6050 Final: DFSPH Simulation

Austin Hartley

1) Project Objectives

- The objective of this final project is to showcase a 2D small-scale Divergence-Free Smoothed Particle Hydrodynamics (DFSPH) simulation in real time with user interactability.
- The DFSPH simulation will make use of a neighborhood search algorithm and boundary collisions.
- From a learning aspect, the goal was to familiarize myself with fluid dynamics and practice discretizing partial differential equations (PDE).

2) Project Description

a) Tasks

i) Review multi-variable calculus and learn Fluid dynamics/the Navier-Stokes equation.

- Isothermal Navier-Stokes equations in Lagrangian coordinates:

$$\frac{D\rho}{Dt} = 0 \quad \Leftrightarrow \quad \nabla \cdot \mathbf{v} = 0 \qquad (1)$$

$$\frac{D\mathbf{v}}{Dt} = -\frac{1}{\rho}\nabla p + \nu\nabla^2\mathbf{v} + \frac{\mathbf{f}}{\rho}, \qquad (2)$$

- To compute the particles' attributes, such as pressure, velocity, and density, the use of differential operators, gradients and Laplacian, are necessary.
- The differential operators are used to split the PDE into simpler ordinary differential equations (ODE) which have been integrated through time using semi-implicit Euler.

ii) Implement an SPH simulation.

- Before making an incompressible fluid simulation, I started off with a basic weakly compressed SPH simulation.
- This included choosing a smoothing kernel, finding the gradient to the smoothing kernel, calculating the density field, and finding the viscosity force by the Laplacian and the pressure force by the pressure gradient. The new velocity and position of each particle was then found by integrating.
- The timestep used for integrating is based off the Courant-Friedrichs-Lewy (CFL) condition to help keep the simulation stable.

iii) Render the particles as circles using Bresenham's circle algorithm.

- The particles were drawn via legacy OpenGL commands.
- All numbers were based off real-life units, and the particle's radius was based off meters, so they were converted to be drawn on the screen.

iv) Implement boundary collisions.

- The original plan was to use particle-based boundary handling, but due to the instabilities it was causing through my implementation, I had to resort to a simpler method.
- If the particle went beyond the simulation domain, its position was forced back in, and the velocity was reversed with a slight elasticity dampener.

v) Make the particles incompressible using DFSPH pressure solvers.

- DFSPH utilizes two pressure solvers, correcting density and divergence errors to ensure equations 1 from section i.
- Correcting Divergence [1]:

Algorithm 2 Divergence-free solver

function CORRECTDIVERGENCEERROR(α, $\mathbf{v}^*$)
  while $\left(\left(\frac{D\rho}{Dt}\right)_{\text{avg}} > \eta^{\text{v}}\right) \vee (\text{iter} < 1)$ **do**
    for all particles i **do** // compute $\frac{D\rho}{Dt}$
      $\frac{D\rho_i}{Dt} = -\rho_i \nabla \cdot \mathbf{v}_i^*$
    for all particles i **do** // adapt velocities
      $\kappa_i^{\text{v}} = \frac{1}{\Delta t}\frac{D\rho_i}{Dt}\alpha_i, \quad \kappa_j^{\text{v}} = \frac{1}{\Delta t}\frac{D\rho_j}{Dt}\alpha_j$
      $\mathbf{v}_i^* := \mathbf{v}_i^* - \Delta t \sum_j m_j \left(\frac{\kappa_i^{\text{v}}}{\rho_i} + \frac{\kappa_j^{\text{v}}}{\rho_j}\right) \nabla W_{ij}$

Calculate the material density derivative for each particle and use it to find the pressure force. If the material density derivative average is not within the threshold, keep adjusting the particle velocities.

- Correcting Density [1]:

Algorithm 3 Constant density solver

1: **function** CORRECTDENSITYERROR(α, $\mathbf{v}^*$)
2: **while** $(\rho_{avg} - \rho_0 > \eta) \vee (\text{iter} < 2)$ **do**
3: **for all** particles i **do** // predict density
4: compute ρ_i^*
5: **for all** particles i **do** // adapt velocities
6: $\kappa_i = \frac{\rho_i^* - \rho_0}{\Delta t^2}\alpha_i, \quad \kappa_j = \frac{\rho_j^* - \rho_0}{\Delta t^2}\alpha_j$
7: $\mathbf{v}_i^* := \mathbf{v}_i^* - \Delta t \sum_j m_j \left(\frac{\kappa_i}{\rho_i} + \frac{\kappa_j}{\rho_j}\right) \nabla W_{ij}$

Predict the density using the material density derivative and subtract it from the rest density. This value will be used to find the pressure force. If the density average – rest density is not within the threshold, keep adjusting particle velocities.

vi) Tune the parameters to make the simulation stable.

- Fluid simulations are inherently unstable due to the discretization of a PDE that has not been analytically solved yet.
- The pressure solver thresholds had to be altered or the simulation would explode resulting in NaN immediately due to the velocity errors.
- Each of the attributes of the fluid particle are also based off real world measurements. For example, the density of water is usually around 1,000 kgm^-3 and that is what was used for the simulation.

vii) Implement neighborhood search algorithm.

- The neighborhood search algorithm is needed because looping through all the particles to step one particle through time has a runtime complexity of O(n^2).
- A neighborhood can be built based off the smoothing radius, bringing the runtime down to a more linear scale, O(mn), where m is the number of particles in the neighborhood.
- A uniform grid was constructed by dividing the simulation domain into cells with length equal to the smoothing diameter. This would allow only queries into the adjacent cells to find the neighborhood for one particle.

viii) Add user interactability

- The user can hold ‘a’ or ‘d’ to move the fluid left or right.

b) Tools Used

i) The only external library that was used is GLUT for the window creation and for the OpenGL API calls.

ii) The IDE used was Visual Studios Code running on a Linux OS.
iii) OpenMP was used to parallelize most of the for loops within the simulation.

c) Workflow

i) The first thing I did was set up the window management and add a vector class to handle data calculations, as well as create a Particle class to store attributes.
ii) The rendering of circles was implemented next.
iii) One of the big tasks was implementing the basic weakly-compressible SPH simulation.
   - The boundary handling was done first as well as a simple particle falling from gravity.
   - I picked a cubic spline as the smoothing kernel and had to solve it analytically to find its gradient.
   - Then the density field and pressure were solved in order to integrate the velocities to find the new positions.
iv) Making the particles incompressible by implementing the two pressure solvers mentioned above was the next challenge.
v) I implemented the neighborhood search algorithm to enhance performance and added OpenMP calls.
vi) Lastly, I Implemented the user interactability by allowing the user to hold ‘d’ or ‘a’ to move the fluid right or left.

d) Technical Challenges

i) One of the first challenges was implementing the gradient of the smoothing kernel, as this required me to analytically solve for it and my calculus skills were a bit rusty.
ii) The next challenge was realizing I was accumulating density and other attributes each timestep, so as the simulation ran, everything was permanently increasing and causing failures.
iii) The instability with the velocity going to NaN was an ongoing problem throughout the entire implementation. This can also occur if the user makes the simulation too chaotic.
iv) The simulation units greatly influenced the stability of the simulation, making it hard to pinpoint if it was a user-tuned variable or if the function itself was calculating incorrectly.
v) The discovery of the adjacent cells during the neighborhood search.

e) Solving Technical Challenges

i) After a refresh on finding the partial derivatives for a composite function and the chain rule, I was able to solve the gradient for the smoothing kernel and implement it.

ii) This required me to reset all values each timestep, so when the next calculations are done, the attributes are not increasing forever and are only based on its current position.

iii) This problem was never truly solved, as when attempting to control the velocity, the simulation starts to behave unrealistically, which defeats the purpose of a physically-based fluid simulation. I have currently pinpointed the problem to the pressure solvers, as depending upon a particle's circumstance can cause it to rapidly increase/decrease in velocity, resulting in all the calculations blowing up. I have chosen to keep the velocity unclamped, so certain runs will result in failures and will require a restart.

iv) To find realistic numbers to be tested as parameters for the fluid simulation, I needed to do research on water itself.

v) I rewrote this algorithm a few times to try to find an efficient way to handle the edge cases and to check whether the neighbors are in the cells above or below its current cell. I found drawing this out on paper to be very beneficial.

3) Experiments

a) The first experiment was figuring out a good rest density for the pressure solver. If it was equal to 1,000 as a particle by itself is, then the simulation explodes trying to keep all the particles' density at 1,000. After recording density of small amounts of particles being near each other, I landed at 1,200 being a reasonable number for the solver to run on. A higher density causes it to no longer act as water.

b) Similar experiments were done to find the divergence error which was around 100x smaller than the rest density. Lowering this number does not seem to cause drastic changes.

c) Another focus was finding a good timestep to help regulate the CFL timestep that can run under a decent number of frames per second, to make the animation look as smooth as possible.

d) Choosing the right viscosity also influences the movement of the particles and can help keep the simulation stable and make it move similar to real water. For this simulation, it is set at 0.01.

e) At the start of the simulation, because the particles are all bunched up, they explode due to the pressure forces, as the parameter is not set for that particular instance, but for the fluid flow at the bottom of the domain.

4) Results

a) The simulation is fairly stable at around 350 particles and will continue to run in real time even at 1,000+ particles, but the chances of a run ending because of NaN increases. The more particles you add also leads to changing the parameters for the simulation.
b) The water moves very similarly to real water, in regards to the oscillations and the splashes.
c) The pressure solver does not always keep the particles perfectly incompressible, especially at the boundaries.
d) The user can interact with it and the simulation runs smoothly at 60 frames per second.

5) Technical learnings

a) I learned about discretizing the Navier-Stokes equation and implementing it in code.
b) Creating my own spatial data structure to conduct neighborhood search queries.
c) All the moving parts in creating a physics simulation, from the simulation details to the user interactability.

6) Conclusion

a) In conclusion, I created a relatively sophisticated SPH simulation using the DFSPH pressure solvers to keep the particles incompressible and creating a realistic water simulation in 2D, that allows the user to move the particles back and forth to create splashes.
b) There is also a lot that can be improved upon, such as boundary handling and increasing the stability of the velocity through more sophisticated methods, such as a complex integration scheme.

7) Appendix

- Link to a video of the fluid simulation:
    - 324 particles: https://youtu.be/JlxsmZlt64M

References

[1] Jan Bender and Dan Koschier. 2015. Divergence-free smoothed particle hydrodynamics. In Proceedings of the 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation (SCA '15). Association for Computing Machinery, New York, NY, USA, 147–155. https://doi.org/10.1145/2786784.2786796

[2] Dan Koschier, Jan Bender, Barbara Solenthaler, and Matthias Teschner. 2022. A survey on SPH methods in computer graphics. In Computer graphics forum, Vol. 41. Wiley Online Library, 737–760.