2D Smoothed particle hydrodynamics simulation

Alex Newland-Jarman

Candidate NO:

Centre No:

Simon Langton Boys

AQA COMPUTER SCIENCE 2023/2024

CONTENTS

CONTENTS **2**
References **4**
Reference list **4**
Analysis **6**
Background To/Identification Of Problem 6
Background: 6
Identification of Problem 6
Description Of the current System 6
Research into fluid simulation techniques 6
Displaying and Animating particles 7
Prototype / Modelling 8
What is SPH and how am I incorporating it in my project? 9
Steps to Make a fluid simulation 10
Handling Particles calculations 10
Smoothing Kernels 11
Estimating Density 12
Calculating Pressure 13
Calculating pressure force 14
Calculating Viscosity Force 15
Finishing off the calculations 15
Optimization 15
Identification of the prospective User 16
Identification Of User Needs 16
Interview 16
Interview with Professor Smith 16
Interview with Sarah 17
Interview conclusion 18
Points based on interview 18
After Project finished 18
Data Flow Diagrams 18
Objectives Of the Project 19
Objectives 20
DOCUMENTED DESIGN **21**
Overview of how system works 21
Modular structure 22
Description of each module 23
Draw: 24

Calculate density: 24
Calculate Pressure: 24
Calculate Pressure force: 24
Calculate Viscosity force: 24
Form structure 24
Mouse controls (Modify user interactions) 24
Fluid controls (sliders and toggles) 25
Pressure multiplier 25
Viscosity 25
Toggle speed colours 25
Frame settings 25
Number of Particles 25
Class Diagrams 25
Designing the Key Algorithms / Fluid simulation algorithms(pseudocode) 26
Quad tree algorithm 26
Nearest neighbour algorithm 27
Calculate Density algorithm 28
Calculate Pressure Force 28
Calculate Acceleration 28
Updating position 29
Drawing Particles onto screen 29
Key Data structures 29
Quad tree to find nearby particles. 29
UI 31
Software requirements 32
Hardware requirements 32
Technical solution **32**
Initialising a Particle 32
Animation/calculation loop 34
Physics Calculations 35
Updating acceleration 35
Implementing Navier stokes equations 36
Calculation of density and pressure using nearby points 36
Calculate the fluid forces on a particle 37
Update Speeds and handle boundary conditions 38
Update Speeds 38
Handle boundary cases 39
Exploring collision calculations 40

Drawing to the canvas	41
Quadtree implementation	42
Shapes	42
Subdivisions	43
Inserting points	44
Fluid simulator Class	46
Interacting with the Fluid and changing Constants	48
External Forces	49
LINKS TO VIDEO AND SOURCE CODE	**50**
Appendix	**50**
File structure	50
Index.html	51
Particle.js	52
Quadtree.js	59
Sketch.js	65
SPHFluidsimulator.js	71
Styles.css	75

References

Reference list

Anderson, P. (1977). Numerical methods Chapter 2 Simulations with Smoothed Particle Hydrodynamics. [PDF] pp.1–38 pages. Available at: https://astro3.sci.hokudai.ac.jp/~alex/Material/ARPettitt_chapter2.pdf [Accessed 14 Nov. 2023].

Bridson, R. (2005). Particle-based Fluids. [online] Available at: https://www.cs.cmu.edu/~scoros/cs15467-s16/lectures/11-fluids2.pdf [Accessed 22 Nov. 2023].

Hubber, D. (2015). Smoothed Particle Hydrodynamics. [online] pp.1–32. Available at: https://gandalfcode.github.io/gandalf-school/talks/SphMethods.pdf [Accessed 14 Nov. 2023].

Müller, M. (2022). How to write a FLIP Water Simulator. [online] Available at: https://matthias-research.github.io/pages/tenMinutePhysics/18-flip.pdf [Accessed 16 Nov. 2023].

Nuclear Power. (n.d.). Navier-Stokes Equations | Definition & Solution | nuclear-power.com. [online] Available at: https://www.nuclear-power.com/nuclear-engineering/fluid-dynamics/navier-stokes-equations/.

Green, S. (2010). Particle Simulation using CUDA. [online] Available at: https://developer.download.nvidia.com/assets/cuda/files/particles.pdf [Accessed 27 Nov. 2023].

Bridson, R. (2007). Smoothed Particle Hydrodynamics 15.1 Simple Particle Systems. [online] Available at: https://www.cs.cornell.edu/courses/cs5643/2015sp/stuff/BridsonFluidsCourseNotes_SPH_pp83-86.pdf [Accessed 6 Mar. 2024].

Shiffman, D. (2018). Coding Challenge #98.3: Quadtree Collisions - Part 3. [online] www.youtube.com. Available at: https://www.youtube.com/watch?v=z0YFFg_nBjw&pp=ygUVY29kaW5nIHRyYWluIHF1YWR0cmVl [Accessed 7 Mar. 2024].

Analysis

Background To/Identification Of Problem

Background:

Students in higher education that study subjects like physics/engineering or applied mathematics will learn advanced ideas and formulas without understanding their physical application. One of these ideas that are hard to visualise are fluid simulations.

Identification of Problem

During high level fluid mechanics courses students receive lots of information and equations but sometimes when learning new topics it's helpful to get intuition about how these skills can be used in real life. So i have decided to create an intuitive tool for understanding fluid mechanics although they can be taught through the textbooks and lecture notes i believe that creating an interactive fluid simulation for a student to mess around with would give a good insight into basic phenomena like viscosity and how density affects fluids it will also allow them to see the undercurrents of the water as i will be creating a 2d simulation. I have researched fluid simulations to find out simple ways of making this simulation and the most useful for the students would be javascript as it allows portability and is easy to change the user interface.

Description Of the current System

Currently students learn the topics without an interactive system using the textbooks available which contain all the information for that level to understand the topic. Before lesson old resources are available for the students to learn the topic but it lacks the hands-on learning approach that lots of students need. This is especially important for those students wanting to get into the engineering field where it is important to know how theory relates to the real world.

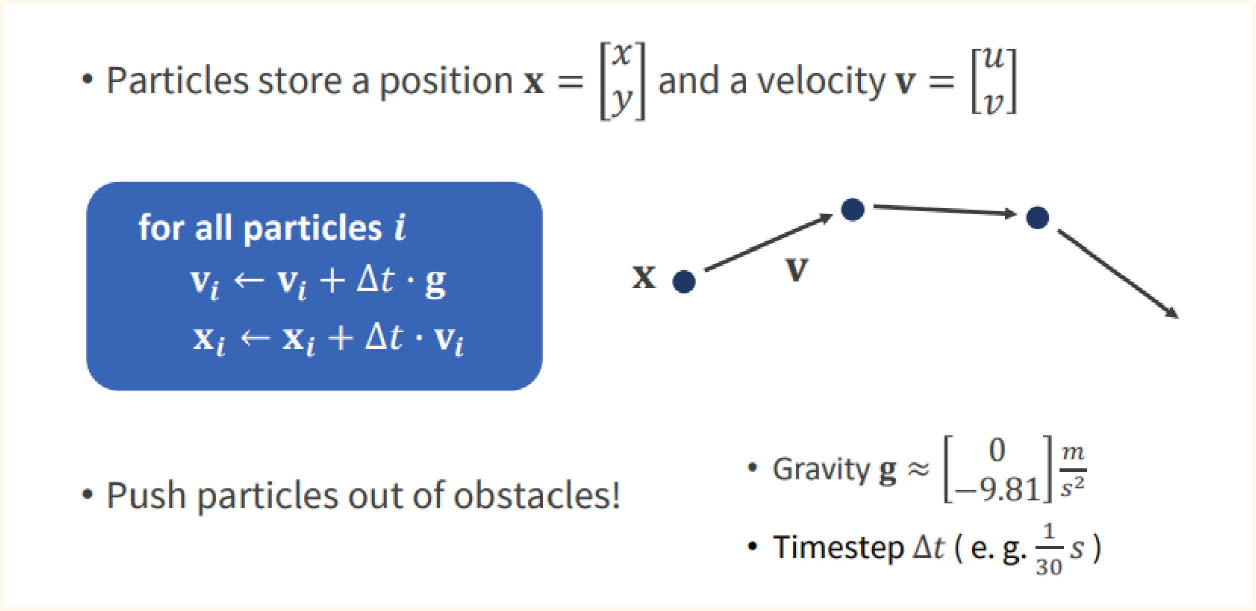
Research into fluid simulation techniques

When researching i found two techniques for rendering a fluid that i understood one was the particle in cell method which involved grouping particles into cells and computing their influence on the cell they are contained in and then re converting the velocities back into the individuals cells you can read more about it here(https://matthias-research.github.io/pages/tenMinutePhysics/18-flip.pdf) The second method i discovered was called SPH also known as smoothed particle hydrodynamics, and it involves treating each particle as a volume of water with an influence around it where the particles can only affect those in a certain radius which gets smoothed out over distance. I chose the SPH method because I had trouble originally with turning particle velocities into grid velocities viceversa it is also really good for unbounded areas and adding extra user input.

Displaying and Animating particles

In this simulation the particles are stored in an array and then plotted onto a screen using the position and radius stored in the object.But just drawing a particle onto a screen once is not enough to create a simulation we need to be able to constantly re animate the particle whenever it moves to do this we will create an animation loop that repeatedly calls a draw and update function. Now that are particle are being constantly updated we can move onto the next step

We need to allow particles the ability to move. We can add 2 new properties to the particle velocity and acceleration. With these 2 properties we can now reconstruct basic kinematic equations. To do this we will update the velocities of are particles by adding on the acceleration multiplied by timestep (this time step will be explained later) and with this new updated velocity we can update the position by again multiplying the velocity by time and adding it to the position.The time step is very important as we are doing a simulation in real time so we have to predict ahead. Below are some diagrams to help understand the idea.



- Particles store a position $\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$ and a velocity $\mathbf{v} = \begin{bmatrix} u \\ v \end{bmatrix}$

```
for all particles $i$
    $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t \cdot \mathbf{g}$
    $\mathbf{x}_i \leftarrow \mathbf{x}_i + \Delta t \cdot \mathbf{v}_i$
```

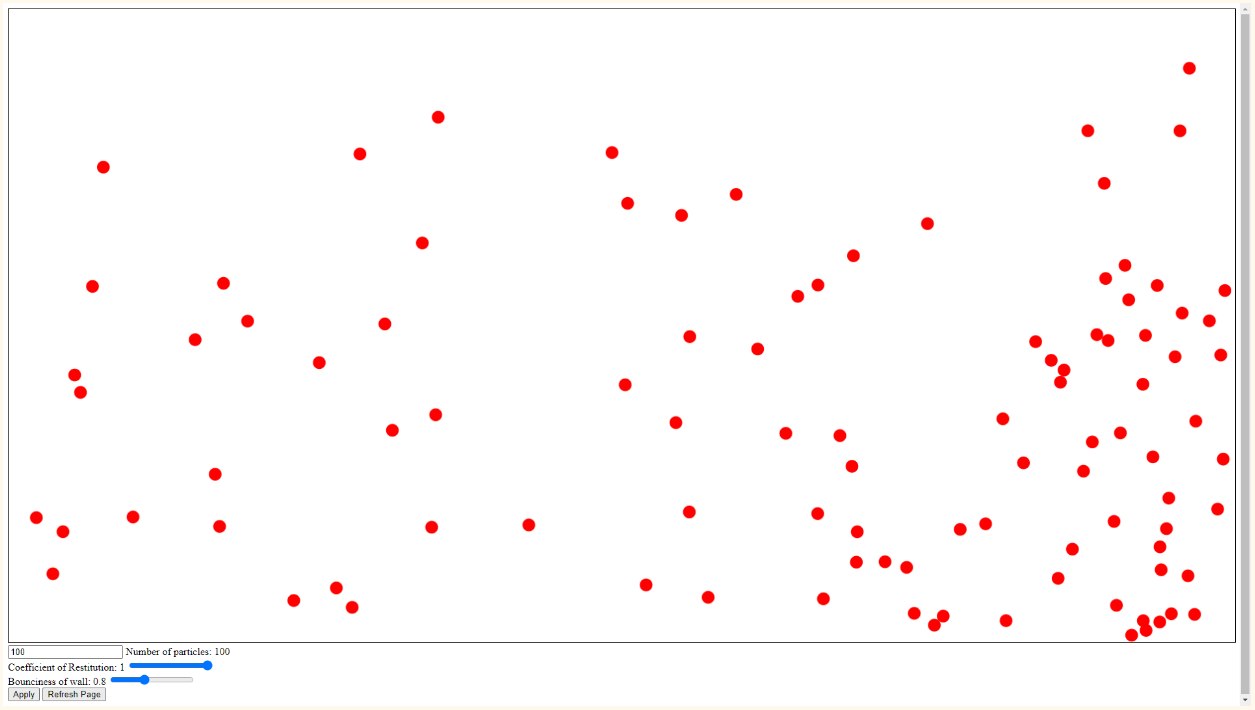
- Push particles out of obstacles!
- Gravity $\mathbf{g} \approx \begin{bmatrix} 0 \\ -9.81 \end{bmatrix} \frac{m}{s^2}$
- Timestep Δt (e. g. $\frac{1}{30} s$)

[Figure 3 from *Müller, M. (2022)*] the above image should help describe how a particle moves using a timestep and velocities and acceleration except in the simulation g is replaced for acceleration which can be altered unlike gravity.

If you were to run your simulation now you would notice that the particle falls out of the screen to prevent this reverse the velocity component and move the position component to the edge of the boundary when it collides or goes through the edges

Prototype / Modelling

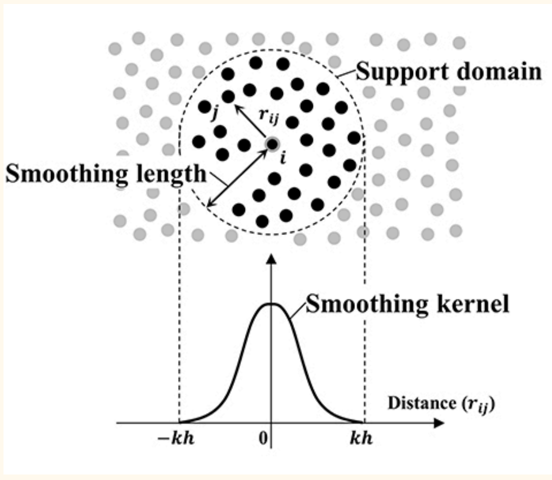
Smoothed Particle Hydrodynamics involves using discrete particles that have multiple properties assigned to them with the strength of each property on another particle decreasing over a distance. During the summer I created a small simulation that just demonstrates drawing particles on the screen and allowing collisions between a barrier and other particles. I also added sliders to learn the skills needed to add changing variables in html.

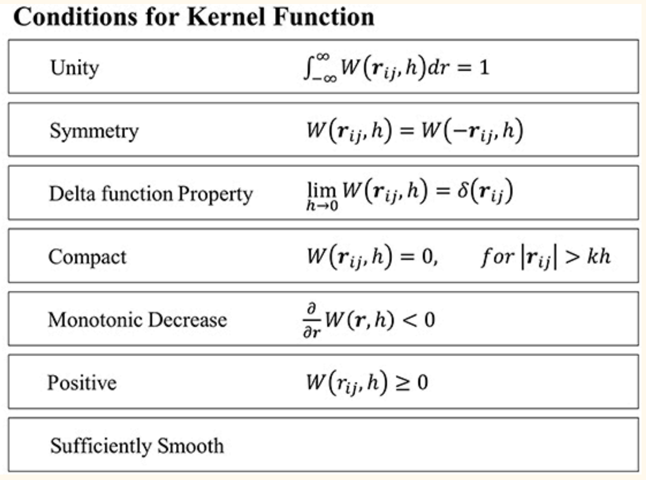

100 Number of particles: 100
Coefficient of Restitution: 1
Bounciness of wall: 0.8

[figure 1] This image shows a javascript and html simulation of multiple red particles colliding. There are parameters that can be adjusted like coefficient of restitution and bounciness of the wall which help mimic energy transfers that occur during real collisions.

What is SPH and how am I incorporating it in my project?

A fluid is a substance that has no fixed shape and is able to flow (water is a fluid). In smoothed particle hydrodynamics the fluid is divided into discrete particles with mass. You could imagine them as water droplets. The effect of each particle on another particle drastically decreases as the distance gets higher until it is 0. We can determine the exact influence of a particle using a smoothing kernel. Shown below




Conditions for Kernel Function

Unity	$\int_{-\infty}^{\infty} W(\mathbf{r}_{ij}, h)dr = 1$
Symmetry	$W(\mathbf{r}_{ij}, h) = W(-\mathbf{r}_{ij}, h)$
Delta function Property	$\lim_{h\to 0} W(\mathbf{r}_{ij}, h) = \delta(\mathbf{r}_{ij})$
Compact	$W(\mathbf{r}_{ij}, h) = 0, \quad for\ \lvert\mathbf{r}_{ij}\rvert > kh$
Monotonic Decrease	$\frac{\partial}{\partial r} W(\mathbf{r}, h) < 0$
Positive	$W(r_{ij}, h) \geq 0$
Sufficiently Smooth	

[figure 2]The smoothing kernel takes parameters r distance from particle and h smoothing length and determines the influence of a particle the particles in the support domain are then used to calculate the property for example density using their influence to determine how much their density affects the centre particles density.

Steps to Make a fluid simulation

By breaking down the creation of a fluid simulation into multiple steps it becomes a lot easier to reconstruct for any skill level where each step calculates a term of the Navier stoke equation .

Handling Particles calculations

In SPH fluid is discretized using particles each particle represents a volume of fluid

$$V_i = \frac{m_i}{\rho_i}$$

(where mass stays constant but density does not)

Each particle stores attributes like density and pressure. We can calculate an attribute at a point in space by taking the weighted average of particle attributes within the support domain.

$$A(x) = \sum_{i=0}^{n} \frac{m_i}{\rho_i} A(x_i) W(\lvert x - x_i \rvert, h)$$

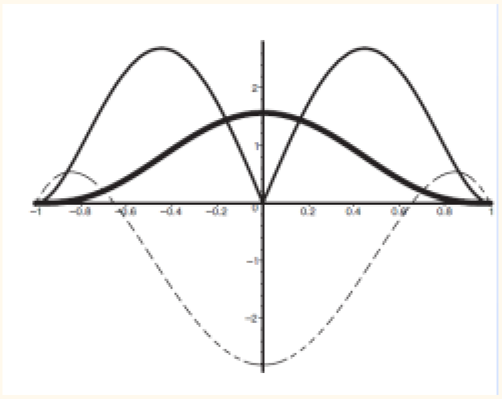
Where W is the smoothing kernel which determines the influence the particle has on the attribute

Smoothing Kernels

There are many different smoothing kernels with different uses:

Poly6 kernel is used for everything except pressure forces & viscosity:

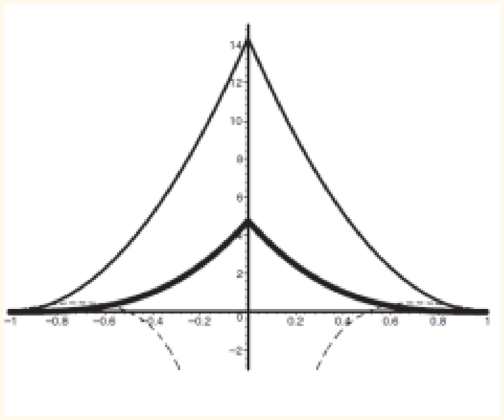
$$W_{\text{poly6}}(\mathbf{r},h) = \frac{315}{64\pi h^9}\begin{cases}(h^2-r^2)^3 & 0 \le r \le h\\ 0 & \text{otherwise}\end{cases}$$



[figure 3 *Bridson, R. (2005)*] The above image is a piecewise function that equals 0 when the smoothing length is 0 and otherwise it rises slowly and is the same on both the positive and negative halves of the graph

The **Spiky Kernel** is used for pressure forces:

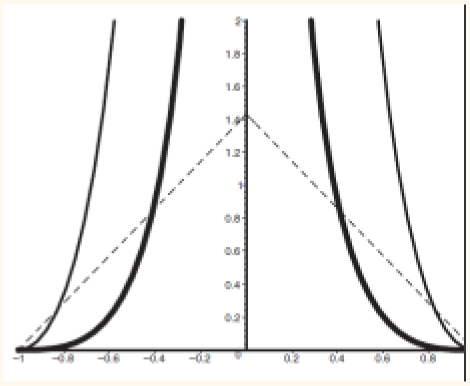
$$W_{\text{spiky}}(\mathbf{r},h) = \frac{15}{\pi h^6}\begin{cases}(h-r)^3 & 0 \le r \le h\\ 0 & \text{otherwise,}\end{cases}$$



[figure 4 *Bridson, R. (2005)*] A spiky kernel with high gradient near r = 0 so that particles do not clump together

The **Viscosity kernel** sadly doesn't have a cool name but has some useful properties:

$$W_{\text{viscosity}}(\mathbf{r},h) = \frac{15}{2\pi h^3}\begin{cases} -\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & 0 \le r \le h \\ 0 & \text{otherwise.} \end{cases}$$



[figure 5 *Bridson, R. (2005)*] Laplacian is always positive

There are many different smoothing kernels but they all have similar properties

- Radially symmetric
- Area under the curve is equal to w
- W is an even function
- And when the magnitude of r is greater than smoothing length equal influence = 0

Estimating Density

$$\rho\frac{\partial \mathbf{u}}{\partial t} = -\nabla p + \mu\nabla^2\mathbf{u} + \rho\mathbf{g}$$

Now we have a function to calculate the attribute of a particle. We can finally calculate the most important attribute of a fluid which will allow us to calculate everything else. This property is Density!

$$A_j = \sum_{i=0}^{n} \frac{m_i}{\rho_i} A_i W(\left|x - x_i\right|, h)$$

To calculate the density we change the A attribute to density(ρ) so that they cancel and we will get a new equation. Make sure to use the smooth Poly6 kernel for densities.

$$\rho_j = \sum_{i=0}^{n} \frac{m_i}{\rho_i} \rho_i W(\left|x - x_i\right|, h) = \sum_{i=0}^{n} m_i W(\left|x - x_i\right|, h)$$

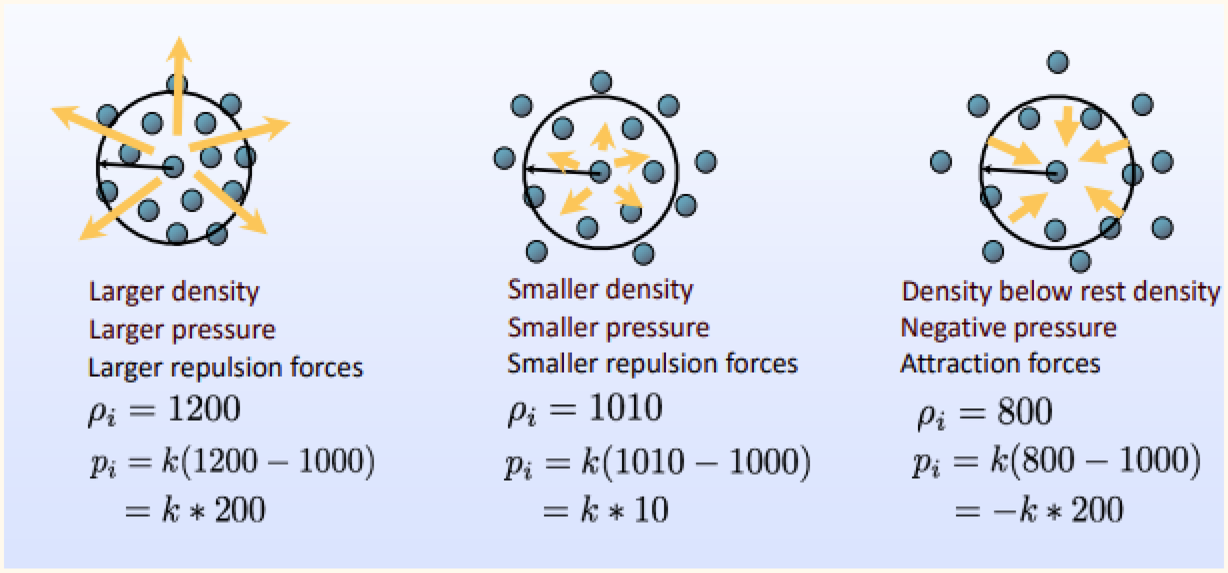
The final density of the particle should never be zero but in are simulation it will sometimes become zero as it is not 100% accurate so when this occurs you could either set the density to be a very small value or do what I did and just don't use that density

Calculating Pressure

We can calculate the pressure using ideal gas laws $P = k\rho$ which can also be derived from the equations $PV = k \ and \ \rho = \frac{m}{V}$ a modified version gives the pressure of particle i:

$$P_i = k(\rho_i - \rho_0) \ where \ \rho_0 \ rest \ density \ is \ 1000 \ for \ water$$

When the density is too high or too low the fluid will try and correct itself so that it reaches the ideal density the speed at which it corrects itself can be altered by changing the value k (stiffness constant)



[figure 6 *Bridson, R. (2005)*] shows the effects of different densities on the movement of particles.

Usually lots of attraction forces can lead to numerical instability so we typically only use repulsion forced to cap the pressure to positive

$P_i = max(k(\rho_i - \rho_0), 0)$

Calculating pressure force

$$\rho \frac{\partial \mathbf{u}}{\partial t} = \boxed{-\nabla p} + \mu \nabla^2 \mathbf{u} + \rho g$$

To calculate the pressure force we can use the derivative of the attribute equation which is very simple please refer to sources for more understanding of the result[Bridson (2007) 15.9]. Changing the attribute A to Pressure P but does not follow conservation of energy rules so we have to find the mean pressure instead for our pressure force this means that the particles will both get equally pushed back.

$$\nabla P_j = \sum_{i=0}^{n} \frac{m_i}{2\rho_i} (P_i + P_j) \nabla W(|x - x_i|, h)$$

For calculating pressure forces we use the derivative of the spiky kernel.

Calculating Viscosity Force

$$\rho \frac{\partial \mathbf{u}}{\partial t} = -\nabla p + \boxed{\mu \nabla^2 \mathbf{u}} + \rho g$$

To calculate viscosity we use the velocity in the attribute equation and also modify it to follow conservation of energy laws so instead we plug in relative velocity this allows for the fluid to slow down and approach rest the closer the particles get to the same speed. Make sure to use the Viscosity kernel .

$$\nabla^2 u_j = \sum_{i=0}^{n} \frac{m_i}{\rho_i}(u_i - u_j)\nabla^2 W(|x - x_i|, h)$$

`

Finishing off the calculations

Now to finish up we can now make basic fluid by using the Navier Stokes equation and re-arranging for acceleration simply plug in are calculated values for each term and use this acceleration when calculating the velocity and new position. For more accurate fluid simulation you can add more external forces or make the acceleration affected by different fluid properties like temperature.

$$\frac{du}{dt} =- \nabla \frac{P}{\rho} + \mu \nabla^2 u/\rho + g$$

Optimization

There are many things that are slowing down are simulation but the biggest one is when we calculate attributes currently we are just iterating through all the particles on the screen but we only need to check the nearby particles to fix this issue we can store particles using a quadtree. Which allows for quick neighbour searching when I implemented the quadtree before I would struggle with a thousand particles and now can run 2000 particles smoothly. Also moving their speeds and boundary calculations into the draw loop will decrease the number of loops so will be able to handle more particles.

Identification of the prospective User

The target audience for my project are students in university studying engineering as they need to understand in depth how fluids work not just on paper but visually to accommodate this I will make it easy for students to add to new equations to the code modularising it so as long as they have a vague understanding of coding which the majority of students studying physics or an engineering degree should be able to pick up fast .It is easily adaptable for future advancements in the field as we simply need to just change the calculation of acceleration.

Identification Of User Needs

Interview

Over the course of a few weeks i planned a visit to my local university where i interviewed professor smith and a keen student sarah who wanted to share their opinions on fluid simulations being used to educate in uni i conducted these interviews so i could see what people wanted from the simulation and what are the down sides

Interview with Professor Smith

(Disclaimer this is only a collection of the key questions asked and is not a transcript of the interview as that would be too long and not get to the point)

Me First Question:Could incorporating fluid simulations enhance students' understanding of engineering concepts in fluid dynamics?

Professor Smith: Yes. Fluid simulations provide a hands-on experience that complements theoretical learning. They allow students to visualise complex mathematical equations bridging the gap between theory and real-world applications which is exactly the skills we need from engineers.

Me: Are there any challenges or considerations when integrating fluid simulations into the curriculum?

Professor Smith: There could be challenges in terms of resources and technology. So ensuring access to appropriate software and equipment is crucial. Additionally, some students might find the learning curve steep initially if there's lots of complex ui.

Interview with Sarah

Me:As a student, what's your take on adding fluid simulations into engineering lectures?

Sarah: It would be really cool.The normal lectures can be unique, but adding fluid simulations will make fluid mechanics easier to understand. It's like seeing equations come to life, and that really helps me connect theory with practical applications.

Me: Have you experienced any significant improvements in your own understanding of engineering topics when using fluid simulations?

Sarah: Definitely. For example, It has helped me understand fluid flow patterns which I found confusing before using simulations. It also helped me realise it's not just about solving equations; it's about visualising how different parameters impact the real-world behaviour of fluids.

Me: Are there any challenges or concerns you've encountered with the use of fluid simulations in your classes?

Sarah: Well, there's a learning curve for the software, and some students might find it intimidating at first. Also, access to computers with the necessary capabilities can be a challenge, so it's hard to ensure equal opportunities for everyone.But overall, the benefits outweigh the challenges.

Interview conclusion

In conclusion the interviews I conducted are in favour of fluid simulations being used to teach students. I noticed that both the students and teachers like the idea of being able to change parameters and especially visualising the transfer of energies and vector fields. To incorporate this I will use discrete particles and colour particles based on speed or heat which will be togglable. From this interview I have looked into 2 methods that could be used to achieve the desired results: a eulerian grid based fluid simulation or a Smoothed Particle Hydrodynamics simulation. I chose to pick the SPH simulation as despite the eulerian simulation being very good at displaying vector fields and heat transfer the SPH simulation is a lot better for representing the real world as it involves particles which act like small droplets of water.Both individuals also mention concerns of accessibility of this technology so to combat this i will be making it using javascript and html so that any device with internet can access it.

Points based on interview

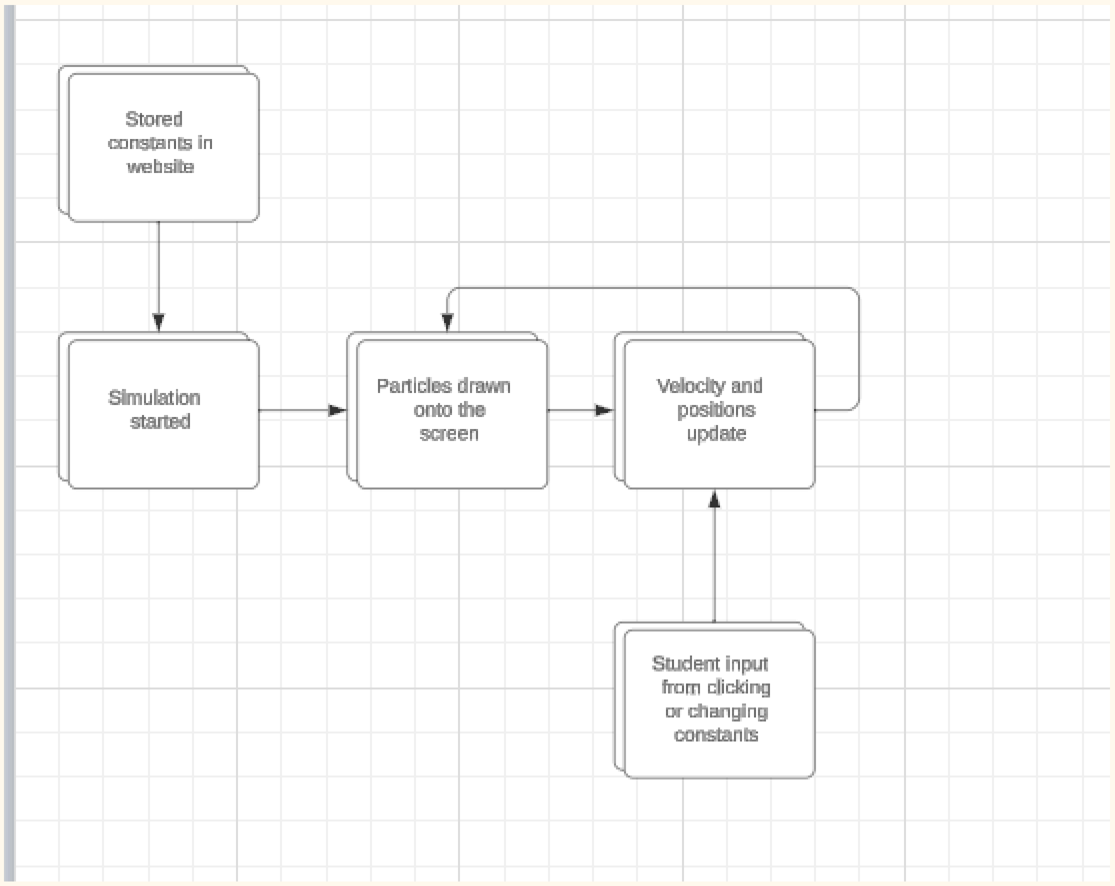
1. Program should be able to simulate fluid flow - not necessarily completely accurately
2. The user should be able to change certain parameters and start conditions and interact with the fluid
3. It should be possible to change visibility mode to see properties of a particle using changing colours
4. The user should be able to pause and step through the program frame by frame to see little changes

After Project finished

Once the project was finished I returned back to the students and asked for their opinion on the software and if it would help them. They all said that it was very good for understanding viscosity and specifically how fluids flow around objects but in the future it would be good to add temperature as they said it makes a key difference on how it would act but overall it is good for demonstrating the basics of fluid dynamics especially for someone just picking it up. It was also found that the performance was good on all devices which was a concern before the project.

Data Flow Diagrams

The below diagram demonstrates how a users input will affect the flow of code the user input gets processed during the velocity and position update so that they will be changed by the next frame


Stored constants in website
Simulation started
Particles drawn onto the screen
Velocity and positions update
Student input from clicking or changing constants

Objectives Of the Project

1. Program should be able to simulate fluid flow - not necessarily completely accurately

    Specific Objectives:

    1. Estimate the solutions to the navier stokes equations using SPH
    2. Use an efficient particle lookup method like quad trees Correctly apply the forces to the particles
    3. Display particles changed position smoothly without teleporting or visual glitches

2.The user should be able to change certain parameters and start conditions and interact with the fluid

1. Allow the user to restart simulation

2. Allow the user to change the particles amount
3. Allow the user to change pressure and viscosity values
4. Allow the user to be able to move the mouse to interact with fluid

3.It should be possible to change visibility mode to see properties of a particle using changing colours

1. Allow the user to change the colouring mode of the liquid

4.The user should be able to pause and step through the program frame by frame to see little changes

5. Create a pause button
6. Create a step through button
7. Allow the user to see properties of a particle based on colour

Objectives

1. Create a html page that renders bouncing particles that interact with the edges
2. Create a particle class to store all the methods that particle needs
3. Store each particle in a quadtree class
4. Calculate densities ,pressure force and viscosity force
5. Update the particle's acceleration in real time using calculated values of pressure and viscosity
6. Add sliders for user to change values and the ability to restart and pause the fluid

Extension objectives

- Now add external forces like a ball the user controls to cut through the liquid or compress it
- Add ability to add items on the water

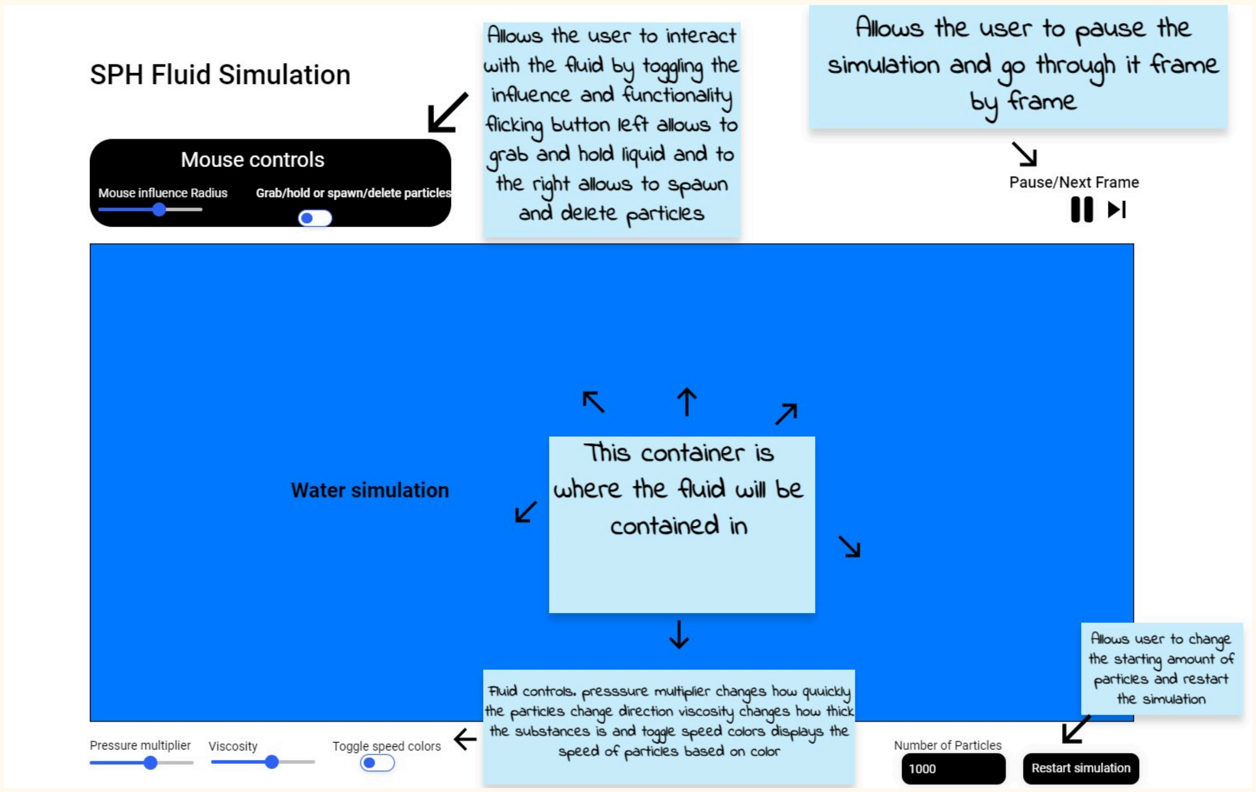
Why is my project A level standard

- Complex mathematical model
- List operations
- Dynamic generation of objects based on complex user-defined use of OOP model
- Hash tables, lists, stacks, queues, graphs, trees or structures of equivalent standard
- Simple OOP model
- Generation of objects based on simple OOP model
- Single-dimensional arrays
- Simple mathematical calculations (eg average)
- Appropriate choice of simple data types

DOCUMENTED DESIGN

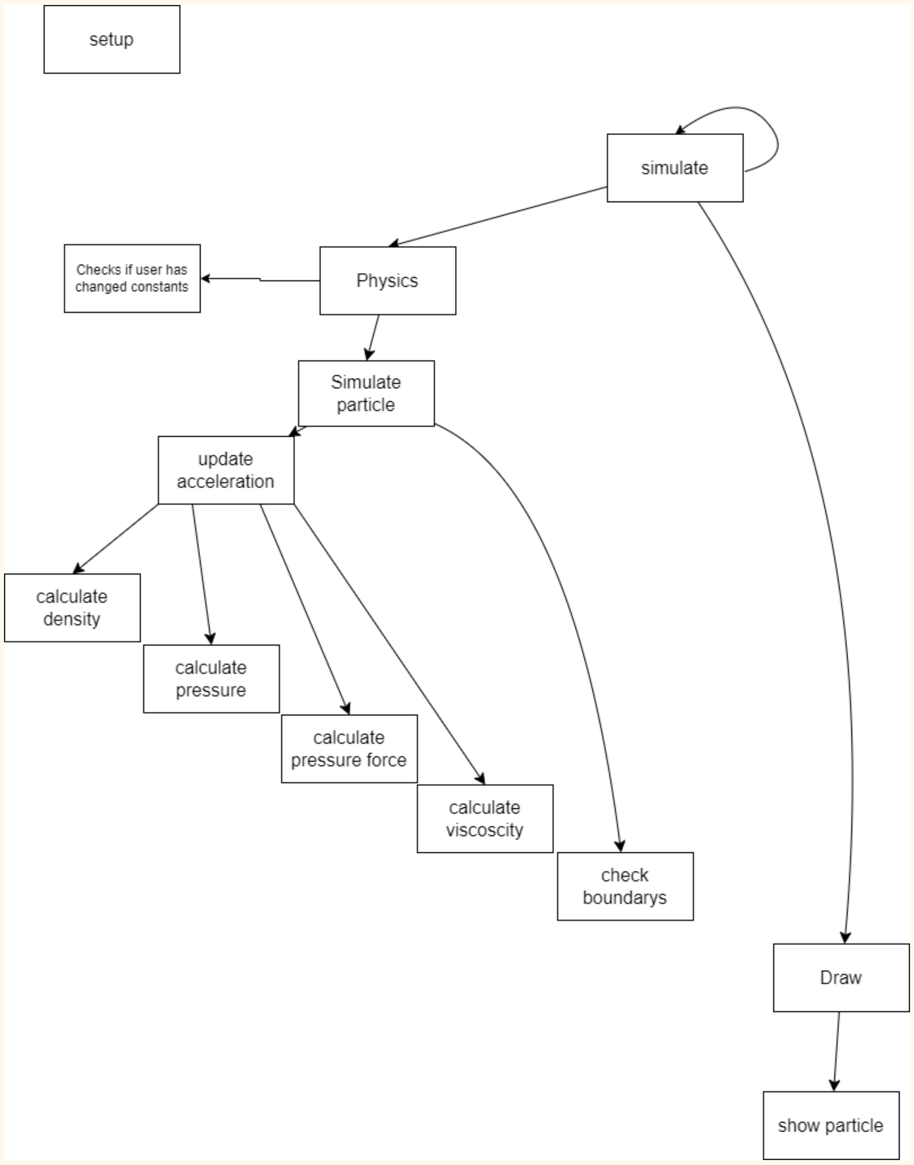
Overview of how system works

My system will be interactive so that students can change different parameters and change the behaviour of the fluid. The system is supposed to be easy to use so will include simple sliders and toggle buttons so as to not overwhelm the user with lots of code. Below is an idea of what I want my final design to look like.


SPH Fluid Simulation
Mouse controls
Mouse influence Radius
Grab/hold or spawn/delete particles
Allows the user to interact with the fluid by toggling the influence and functionality flicking button left allows to grab and hold liquid and to the right allows to spawn and delete particles
Allows the user to pause the simulation and go through it frame by frame
Pause/Next Frame
Water simulation
This container is where the fluid will be contained in
Allows user to change the starting amount of particles and restart the simulation
Fluid controls. presssure multiplier changes how quickly the particles change direction viscosity changes how thick the substances is and toggle speed colors displays the speed of particles based on color
Pressure multiplier
Viscosity
Toggle speed colors
Number of Particles
1000
Restart simulation

Modular structure

This is the modular structure of the internal system and shows how the functions interact with each other. It goes from top to bottom to show the order when the functions are called and are connected by arrows to show which function calls others functions or flows into it.



Description of each module

The check boundaries function:

checks if the ball is inside the simulation and if it isn't it will reverse the velocity and change position to inside.

Draw:

Iterates through all the objects on the screen and draws them onto the screen at their coordinates.

Calculate density:

Sums up all the nearby particles mass multiplied by their influence and equates it to the particle's density.

Calculate Pressure:

Uses the particle density and the tait equation to calculate the pressure and when density is too high the particles will spread out.Pressure is capped so it cannot go below zero to avoid numerical instability

$Pressure = max(k(density - restdensity), 0)$

Calculate Pressure force:

Uses the derivative of the attribute approximation function and the plug in the shared pressure which is the average pressure of the current particle and the one being iterated making sure to use the spiky kernel

Calculate Viscosity force:

Using the second derivative of the attribute equation we plug in the relative velocity of the current particle and multiply the result by a constant multiplier

Form structure

Forms are where information is passed into the system by the user and then updates the values of the system. The forms are going to be in the form of sliders or toggles for simplicity and the forms will be processed either before the physics function or during the physics function.

Mouse controls (Modify user interactions)

The mouse strength can be toggled by changing the influence radius and then further changed by adjusting the options by having it to the left the mouse can push and pull the liquid and having it to the right the mouse can spawn or delete particles.

The mouse controls have now been changed so you control a ball that you can collide with the fluid to create ripples .This ball can be increased or decreased to simulate obstacles which helps with are extension objective of putting items onto the fluid.

Fluid controls (sliders and toggles)

Pressure multiplier

The pressure multiplier determines how quickly the fluid responds to changes in density so a higher pressure will make the fluid a lot quicker. The pressure value will be checked once every function call but if it gets too slow I will change it every time the value changes.

Viscosity

The viscosity multiplier changes how fast the particle's speed is decreased; it stops high velocity particles from being next to low velocity particles; the effect on this visually makes the fluid thicker.

Toggle speed colours

This changes the colour of the particles based on speed

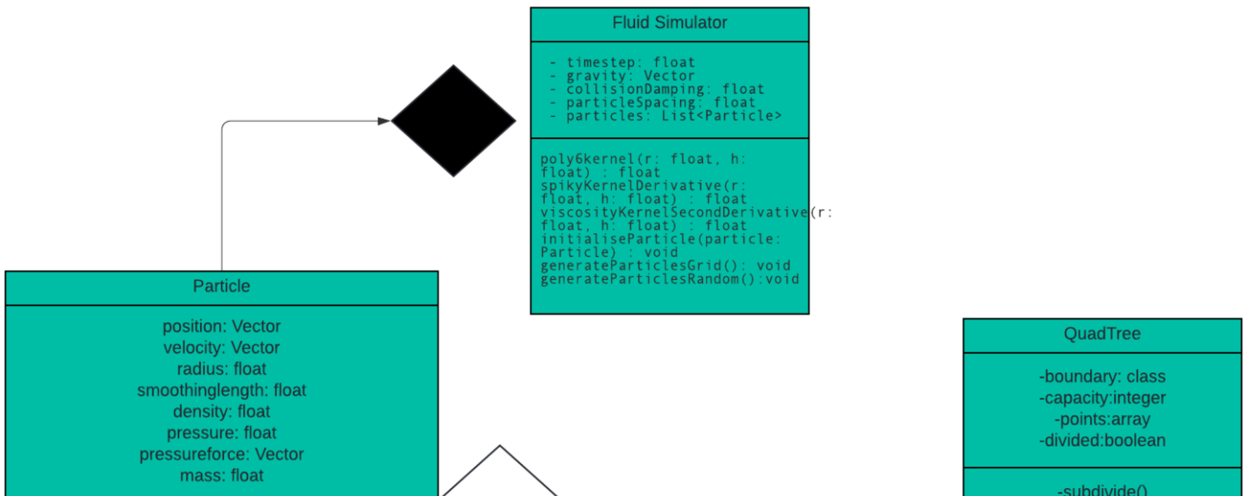
Frame settings

Pauses the simulation and allows the user to step through each frame bit by bit to see how it works

Number of Particles

Takes an integer value and everytime the simulation is reset it will use that as the starting particle amount

Class Diagrams


Fluid Simulator
- timestep: float
- gravity: Vector
- collisionDamping: float
- particleSpacing: float
- particles: List<Particle>
poly6kernel(r: float, h: float) : float
spikyKernelDerivative(r: float, h: float) : float
viscosityKernelSecondDerivative(r: float, h: float) : float
initialiseParticle(particle: Particle) : void
generateParticlesGrid(): void
generateParticlesRandom():void
Particle
position: Vector
velocity: Vector
radius: float
smoothinglength: float
density: float
pressure: float
pressureforce: Vector
mass: float
QuadTree
-boundary: class
-capacity:integer
-points:array
-divided:boolean
-subdivide()

}

Designing the Key Algorithms / Fluid simulation algorithms(pseudocode)

Quad tree algorithm

This algorithm takes a point and checks if it is in a region if it is and there's no duplicates it will check if the cell is at the max size if it is not at the max size it will insert if it is it will split into quarters and repeat and check if it's in the region and if the cell is at max capacity.

```
subdivide() {
  let x = this.boundary.x;
  let y = this.boundary.y;
  let w = this.boundary.w;
  let h = this.boundary.h;
  let ne = new Rectangle(x + w / 2, y - h / 2, w / 2, h / 2);
  this.northeast = new QuadTree(ne, this.capacity);
  let nw = new Rectangle(x - w / 2, y - h / 2, w / 2, h / 2);
  this.northwest = new QuadTree(nw, this.capacity);
  let se = new Rectangle(x + w / 2, y + h / 2, w / 2, h / 2);
  this.southeast = new QuadTree(se, this.capacity);
  let sw = new Rectangle(x - w / 2, y + h / 2, w / 2, h / 2);
  this.southwest = new QuadTree(sw, this.capacity);
  this.divided = true;
}
```

```
insert(point) {

  if (!this.boundary.contains(point)) {
    return false;
  }

  if (this.points.length < this.capacity) {
    this.points.push(point);
    return true;
  } else {
    if (!this.divided) {
      this.subdivide();
    }
    if (this.northeast.insert(point)) {
      return true;
    } else if (this.northwest.insert(point)) {
      return true;
    } else if (this.southeast.insert(point)) {
      return true;
    } else if (this.southwest.insert(point)) {
      return true;
    }
  }
}
```

Nearest neighbour algorithm

This algorithm finds all the points in a given range which can be a circular or rectangular shape by recursively going through all cells and adding their particles to the array

```
query(range, found) {
  if (!found) {
    found = [];
  }
  if (!this.boundary.intersects(range)) {
    return;
  } else {
    for (let p of this.points) {
      if (range.contains(p)) {
        found.push(p.pointdata);
      }
    }
    if (this.divided) {
      this.northwest.query(range, found);
      this.northeast.query(range, found);
      this.southwest.query(range, found);
      this.southeast.query(range, found);
    }
  }

  return found;
}
```

Calculate Density algorithm

This algorithm calculates the density of a particle given the nearby particles.

```
Function calculateDensity(particle,nearbyParticles){
  Let density = 0;
  For currentparticle in nearbyParticles:
    Distance = particle.distanceFrom(currentparticle)
    Density += currentparticle.mass * smoothingKernel(distance,smoothinglength)
  Particle.density = density
}
```

Calculate Pressure Force

This calculates how much the particle is moved by calculating the direction the force acts and then how powerful the force is.

```
Function calculatePressureForce(particle,nearbyParticles){
 Let PressureForce = Vector(0,0)
 Let Pressuremagnitude = 0;
 For currentparticle in nearbyParticles:
  Distance = particle.position.distanceFrom(currentparticle.positon);
  If Distance > 0:
   DirectionVector = (currentparticle.position.sub(particle.position))/Distance;
   Pressuremagnitude=
currentparticle.pressure*currentparticle.mass*smoothingKernelDerivative(distance,smoothingLength))/currentparticle.d
ensity
   PressureForce.add(DirectionVector.mulScalar(Pressuremagnitude))
  particle.PressureForce = PressureForce

}
```

Calculate Acceleration

Updates the acceleration of a particle

```
Function calculateAcceleration{
 Particle.acceleration = -particle.pressureforce/particle.density +
particle.viscoscityforce/particle.density+ gravity;
}
```

Updating position

This function updates the position and velocity of a single particle and calls the acceleration update function

```
Function updatevectors{
 calculateacceleration()
 Particle.velocity = particle.velocity.add(particle.acceleration.mulScalar(timestep))
 Particle.position = particle.position.add(particle.velocity.mulScalar(timestep))
}
```

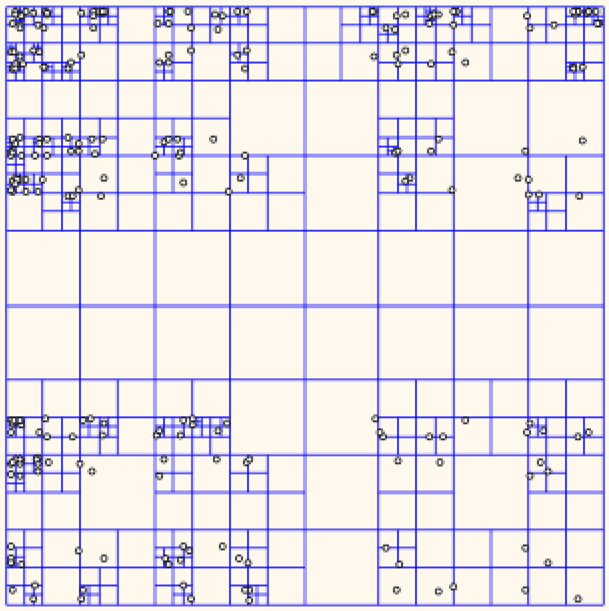
Drawing Particles onto screen

```
Function Draw(){
 For particle in particles:
  particle.show();
}
```

Key Data structures

Quad tree to find nearby particles.

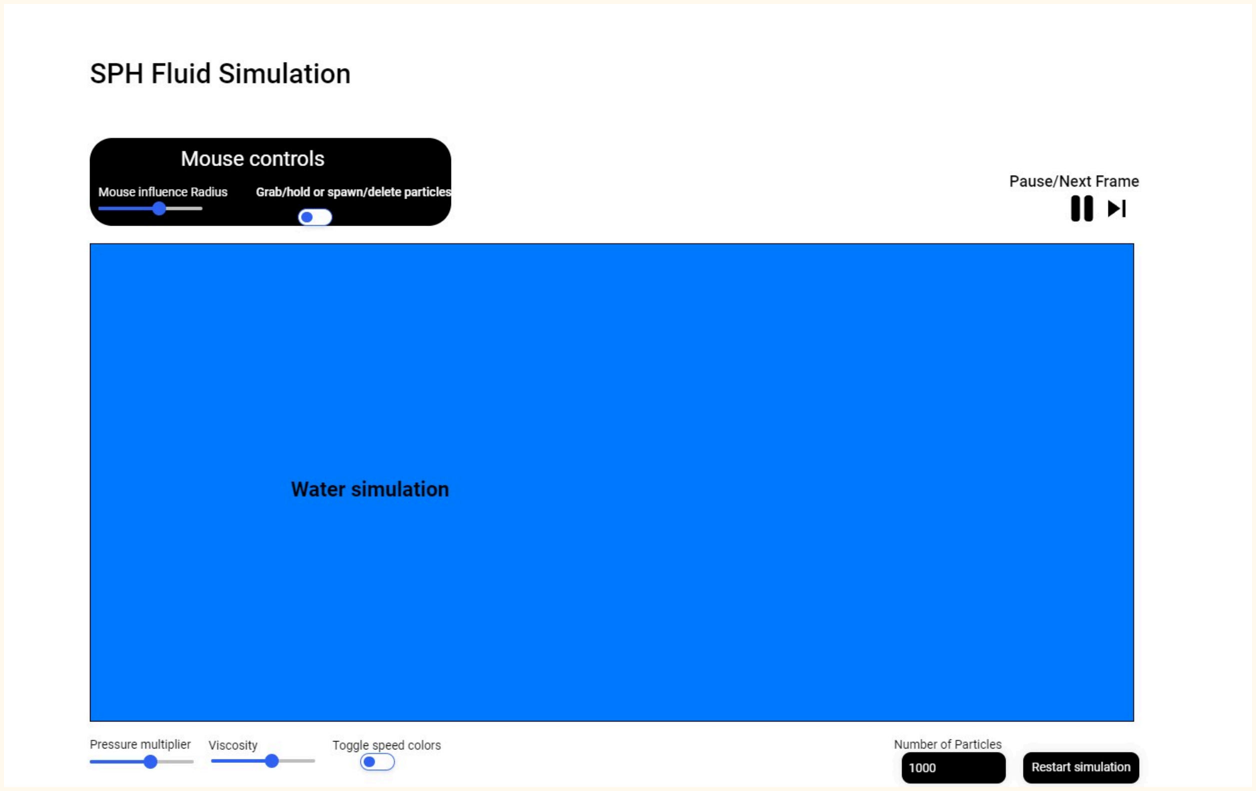
Quad trees allow us to find nearby particles quickly and easily [*Hubber, D. (2015)*]. We do this by splitting up the canvas into subdivisions of cells each with a maximum amount of predetermined particles which if exceeded the cell is split into four. To store the particle we check if the particle is in the current area if it is we then check if the area is at capacity if it isn't we then divide and repeat the process of checking if it's in the area already and dividing until there is a cell with enough space and it is then inserted to the points array of that cell. Below is a diagram showing how a quadtree operates.



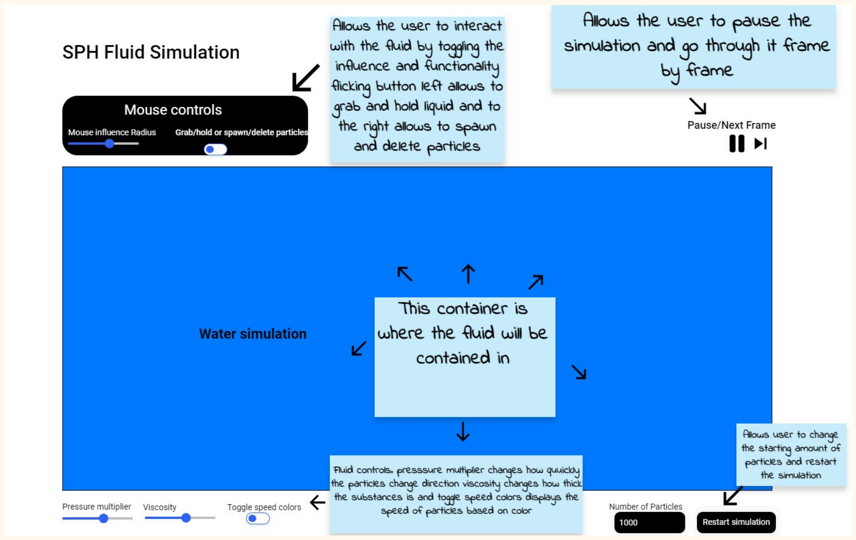
To find points in a quad tree we define a range which could be circular or rectangular that we want to search we then check all the cells in that range and their subdivisions using recursive techniques and whenever we find a point we add it to an array of found points that is then returned to the user.

UI

I have created a design for the project that shows how the user will be able to interact with the fluid simulation.



The above image is very user friendly and easy to figure out but in case any more detail is needed I have provided an annotated screen view in case I forget where to put everything or what it does. This is provided in the figure below.



Allows the user to interact with the fluid by toggling the influence and functionality flicking button left allows to grab and hold liquid and to the right allows to spawn and delete particles

Allows the user to pause the simulation and go through it frame by frame

This container is where the fluid will be contained in

Fluid controls. presssure multiplier changes how quuickly the particles change direction viscosity changes how thick the substances is and toggle speed colors displays the speed of particles based on color

Allows user to change the starting amount of particles and restart the simulation

Software requirements

- Browser preferably on a computer

Hardware requirements

Preferably lots of memory for optimal performance and a cpu with a high clock speed and many cores

Technical solution

Initialising a Particle

To initialise a particle we first need to create a new instance of the particle class we do this By assigning new Particle(x,y,radius) to a variable which generates a particle with a specific position on the canvas with a radius we also have to add the particle to the quadtree and fluid simulator classes this can be done with the below code which will generate a grid of particles on the screen and them to a quadtree.

Initialising my particles inside of my fluid simulator class allows me to easily change the values and reset the simulation this also means i can have multiple different fluids inside of my simulation which helps towards my objective of making it interactable

Generate Particles in a grid

```
function generateParticlesGrid() {
  const numRows = Math.floor(Math.sqrt(numParticles));
  const numCols = Math.ceil(numParticles / numRows);

  for (let i = 0; i < numRows; i++) {
    for (let j = 0; j < numCols; j++) {
      // Calculate the x and y coordinates based on the grid
      let particleX = (radius * i)*(-1)**i + simWidth / 2;
      let particleY = (radius * j)*(-1)**j + simHeight / 2;

      // Create a new particle and initialize it
```

```
        let particle = new Particle(particleX, particleY,radius);
        fluidSimulator.initialiseParticle(particle);

    }
  }
}
```

This creates a grid of particles centred at the origin and initialises them into the fluidSimulator which is a class with all the particles and kernel functions

Build quadtree

```
function buildqtree(){

  qtree = new QuadTree(boundary, 18);
  for (const particle of fluidSimulator.particles){
    let p = new Point(particle.position.x,particle.position.y,particle)
    qtree.insert(p)
  }
}
```

This builds a quadtree for all the particles initialised in the fluid simulator so that they are ready to perform searches on

Animation/calculation loop

When creating simulations there are two main approaches: the first is to pre-render all the frames for a predetermined amount of time and the second is to go frame by frame calculating the positions for the next frames . The real time approach is a lot better for interactive fluid simulations so i used that approach this involves a repeated loop

Simulate

—| Rebuild the quad tree

—| Perform physics calculations

—| Update canvas (draw)

—| requestAnimationFrame(simulate) this slows down code from running to fast and then runs the loop again

Animation loop

```
function simulate(){
  buildqtree()
  //console.log(qtree)
  if (!paused || skips > 0){
    physics();

    if (paused || skips > 0){
      skips -=1
    }
  }


  draw();
  requestAnimationFrame(simulate);
}
```

This animation loop is modified so the user can pause and go through each frame one by one But is essentially the same as state above

Physics Calculations

To get the position of the particles on the next frame we need to calculate their current acceleration to update their current velocity and then their current position it is important that we update the acceleration of all the particles together and then separately update their velocity and positions to avoid using calculations from the next frame on the current frame.

```
function physics(){
  let particles = fluidSimulator.particles
  for (let p of particles){

      let range = new Circle(p.position.x,p.position.y,p.smoothinglength*2)
      let others = qtree.query(range);
```

```
    p.updateacceleration(others);



  }
  for (let p of particles){
    p.checkboundarys()
    p.updatespeeds()
  }

}
```

This calculates the acceleration of the particles using the nearby particles that are in the range of double the smoothing length and then after the particles are corrected if they are outside of the canvas and then finally the speeds and positions are update

Updating acceleration

Acceleration is calculated in a 3 step process

1. Calculate Density and pressure using nearby points
2. Update the forces on the particle
3. Finally calculate the acceleration using the forces on the particle and the Navier stokes equations

Implementing Navier stokes equations

Procedure for updating a particle's acceleration given the nearby points

```
  updateacceleration(nearbypoints){
    this.calcdensityandpressure(nearbypoints);
    this.updateforces(nearbypoints);
    this.currentforce = fluidSimulator.gravity
    this.currentforce = this.currentforce.add(this.viscosityforce).sub(this.pressureforce)
    this.acceleration = this.currentforce.divScalar(this.density);

  }
```

This updates the viscosity and pressure forces and then plugs them into the Navier stokes equation to get the particle acceleration

Calculation of density and pressure using nearby points

```
  calcdensityandpressure(nearbypoints){
    let density = 0;

    for (let cparticle of nearbypoints){

      if (cparticle != self){
      let distance = this.position.distanceFrom(cparticle.position);
      density += cparticle.mass *
fluidSimulator.poly6kernel(distance,this.smoothinglength,this.numX)
      }
    }
    this.density = density ;
    let densityerror =this.density-fluidSimulator.restdensity
    if (densityerror < 0){
      densityerror = 0
    }
    this.pressure = fluidSimulator.pressuremultiplier*(densityerror);

  }
```

Iterate through the nearby points and add up the mass multiplied by the influence of the current particle influence at that position then to calculate pressure we do the difference in current density from the rest density and multiply by the pressure multiplier if density error is negative set pressure to zero this prevents too much numerical instability.

Optimisation to optimise make sure that you do not check the original particle that property is being calculated for and either make sure that density is never zero or in the future dont include values with zero density.

Calculate the fluid forces on a particle

```
    updateforces(nearbypoints){
        let pressureforce = new Vector;
        var pressurecontribution = 0;
        var viscositycontribution = new Vector;

        for (let cparticle of nearbypoints) {

            if (cparticle.particle != this.position || cparticle.density ==0){
                let distance = this.position.distanceFrom(cparticle.position);
                if (distance !== 0){

                let directionvector = cparticle.position.sub(this.position).normalize();

                pressurecontribution = ((this.pressure + cparticle.pressure)/(2*cparticle.density) *
cparticle.mass * fluidSimulator.spikykernelderrivative(distance, this.smoothinglength));
                pressureforce = pressureforce.sub(directionvector.mulScalar(pressurecontribution));
                viscositycontribution =
viscositycontribution.add(cparticle.velocity.sub(this.velocity).mulScalar(fluidSimulator.viscos
city * cparticle.mass *
fluidSimulator.viscositykernelsecondderivative(distance,this.smoothinglength)/cparticle.dens
ity));


                }

            }
        }

        this.viscosityforce = viscositycontribution;
        this.pressureforce = pressureforce;
        this.currentforce = fluidSimulator.gravity
        this.currentforce = this.currentforce.add(this.viscosityforce).sub(this.pressureforce)


    }
```

Initialise the pressure and viscosity contribution and forces to zero or zero vectors then iterate instead of plugging cparticle.pressure use the shared pressure to conserve energy as energy cannot be created or destroyed and then use shared pressure in the equation

$$\nabla P_j = \sum_{i=0}^{n} \frac{m_i}{2\rho_i} (P_i + P_j) \nabla W(\left|x - x_i\right|, h)$$

And the spiky kernel for high pressure closer to the particle make sure to multiply by the direction of the line of centres for each pressure

For viscosity We have to calculate the relative speed of the current particle with respect to the main particle and plug this into the this adds a vanishing velocity divergence if the velocities are the same and turns out to be more accurate

Update Speeds and handle boundary conditions

Updating Particles speed allows us to update the position of the particle we first have to update the particles speed and position using newton's laws of motions to update veloctiy we add the original velocity and add acceleration multiplied by time step v= u + at and for position we add the displacement travelled in that time to original position so final position = original + vt we then modify the speeds and particle position to ensure they are within the boundary this makes sure no particles fly outside of the region.

```
for (let p of particles){

  p.updatespeeds()
  p.checkboundarys()

}
```

Update Speeds

Method inside the Particle class

```
updatespeeds(){
    this.velocity = this.velocity.add(this.acceleration.mulScalar(fluidSimulator.timestep))
    this.position = this.position.add(this.velocity.mulScalar(fluidSimulator.timestep));

}
```

This is an implementation of the described algorithm for updating speeds allowing a frame by frame approach. The downsides of this approach are that inside the frames something could happen that is not accounted for so a small timestep is ideal to minimise this otherwise particles may look like they are teleporting. A greater time step however greatly reduces the speed of the program.

Handle boundary cases

Method inside particle class for handling boundary cases

```
checkboundarys(){
  if (this.position.x - this.radius < 0.0) {
    this.position.x = 0.0 + this.radius;
    this.velocity.x = -fluidSimulator.collisionDamping*this.velocity.x;
  }

  if (this.position.x + this.radius > simWidth) {
      this.position.x = simWidth - this.radius;
      this.velocity.x = -fluidSimulator.collisionDamping*this.velocity.x;
  }

  if (this.position.y - this.radius < 0.0) {
      this.position.y = 0.0 + this.radius;
      this.velocity.y = -fluidSimulator.collisionDamping*this.velocity.y;
  }

  if (this.position.y + this.radius > simHeight) {
      this.position.y = simHeight - this.radius;
      this.velocity.y = -fluidSimulator.collisionDamping*this.velocity.y;
  }
 }
```

As my boundary for my canvas is a rectangle I only have to check if the top or the sides of the particle are outside my boundary and when this occurs I reverse the velocity of the particle so it is going in the opposite direction and change the position so it is just touching the boundary. I also multiply by a damping constant to simulate energy loss from wall collisions

Handle collisions with the users interactive ball
```
  let range = new Circle(userparticle.position.x,userparticle.position.y,userparticle.radius)
 let others = qtree.query(range);
 for (let p of others){
  userparticle.velocity = new Vector
  userparticle.handlecollision(p)

 }
 userparticle.velocity = new Vector
```
As we want to treat the user particle as a solid we don't need to perform any extra fluid simulation calculations on it we just need to find where it intersects other particles and perform collision calculation making sure to set its velocity to zero afterwards to avoid inaccuracies as we want the ball to a mass with zero velocity otherwise it will move out of the users mouse position.

Exploring collision calculations

Method inside of the particle class

```
  handlecollision(particle) {
    if(this != particle){

      let distance = this.position.distanceFrom(particle.position);
      let minDistance = this.radius + particle.radius;


      if(distance <= minDistance){

        let normal = particle.position.sub(this.position).normalize();
        let relativeVelocity = particle.velocity.sub(this.velocity);
        let impulse = normal.mulScalar(2 * relativeVelocity.dot(normal) / 2);


        let repulsion = normal.mulScalar( minDistance - distance);


        this.velocity = this.velocity.add(impulse);
        particle.velocity = particle.velocity.sub(impulse);
          if (!this.userball){
                this.position = this.position.sub(repulsion);
            }

          particle.position = particle.position.add(repulsion);

        }
      }


}
```

First make sure that particle is not itself then check if it is touching the current particle if it is touching then calculate the normal, relative velocity, impulse and repulsion then updating velocity and position and making sure to not change userball position as it should only be affected by the mouse

Drawing to the canvas

Every frame we update the canvas to do this we wipe the screen clean and then iterate through all the particles making them visible using the show() method which is O(n) complexity

Particle show method

```
  show(){
    if (speedcolors) {

      let speed = Math.sqrt(this.velocity.x ** 2 + this.velocity.y ** 2);


      let normalizedSpeed = speed / 3; // Assuming maxSpeed is the m


      let redComponent = Math.floor(255 * normalizedSpeed);
      let blueComponent = Math.floor(255 * (1 - normalizedSpeed));


      c.fillStyle = `rgb(${redComponent}, 0, ${blueComponent})`;
    } else {

      c.fillStyle = "#005493";
    }
    c.beginPath();
    c.arc(cX(this.position), cY(this.position), cScale * this.radius, 0.0, 2.0 * Math.PI);
    c.closePath();
    c.fill();
  }
```

This code essentially checks if a button is toggled and when its toggled it colours the particle based on how close it is to that speed this is useful for viewing the effects of viscosity as it makes nearby particles act at the same speed which causes chain effects where when the fluid at one side is interacted with those at the other end receive a force as well so we can visualise the movement of that wave using colours.
If the button is not pressed It colours each particle to a calm blue colour.
Once the fillstyle is set
It begins a path and creates an arc at the translated position from my coordinates to canvas coordinates and the scaled up radius it then multiples by 2 pi for a full circle(360 degrees)
And finally the arc is filled in with the fillstyle

Quadtree implementation

Shapes

To define regions in my quadtree i have defined two shapes a rectangle and a circle the rectangle is used mainly when creating groups of particles and circle is made for searching a specific region I have coded two methods in each shape class to check if a given point is inside the region or a given region intersects the region.

```
class Rectangle {
  constructor(x, y, w, h) {
    this.x = x;
    this.y = y;
    this.w = w;
    this.h = h;
  }

  contains(point) {
    return (point.x >= this.x - this.w &&
      point.x < this.x + this.w &&
      point.y >= this.y - this.h &&
      point.y < this.y + this.h);
  }

  intersects(range) {
    return !(range.x - range.w > this.x + this.w ||
      range.x + range.w < this.x - this.w ||
      range.y - range.h > this.y + this.h ||
      range.y + range.h < this.y - this.h);
  }


}
class Circle{
  constructor(x,y,r){
    this.x =x;
    this.y =y;
    this.r =r;
    this.rSquared = this.r * this.r;
  }
  contains(point){
    let d = Math.pow((point.x - this.x),2) + Math.pow((point.y-this.y),2)
    return d <= this.rSquared;

  }
  intersects(range){
    var xDist = Math.abs(range.x - this.x);
    var yDist = Math.abs(range.y - this.y);

    var r = this.r;

    var w =range.w;
    var h = range.h;

    var edges = Math.pow((xDist - w), 2) + Math.pow((yDist - h), 2);
    if (xDist > (r + w) || yDist > (r + h))
    return false;

    if (xDist <= w || yDist <= h)
      return true;

    return edges <= this.rSquared;
  }
```

```
}
```

Mistakes
Before adding the circle shape I would attempt to search the region around a particle using an oversized square so adding a circle greatly improved functionality and speed of the program.

Subdivisions

A quadtree is composed of smaller quadtrees using iteration to divide one of are quad trees. We need to split it into four different regions: northeast , northwest , southeast and southwest and new quadtrees are then made with that as the boundary area. Finally set this.divided equal to true so there is no repeatedly dividing the same thing cell when there is space in a quadrant.

```
  subdivide() {
    let x = this.boundary.x;
    let y = this.boundary.y;
    let w = this.boundary.w;
    let h = this.boundary.h;
    let ne = new Rectangle(x + w / 2, y - h / 2, w / 2, h / 2);
    this.northeast = new QuadTree(ne, this.capacity);
    let nw = new Rectangle(x - w / 2, y - h / 2, w / 2, h / 2);
    this.northwest = new QuadTree(nw, this.capacity);
    let se = new Rectangle(x + w / 2, y + h / 2, w / 2, h / 2);
    this.southeast = new QuadTree(se, this.capacity);
    let sw = new Rectangle(x - w / 2, y + h / 2, w / 2, h / 2);
    this.southwest = new QuadTree(sw, this.capacity);
    this.divided = true;
  }
```

As you can see we create new quadtrees for each region

Inserting points

To add points it is quite easy. We first check if the point is inside the boundary which can be done with basic logic in the previous shape methods. Once we have found the correct boundary we check if the boundary is at the max amount of points if it is then divided until it can be inserted once inserted into one of the quadrants that are not at max capacity.

```
insert(point) {

  if (!this.boundary.contains(point)) {
    return false;
  }

  if (this.points.length < this.capacity) {
    this.points.push(point);
    return true;
  } else {
    if (!this.divided) {
      this.subdivide();
    }
    if (this.northeast.insert(point)) {
      return true;
    } else if (this.northwest.insert(point)) {
      return true;
    } else if (this.southeast.insert(point)) {
      return true;
    } else if (this.southwest.insert(point)) {
      return true;
    }
  }
}
```

Find nearby particles

We check if the search range is in the boundary if it is push all the points in that boundary into the found particles repeat on all the quadrants

```
query(range, found) {
  if (!found) {
    found = [];
  }
  if (!this.boundary.intersects(range)) {
    return;
  } else {
    for (let p of this.points) {
      if (range.contains(p)) {
        found.push(p.pointdata);
      }
    }
    if (this.divided) {
      this.northwest.query(range, found);
      this.northeast.query(range, found);
      this.southwest.query(range, found);
      this.southeast.query(range, found);
    }
  }

  return found;
}
```

Problems
I found that it was time consume to return the point and then getting the data of the point so I modified my query so it always returns the data in the point to avoid confusion and extra for loops

Fluid simulator Class

This class aims to store all the constants and algorithms needed to set up the simulation. I have put in the particle generation algorithms and smoothing kernels. I have also added the constants this means in the future I could have multiple different liquids inside of on simulation by iterating over all the particles.

```
class SPHFluidSimulator{
    constructor(){
        this.timestep = 1/30;
        this.gravity = new Vector(0,50);
        this.collisionDamping =0.7;
        this.particleSpacing = 0.1;
        this.pressuremultiplier = 3.2;
        this.restdensity = 0.2;
        this.viscoscity = 6;
        this.numParticles = 1000
        this.particles = [];
```

```
}
generateParticlesGrid() {
  const numRows = Math.floor(Math.sqrt(this.numParticles));
  const numCols = Math.ceil(this.numParticles / numRows);

  for (let i = 0; i < numRows; i++) {
      for (let j = 0; j < numCols; j++) {
          let particleX = (radius * i)*(-1)**i + simWidth / 2;
          let particleY = (radius * j)*(-1)**j + simHeight / 2;



          let particle = new Particle(particleX, particleY);
          this.initialiseParticle(particle);

      }
  }


}

 generateParticlesRandom(){
  const randomGauss = () => {
    const theta = 2 * Math.PI * Math.random();
    const rho = Math.sqrt(-2 * Math.log(1 - Math.random()));
    return (rho * Math.cos(theta)) / 10.0 + 0.5;
  };
  for (let i = 0 ; i < this.numParticles ; i++){
    let particle = new Particle(simWidth*randomGauss(), simHeight*randomGauss());
    this.initialiseParticle(particle);

  }
}
poly6kernel(r,h){
  let influence = 0
  if ( r>= 0 & r<= h ){
    influence = (h**2 - r**2)**3

  }
  return (4/(Math.PI*h**8))*influence

}
spikykernelderrivative(r,h){
  let influence = 0
  if (r>= 0 & r<= h){
    influence = -3*(h-r)**2
    }
    return (30/(Math.PI*h**5))*influence


  }
  viscositykernelseconddervative(r,h){
    let influence = 0
    if (r>= 0 & r<= h){
      influence = (h-r)
    }
    return (20/(Math.PI*h**5))*influence
  }
  initialiseParticle(particle) {
      this.particles.push(particle);
  }


}
```

All the kernels are the same as mentioned earlier just written up in code make sure to use the correct dimension kernels

Interacting with the Fluid and changing Constants

One of my objects was to make the fluid interactable so i made sure to modularise my code as much as possible and use classes so values can easily be changed without having to iterate through values inside of a list I have implemented sliders and a setting panel which allows to change constants of the simulation this involves creating html sliders for the specific value we want to change and then in javascript grabbing this value and assigning it to the constant in the code

```
 <div id="settings-container">

   <div class="slider">
      <label for="viscoscity">Viscosity:</label>
```

```
        <input type="range" id="viscoscity" name="viscoscity" min="0" max="10" value="6">
    </div>
    <div class="slider">
        <label for="restdensity">Rest Density:</label>
        <input type="range" id="restdensity" name="restdensity" min="0" max="10" value=" 1">
    </div>
    <div class="slider">
        <label for="pressuremultiplier">Pressure Multiplier:</label>
        <input type="range" id="pressuremultiplier" name="pressuremultiplier" min="0"
max="18" value=" 3">
    </div>
    <div class="slider">
        <label for="gravity">Gravity:</label>
        <input type="range" id="gravity" name="gravity" min="0" max="100" value="50">
    </div>
    <div class="slider">
      <label for="ballradius">Ball Radius:</label>
      <input type="range" id="ballradius" name="ballradius" min="1" max="9" value="2" step =
0.01>
  </div>
    <div class="input-container">
      <label for="particlecolours">Toggle speed colours</label>
      <input type="checkbox" id="particlecolours" name="particlecolours" class="input-switch">

  </div>
  <div class="input-container">
      <label for="numparticles">Number of Particles:</label>
      <input type="number" id="numparticles" name="numparticles" min="0" max="1600">

  </div>
    <button id="Restart" name="Restart">Restart Simulation</button>
</div>
```

I have assigned the slider class for all sliders so i can just write one line of code to get the assign the value of a slider0

```
const sliders = document.querySelectorAll('.slider input[type="range"]');
sliders.forEach(slider => {
    const variableName = slider.id.replace('-', '_');
    window[variableName] = parseInt(slider.value);
});


sliders.forEach(slider => {
    slider.addEventListener('input', function() {
        const variableName = this.id.replace('-', '_');
        window[variableName] = parseInt(this.value);
        fluidSimulator.viscoscity = window.viscoscity
        fluidSimulator.pressuremultiplier = window.pressuremultiplier
        fluidSimulator.restdensity = window.restdensity
        fluidSimulator.gravity.y = window.gravity
        userparticle.radius = window.ballradius

    });
});
```

First I set the window variable for each slider to the default value stored so that it doesn't assign the actually html element to one of my variables and then assigner a listener so when a slider slider value is changed the subsequent variables are updated this allows for live changing of the simulation and the desired intractability .

Problems
I am not too fluent with html so i struggled with assigned values and many time the values would just be html code so took a while to fix

External Forces

I had trouble implementing a repulsive and attractive force with the mouse so switched to a different approach of making a large ball that the user can control with the mouse to observe how the fluid is affected by solids to add further creativity the ball can be expanded in size so the student can create different experiments like fluid falling on a surface or visualising how fluids will reach the same level on either side of an object i found this approach of adding a user controlled ball on a ten minute physics paper where he uses a grid based approach of simulating a fluid

```
let userparticle = new Particle(simWidth/2,simHeight/2)
userparticle.radius = 2
userparticle.userball = true
```

This creates a particle with userball set to true this means that fluid simulation equations won't be performed on it and will not be affected by gravity and only by the movement of the mouse

Problems

I encountered some problems with the ball as it would constantly bounce or flash as the speed changes so i make sure to set the speed to 0 before and after collisions so it is treated as a stationary mass which appeared to work very well and didn't have any problems with the fluid stopping its movement

LINKS TO VIDEO AND SOURCE CODE

https://www.youtube.com/watch?v=KYsOBvfGS2A

https://www.youtube.com/watch?v=LUMSm2m9vEY

https://github.com/anewland-jarman/Alex-Newland-jarman-SPH-fluid-sim-2024-NEA

Appendix

File structure

```
├───index.html
├───particle.js
├───quadtree.js
├───sketch.js
├───SPHFluidsimulator.js
├───styles.css
```

Index.html

```
<!DOCTYPE html>
<html>

<head>
  <script src="https://cdn.jsdelivr.net/gh/RonenNess/Vector2js/dist/vector2js.js"></script>
  <link rel="stylesheet" href="styles.css">
  <meta charset="utf-8" />
</head>
<h1>Sph Fluid Simulation</h1>
<div id="topcontainer">


    <div id="mediacontrols">
        <label for="mediacontrols">Pause/Next Frame</label>
        <button id="pause" name="Pause"  ><img
src="https://upload.wikimedia.org/wikipedia/commons/thumb/2/2d/Play_Pause_icon_2283501.s
vg/1024px-Play_Pause_icon_2283501.svg.png"></button>
        <button id="skip" name="skip"><img
src="https://upload.wikimedia.org/wikipedia/commons/thumb/8/83/Fast_forward_font_awesom
e.svg/1024px-Fast_forward_font_awesome.svg.png"></button>
    </div>

</div>


  <canvas  id="canvas" onmousemove="updateuserparticle(event)" width="200" height="200">
  </canvas>




</div>


  <div id="settings-container">

    <div class="slider">
        <label for="viscoscity">Viscosity:</label>
        <input type="range" id="viscoscity" name="viscoscity" min="0" max="10" value="6">
    </div>
    <div class="slider">
        <label for="restdensity">Rest Density:</label>
        <input type="range" id="restdensity" name="restdensity" min="0" max="10" value=" 1">
  </div>
  <div class="slider">
    <label for="pressuremultiplier">Pressure Multiplier:</label>
    <input type="range" id="pressuremultiplier" name="pressuremultiplier" min="0" max="18"
value=" 3">
  </div>
  <div class="slider">
    <label for="gravity">Gravity:</label>
    <input type="range" id="gravity" name="gravity" min="0" max="100" value="50">
  </div>
  <div class="slider">
   <label for="ballradius">Ball Radius:</label>
   <input type="range" id="ballradius" name="ballradius" min="1" max="9" value="2" step =
0.01>
 </div>
  <div class="input-container">
   <label for="particlecolours">Toggle speed colours</label>
   <input type="checkbox" id="particlecolours" name="particlecolours" class="input-switch">

 </div>
 <div class="input-container">
   <label for="numparticles">Number of Particles:</label>
   <input type="number" id="numparticles" name="numparticles" min="0" max="1600">

 </div>
  <button id="Restart" name="Restart">Restart Simulation</button>






</div>
<script src="particle.js"></script>
<script src="quadtree.js" ></script>
<script src="SPHFluidsimulator.js"></script>
<script src="sketch.js"></script>

</body>

</html>
```

Particle.js

```
class Particle{

    constructor(x,y){

        this.position = new Vector(x,y);

        this.velocity = new Vector(0,0);

        this.acceleration = fluidSimulator.gravity

        this.radius = radius;

        this.smoothinglength = smoothinglength

        this.mass =1;

        this.density = 1;

        this.pressure =1;

        this.userball = false

    }

    intersects(particle){


        let d = particle.position.distanceFrom(this.position)

        return ( d< this.radius + particle.radius)

    }

    updatespeeds(){

        this.velocity = this.velocity.add(this.acceleration.mulScalar(fluidSimulator.timestep))

        this.position = this.position.add(this.velocity.mulScalar(fluidSimulator.timestep));
    }

    updateacceleration(nearbypoints){

        this.calcdensityandpressure(nearbypoints);

        this.updateforces(nearbypoints);

        this.currentforce = fluidSimulator.gravity

        this.currentforce = this.currentforce.add(this.viscosityforce).sub(this.pressureforce)

        this.acceleration = this.currentforce.divScalar(this.density);




    }

    calcdensityandpressure(nearbypoints){

        let density = 0;

        for (let cparticle of nearbypoints){

            if (cparticle.userball == false){


                if (cparticle != self){

                    let distance = this.position.distanceFrom(cparticle.position);

                    density += cparticle.mass *
fluidSimulator.poly6kernel(distance,this.smoothinglength,this.numX)

                    }

                }

                this.density = density ;
            let densityerror =this.density-fluidSimulator.restdensity

            if (densityerror < 0){

                densityerror = 0

            }

            this.pressure = fluidSimulator.pressuremultiplier*(densityerror);


        }

    }


    updateforces(nearbypoints){

        let pressureforce = new Vector;

        var pressurecontribution = 0;

        var viscositycontribution = new Vector;


        for (let cparticle of nearbypoints) {




            if (cparticle.particle != this.position && cparticle.userball == false){

                let distance = this.position.distanceFrom(cparticle.position);

                if (distance !== 0){


                let directionvector = cparticle.position.sub(this.position).normalize();
                pressurecontribution = ((this.pressure + cparticle.pressure)/(2*cparticle.density) *
cparticle.mass * fluidSimulator.spikykernelderrivative(distance, this.smoothinglength));

                pressureforce = pressureforce.sub(directionvector.mulScalar(pressurecontribution));

                viscositycontribution =
viscositycontribution.add(cparticle.velocity.sub(this.velocity).mulScalar(fluidSimulator.viscoscit
y * cparticle.mass *
fluidSimulator.viscositykernelsecondderivative(distance,this.smoothinglength)/cparticle.densit
y));



                }



            }

        }


        this.viscosityforce = viscositycontribution;

        this.pressureforce = pressureforce;




    }

    checkboundarys(){

        if (this.position.x - this.radius < 0.0) {

          this.position.x = 0.0 + this.radius;

          this.velocity.x = -fluidSimulator.collisionDamping*this.velocity.x;

        }
    if (this.position.x + this.radius > simWidth) {
      this.position.x = simWidth - this.radius;
      this.velocity.x = -fluidSimulator.collisionDamping*this.velocity.x;
    }

    if (this.position.y - this.radius < 0.0) {
      this.position.y = 0.0 + this.radius;
      this.velocity.y = -fluidSimulator.collisionDamping*this.velocity.y;
    }

    if (this.position.y + this.radius > simHeight) {
      this.position.y = simHeight - this.radius;
      this.velocity.y = -fluidSimulator.collisionDamping*this.velocity.y;
    }
   }

  show(){
    if (speedcolors) {

      let speed = Math.sqrt(this.velocity.x ** 2 + this.velocity.y ** 2);
```

```
        let normalizedSpeed = speed / 3; // Assuming maxSpeed is the m



        let redComponent = Math.floor(255 * normalizedSpeed);

        let blueComponent = Math.floor(255 * (1 - normalizedSpeed));



        c.fillStyle = `rgb(${redComponent}, 0, ${blueComponent})`;

    } else {

        c.fillStyle = "#005493";

    }

    c.beginPath();

    c.arc(cX(this.position), cY(this.position), cScale * this.radius, 0.0, 2 * Math.PI);

    c.closePath();

    c.fill();

}

handlecollision(particle) {

    if(this != particle){

        let distance = this.position.distanceFrom(particle.position);

        let minDistance = this.radius + particle.radius;
    if(distance <= minDistance){


     let normal = particle.position.sub(this.position).normalize();

     let relativeVelocity = particle.velocity.sub(this.velocity);

     let impulse = normal.mulScalar(relativeVelocity.dot(normal) );




     let repulsion = normal.mulScalar( minDistance - distance);




     this.velocity = this.velocity.add(impulse);

     particle.velocity = particle.velocity.sub(impulse);

       if (!this.userball){

           this.position = this.position.sub(repulsion);

       }


     particle.position = particle.position.add(repulsion);


   }

  }
```

```
}

}
```

Quadtree.js

```
class Point {

    constructor(x, y,pointdata) {

      this.x = x;

      this.y = y;

      this.pointdata =pointdata

    }

    getpointdata(){

      return this.pointdata

    }

}


class Rectangle {

  constructor(x, y, w, h) {

    this.x = x;

    this.y = y;

    this.w = w;
    this.h = h;
  }

  contains(point) {
    return (point.x >= this.x - this.w &&
      point.x < this.x + this.w &&
      point.y >= this.y - this.h &&
      point.y < this.y + this.h);
  }

  intersects(range) {
    return !(range.x - range.w > this.x + this.w ||
      range.x + range.w < this.x - this.w ||
      range.y - range.h > this.y + this.h ||
      range.y + range.h < this.y - this.h);
  }



}
class Circle{
  constructor(x,y,r){
    this.x =x;
    this.y =y;
    this.r =r;

    this.rSquared = this.r * this.r;

  }

  contains(point){

    let d = Math.pow((point.x - this.x),2) + Math.pow((point.y-this.y),2)

    return d <= this.rSquared;


  }

  intersects(range){

    var xDist = Math.abs(range.x - this.x);

    var yDist = Math.abs(range.y - this.y);


    var r = this.r;


    var w =range.w;

    var h = range.h;


    var edges = Math.pow((xDist - w), 2) + Math.pow((yDist - h), 2);

    if (xDist > (r + w) || yDist > (r + h))

    return false;


    if (xDist <= w || yDist <= h)

      return true;
    return edges <= this.rSquared;

  }

}


  class QuadTree {

    constructor(boundary, n) {

      this.boundary = boundary;

      this.capacity = n;

      this.points = [];

      this.divided = false;

    }


    subdivide() {

      let x = this.boundary.x;

      let y = this.boundary.y;

      let w = this.boundary.w;

      let h = this.boundary.h;

      let ne = new Rectangle(x + w / 2, y - h / 2, w / 2, h / 2);

      this.northeast = new QuadTree(ne, this.capacity);

      let nw = new Rectangle(x - w / 2, y - h / 2, w / 2, h / 2);

      this.northwest = new QuadTree(nw, this.capacity);

      let se = new Rectangle(x + w / 2, y + h / 2, w / 2, h / 2);
      this.southeast = new QuadTree(se, this.capacity);
      let sw = new Rectangle(x - w / 2, y + h / 2, w / 2, h / 2);
      this.southwest = new QuadTree(sw, this.capacity);
      this.divided = true;
    }

    insert(point) {

      if (!this.boundary.contains(point)) {
        return false;
      }

      if (this.points.length < this.capacity) {
        this.points.push(point);
        return true;
      } else {
        if (!this.divided) {
          this.subdivide();
        }
        if (this.northeast.insert(point)) {
          return true;
        } else if (this.northwest.insert(point)) {
          return true;
        } else if (this.southeast.insert(point)) {
          return true;
        } else if (this.southwest.insert(point)) {
          return true;
        }
      }
    }

    query(range, found) {
      if (!found) {
        found = [];
      }
      if (!this.boundary.intersects(range)) {
        return;
      } else {
        for (let p of this.points) {
          if (range.contains(p)) {
            found.push(p.pointdata);
          }
        }
        if (this.divided) {
          this.northwest.query(range, found);
          this.northeast.query(range, found);
```

```
          this.southwest.query(range, found);

          this.southeast.query(range, found);

        }

      }


      return found;

    }









  }
```

Sketch.js

```
var canvas = document.getElementById("canvas");

var c = canvas.getContext("2d");


var simMinWidth = 20.0;
var cScale = Math.min(canvas.width, canvas.height) / simMinWidth;

var simWidth = canvas.width / cScale;

var simHeight = canvas.height / cScale;

var boundary = new Rectangle(simWidth/2, simHeight/2, simWidth, simHeight);

var numParticles = 1000;

var radius =0.1;

var restitution =1;

var smoothinglength = radius*3

var paused = false;

var skips = 0;

var speedcolors = false;


function cX(position){

  return position.x * cScale;

}


function cY(position){

  return position.y * cScale;

}

document.getElementById("pause").addEventListener("click", function(){

  paused = !paused;

});

document.getElementById("skip").addEventListener("click", function(){
  skips +=1
});
document.getElementById("particlecolours").addEventListener("click", function(){
  speedcolors = !speedcolors;


});
const sliders = document.querySelectorAll('.slider input[type="range"]');
sliders.forEach(slider => {
    const variableName = slider.id.replace('-', '_');
    window[variableName] = parseInt(slider.value);
});




sliders.forEach(slider => {
    slider.addEventListener('input', function() {
        const variableName = this.id.replace('-', '_');
        window[variableName] = parseInt(this.value);
        console.log(variableName + ": " + window[variableName]);
        fluidSimulator.viscoscity = window.viscoscity
        fluidSimulator.pressuremultiplier = window.pressuremultiplier
        fluidSimulator.restdensity = window.restdensity
        fluidSimulator.gravity.y = window.gravity
        userparticle.radius = window.ballradius
    });

});


document.getElementById("Restart").addEventListener("click", function () {


  fluidSimulator.numParticles = document.getElementById("numparticles").value

  fluidSimulator.particles = [];

  fluidSimulator.generateParticlesGrid();




  simulate();

});


function updateuserparticle(e){

  let rect = canvas.getBoundingClientRect();

  let x =( e.clientX - rect.left)/(cScale);

  let y = (e.clientY - rect.top)/(cScale);

  userparticle.position  = new Vector(x,y)


}
```

```
function buildqtree(){


  qtree = new QuadTree(boundary, 18);

  p = new Point(userparticle.position.x,userparticle.position.y,userparticle)

  qtree.insert(p)

  for (const particle of fluidSimulator.particles){

    let p = new Point(particle.position.x,particle.position.y,particle)

    qtree.insert(p)

  }

}

function physics(){

  let particles = fluidSimulator.particles


  for (let p of particles){

    let range = new Circle(p.position.x,p.position.y,p.smoothinglength)

    let others = qtree.query(range);

    p.updateacceleration(others);


  }


  let range = new Circle(userparticle.position.x,userparticle.position.y,userparticle.radius)

  let others = qtree.query(range);

  for (let p of others){
    userparticle.velocity = new Vector

    userparticle.handlecollision(p)



  }

  userparticle.velocity = new Vector








}




function draw() {

  c.clearRect(0, 0, canvas.width, canvas.height);

  userparticle.show()



  for (const p of fluidSimulator.particles){



    p.updatespeeds()

    p.checkboundarys()



    p.show()
  }


}

function simulate(){

  buildqtree()


  if (!paused || skips > 0){

    physics();

    draw();


    if (skips > 0){

      skips -=1

    }

  }






  requestAnimationFrame(simulate);

}

var fluidSimulator = new SPHFluidSimulator();
```

```
let userparticle = new Particle(simWidth/2,simHeight/2)

userparticle.radius = 2

userparticle.userball = true

let qtree;

fluidSimulator.generateParticlesGrid()


simulate();
```

SPHFluidsimulator.js

```
class SPHFluidSimulator{

    constructor(){

        this.timestep = 1/30;

        this.gravity = new Vector(0,50);

        this.collisionDamping =0.7;

        this.particleSpacing = 0.1;

        this.pressuremultiplier = 3.2;

        this.restdensity = 0.2;
    this.viscoscity = 6;

    this.numParticles = 2000

    this.particles = [];




  }

  generateParticlesGrid() {

   const numRows = Math.floor(Math.sqrt(this.numParticles));

   const numCols = Math.ceil(this.numParticles / numRows);


   for (let i = 0; i < numRows; i++) {

     for (let j = 0; j < numCols; j++) {

       let particleX = (radius * i)*(-1)**i + simWidth / 2;

       let particleY = (radius * j)*(-1)**j + simHeight / 2;



       let particle = new Particle(particleX, particleY);

       this.initialiseParticle(particle);


     }

   }
```

```
}


 generateParticlesRandom(){

  const randomGauss = () => {

    const theta = 2 * Math.PI * Math.random();

    const rho = Math.sqrt(-2 * Math.log(1 - Math.random()));

    return (rho * Math.cos(theta)) / 10.0 + 0.5;

  };

  for (let i = 0 ; i < this.numParticles ; i++){

    let particle = new Particle(simWidth*randomGauss(), simHeight*randomGauss());

    this.initialiseParticle(particle);


  }

}

poly6kernel(r,h){

  let influence = 0

  if ( r>= 0 & r<= h ){

    influence = (h**2 - r**2)**3


  }

  return (4/(Math.PI*h**8))*influence
  }

  spikykernelderrivative(r,h){

    let influence = 0

    if (r>= 0 & r<= h){

      influence = -3*(h-r)**2

    }

    return (30/(Math.PI*h**5))*influence




  }

  viscositykernelsecondderivative(r,h){

    let influence = 0

    if (r>= 0 & r<= h){

      influence = (h-r)

    }

    return (20/(Math.PI*h**5))*influence

  }

  initialiseParticle(particle) {

      this.particles.push(particle);

  }




}
```

Styles.css

```
canvas{
    border: 1px solid black;


}
button img {
    width: 20px;
    height: 20px;
}
#mousecontrols{
    background-color: black;
    border-radius: 25px;
    width: 500px;
    height: 100px;
    display: flex;
    justify-content: center;
    align-items: center;
    color: white;
}
```

```
#topcontainer {

    display: flex;

    justify-content: space-between;

}




    #settings-container {

        width: 300px;

        height: 300px;

        overflow: auto;

        border: 1px solid #ccc;

        padding: 10px;

    }




    .slider {

        margin-bottom: 10px;

    }


#userinputs {

    display: flex;

    flex-wrap: wrap;
    justify-content: space-between;

}

.input-container {

    text-align: center;

}

.input-container label {

    display: block;

    margin-bottom: 5px;

}
```