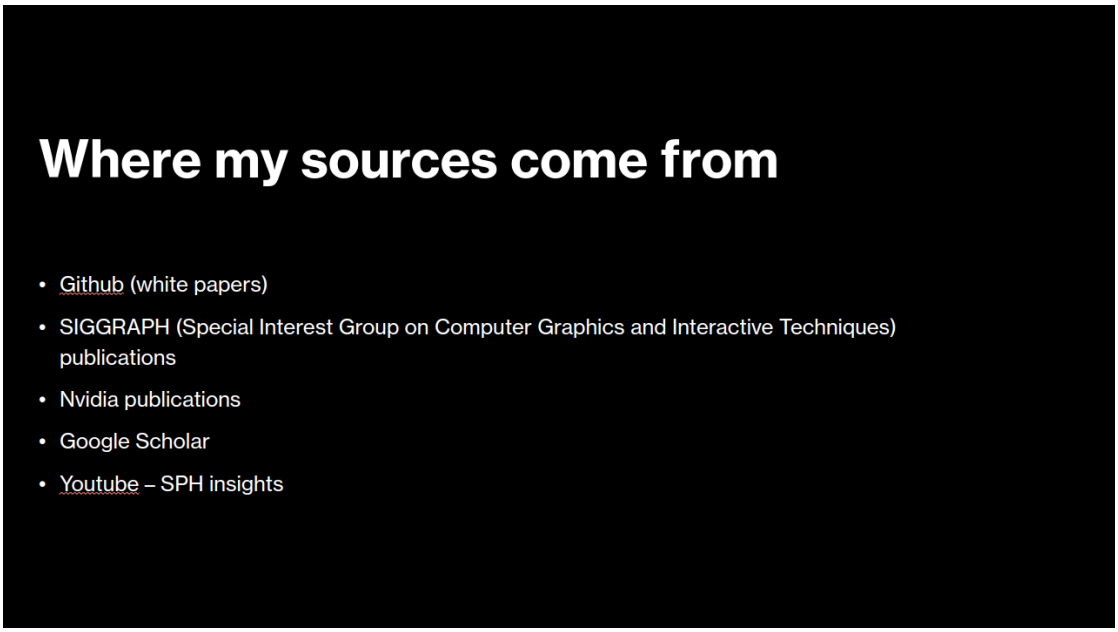


Figure 26: Slide 16.

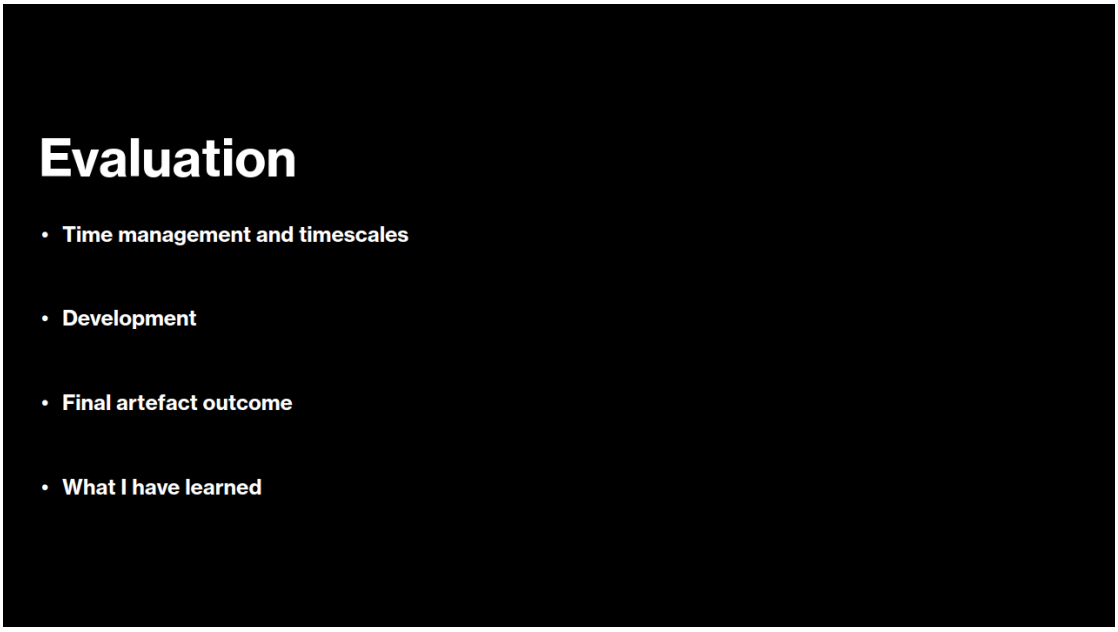


Figure 27: Slide 17.

Evaluation

- **Time management and Timescales**
  - I initially spent 2 hours weekly on Thursdays/Fridays during my study periods. Researching at school and developing at home.
  - The first half of my timescale was extremely accurate.
  - After the completion of my design brief, it became more and more inaccurate.
  - Extension of deadlines meant I nearly doubled development over the summer.

Activities and timescales

Activities to be carried out during the project	How long this will take
Complete initial research for the project - look at articles on SPH published by reputable sources via Google scholar. View past projects using Github and watch any relevant Youtube videos to gain initial understanding. Read the SFML library documentation and try initial experimentation with it on C++.	Until 19/02/24
Complete the Abstract of the project write-up	19/02/24 - 22/02/24
Complete the Introduction of the project write-up	22/02/24 - 26/02/24
Finish the Literature Review by carefully evaluating all sources.	26/02/24 - 11/03/24
Complete the Theoretical Model (Design Brief) of the artifact in the project write-up. Explain each section of the model mathematically and outline it's purpose in the overall project.	18/03/24 - 15/04/24
Complete project development according to the Theoretical model, record the code for each section and explain how it incorporates the theoretical model.	15/04/24 - 1/07/24
Prepare for the presentation	1/07/24 - 10/07/24
Conduct my presentation on the day of the EPQ showcase.	10/07/24
Complete the final evaluation, finalize the Bibliography and the Appendix ready for submission.	10/07/24 - Day of submission

View History

Figure 28: Slide 18.

Evaluation

- **Development**
  - I managed the development of my project using Github.
  - This allows for code to easily be uploaded on the cloud and be managed across multiple devices.
  - Initially, development was slow. An aim for this project was to learn C++, therefore a lot of time was spent debugging.
  - Implementing Pressure/Forces was greatly simplified due to well-planned research and model development.
  - Implementing interactive elements incrementally over the summer as well as viscosity also went relatively smoothly.

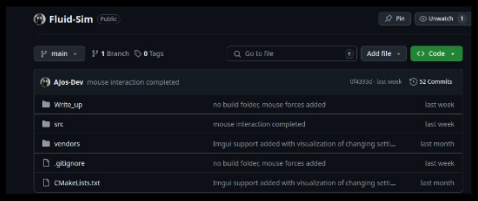


Figure 29: Slide 19.

Evaluation

- **Final artefact outcome**
  - Behaves semi-realistically as a fluid by fulfilling the theoretical model.
    - Moves particles down pressure gradients evidently is successful.
    - Attempts to ensure density is constant is also well handled as a result.
  - Highly successful at being interactive.
  - Gravity implementation could be improved, currently particles behave best as a liquid under gravity when there are a low number of particles.
  - Boundary and edges are notorious for being a disadvantage of SPH, implementing optimisations to render more particles would smooth out the anomalous boundary cases.
  - Develop the project in a game engine e.g. Unity to handle distance calculations, this would lead to more standardised calculations.

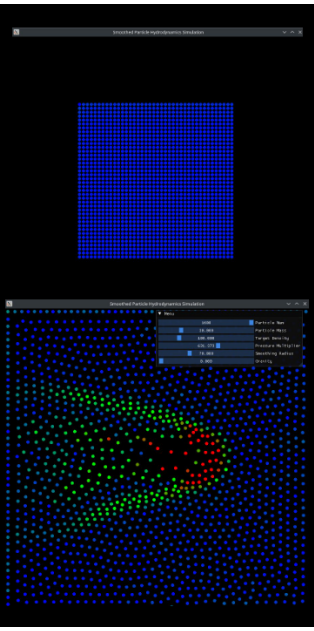


Figure 30: Slide 20.

Evaluation

- **What I have learned**
  - Manage my time and adapt to changes in timescales.
  - Interpret academic publications in a Maths/CS background.
  - Write and structure an academic dissertation, including using LaTeX.
  - Reference sources properly.
  - Code in C++.
  - A method of simulating fluids.

Figure 31: Slide 21.