Fluid/Particle Simulation Report

Libraries Used: We used the assignment template for OpenGL for the previous assignments but removed a lot of the assets and scenes. We are still using http-server to deploy the compiled frontend code.

Background

Fluids and other particle systems in games and graphics have remained a challenge to simulate. We hope to apply physics to many particles interacting with each other in order to simulate the movement of fluids. Our goal is to create different fluids with different parameters (like water, air, fire, etc.) but in lieu of time a good objective is to model a simple fluid within OpenGL, allowing the user to define parameters such as pressure and density.

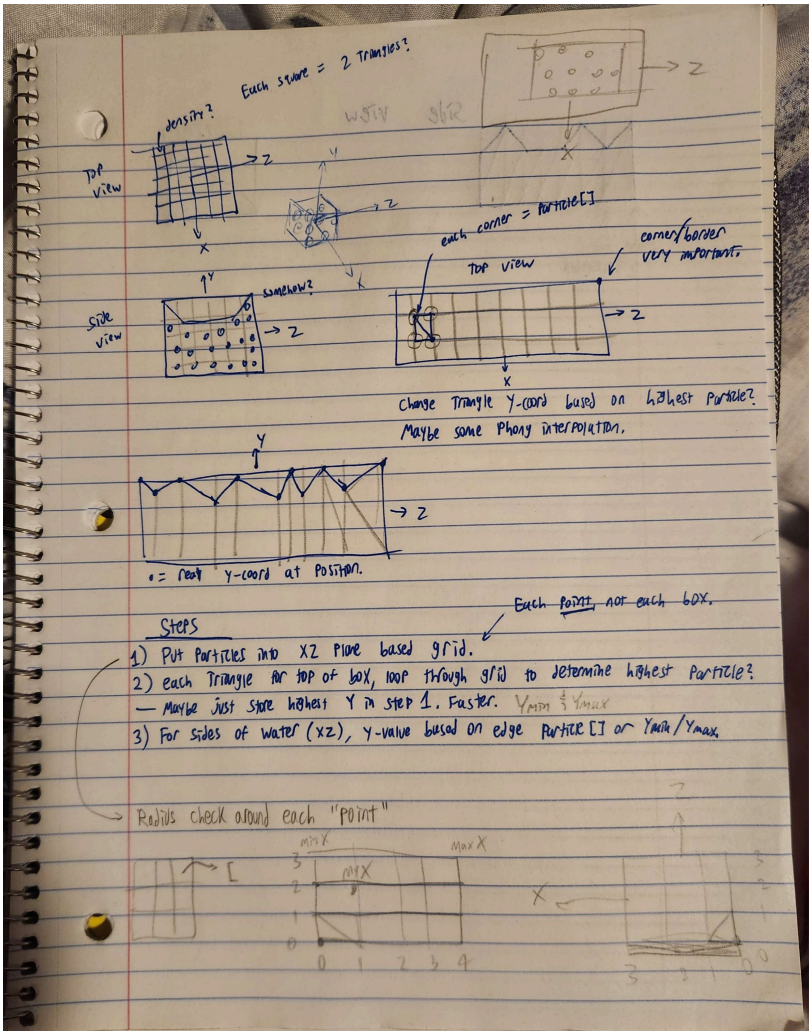
Fluids have many distinct properties such as gravity, cohesion and adhesion, the tendency to diffuse in order to evenly distribute density, Navier forces, etc. Implementing a system of particles to observe all of these effects is a daunting challenge.

Completions

- **Particle Class & Rendering**
    - We set up a lot of static variables here to be shared across all particles (for rendering) since we moved the RenderPass into this class (because we want to render particles if the user checks the box).
    - We found a guide online (David W Parker) and used that to generate our model for the sphere. Since this can be pretty expensive (spheres need a lot of triangles) we use a relatively low poly sphere but it looks good enough. The algorithm works by using sin/cos to calculate the correct coordinates for triangles and it works well enough. We also modified it to have normals for our diffuse shading.
    - For rendering, we took a create once call many (for drawing) approach. We initialize the Render Pass with initial particle positions, normals, and uniforms like the light position and projection matrices. However the main part we needed for moving these particles around all in one Render Pass will be explained in the “Rendering Movement” section.
- **Bounding Box & Rendering**
    - This wasn’t a very complex class and was the first class we created since we needed a container to base our simulation off of. Initially we hard coded bounds for the box, but later on added UI sliders that can update the values and re-create the Bounding Box if needed (only if needed since that’s more optimal). It’s a simple Rectangular Prism so there’s no crazy math here.

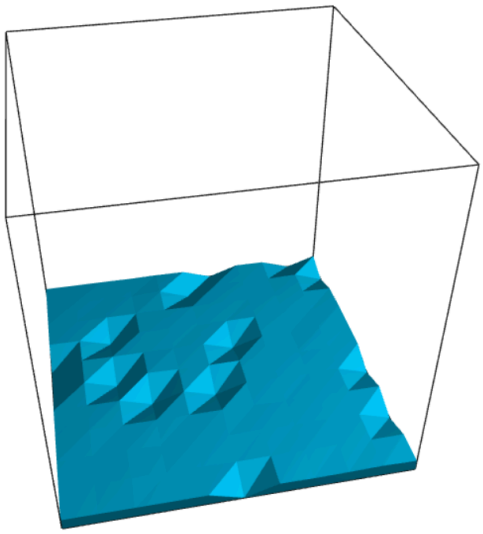
    - One thing we added here was checking if a Particle was inside the Bounding Box which was useful for Physics and ensuring the Particle doesn't fly out of the box. We take in a position and radius (particles are spheres) and compare the position to the bounds which wasn't that difficult.
    - We utilized this position check inside our simulation step and if the position after applying velocity to the current position was outside the box, we would make it so the particle stays inside bounds and redirects the velocity (and dampens it based on Bounce Falloff).
- **Rendering Movement**
    - Since we had all our Particles in a Single Renderpass, it would be pretty expensive to recreate the RenderPass each time (based on our experience) so we decided to see how efficient passing in matrices to translate particles was.
    - It turns out, despite thinking it would be slow, it turns out to run around the same speed as it did when it just had non-moving particles, so we decided to go with it. We calculate the matrices for position for each particle every frame which is definitely expensive, but modern technology saves the day. However, one issue here was by passing a Uniform we needed some way to index into it, we did this by passing in a vertIndex attribute and used some calculations to figure out how many vertices there were for each Sphere and used that to get our actual transformation matrix.
    - We used this same concept to pass in a giant velocity uniform to the shader which we used to make faster particles brighter and slower particles darker.
    - We later found out setting up a RenderPass (.setup()) was slow before because we didn't re-instantiate the RenderPass each time (when we were doing the Water Mesh), but due to time we didn't get to see whether this approach would have worked better for Particles, though theoretically it should be more efficient than our current uniform-based solution.
- **Physics**
    - There was a lot of Physics knowledge and to be clear we did figure out a lot of it by referencing Sebastian Lague's Video (Linked below) though his was in 2D while we did ours in 3D so we had to figure a lot of changes out to work for us.
    - In each simulation step we want to calculate gravity, pressure (dispersion) force, and density (cohesion) force.
    - For the density we calculate the influence of each particle on each other and figure out the density at each particle's position which lets us get the cohesion force.
    - For pressure force it's pretty similar, each particle contributes to the dispersion force at a position and we use a smoothing kernel to convert density at a position to pressure to get the overall pressure force. It's really complicated to explain by text but the video does a great job.
    - After we get the overall force (acceleration) for each particle, we check the collisions to ensure it doesn't go out of bounds (as mentioned before). Then if everything is fine we update the position and move on to the next frame.

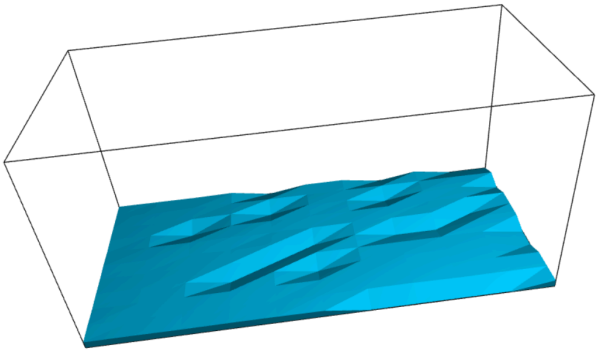
    - There's not too much detail in our explanation here since a lot of it is just math equations that are difficult to explain by text, and a lot of math which we ourselves don't understand to begin with since there's a lot when it comes to simulating fluids. However, we provide more documentation inside our Physics.ts file if you wish to see it a bit more in-depth.
- **Mouse Interactions**
    - In order to have users/players interact with our fluid, we implemented a raytracing system in order to allow players to select an area on the graph to drag the fluid around (Players can pick up blobs of fluid).
    - To do this we used the raytracing techniques from the other assignments such as cylinder ray intersection, except we projected each particle onto the line and determined if the distance was within a certain threshold.
    - If so, we add a force that pulls particles toward the mouse, allowing us to pick up parts of the fluid.
    - We also plan on having a reverse effect whereupon the particles/mesh is repelled by the mouse interaction.
- **Water Mesh Constructing & Rendering**
    - Water mesh was something we were recommended to do after our Presentation and took a decent amount of planning to get a result we were fine with.
    - We didn't want to make the process too intensive and we also wanted it to look reasonable, so we decided on what we could optimize and what needed the most work. The bottom face of the water mesh was the biggest optimization, we just created 2 triangles for the entire bottom based on the lowest y value particle.
    - Another big optimization is keeping our water mesh boundaries a square. We determine the lowest X/Z axis particles and then the greatest X/Z axis particles to iterate over.
    - We also sorted our particles into "buckets" based on their XZ coordinate values. We determine the bounds using a modifiable "Mesh Subdivisions" field and we divide the Bounding Box then index relative to the Bounding Box. This is useful for when we need to calculate the Y-values for each individual grid square that we render.
    - For the Left/Right/Front/Back triangles they are relatively simple, we just go through that axis from start to end and create triangles based on the closest bucket. (Can be seen better in the paper attached below).
    - Finally we have the top face of the mesh which is the most complicated. For each bucket on the grid we create 2 triangles, and doing this we realized for storing buckets we need to do it not based on the grid, but rather the lines on the grid (giving us 1 extra row/column essentially, again more details on the paper). Using this we can create 2 triangles with different angles depending on the Y-value of the particles and the result is pretty good in our eyes. However, we found this solution to be a bit too jumpy, so we changed it to average the y-value across all particles in a bucket and used that for the y-value of the mesh at that bucket location. This did make the frames a bit smoother but it was still a bit jumpy.

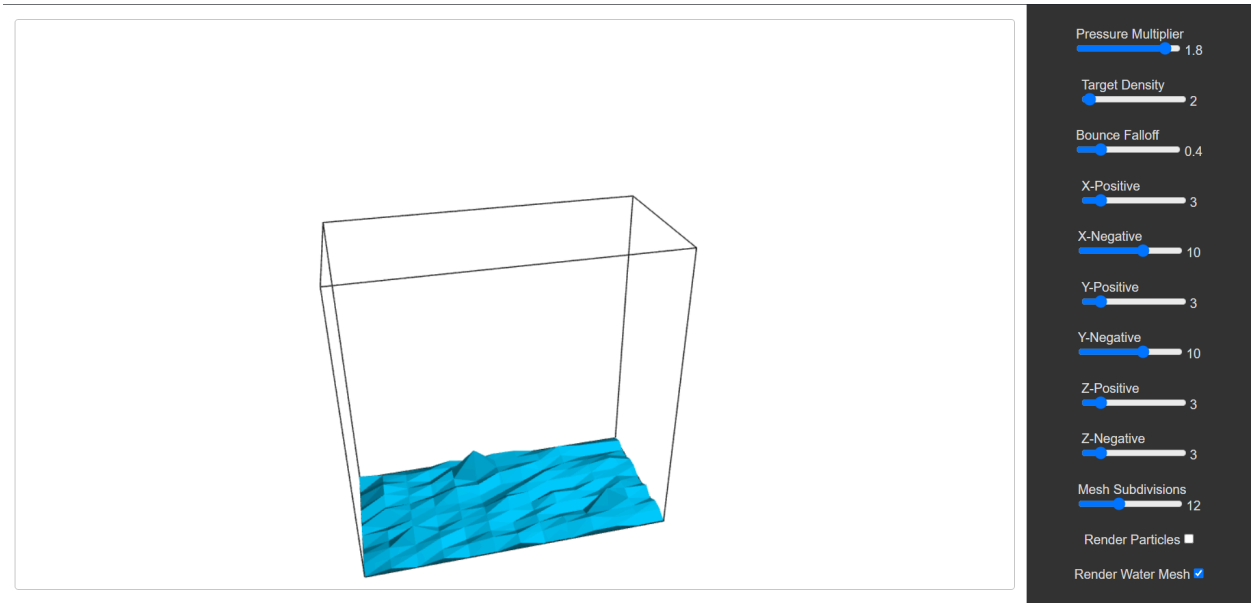
- We also calculate the normals for each triangle using cross product and the triangle coordinates we calculated. It might be inefficient but it seems to be working fast enough and it looks presentable so we decided to stop there.
- 

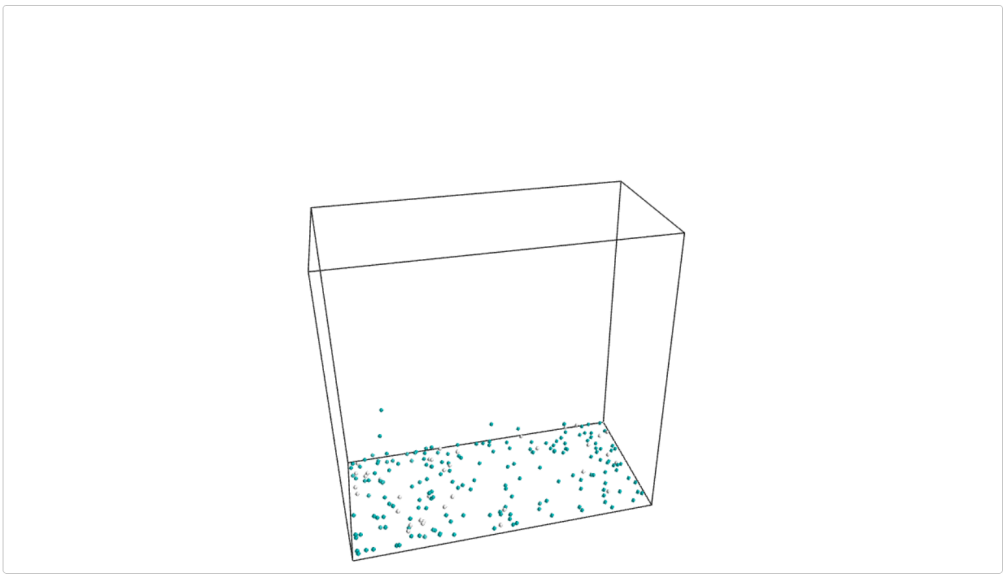
- **UI Components**
  - Our UI isn't too complex, we just have a dynamically sized Canvas and an area for our inputs. Our inputs are a bunch of sliders and checkboxes that we linked to our simulation using Class static variables which we can easily access from any file.
- **Smooth Shading**
  - This is supposed to work by default according to LearnWebGL, but this doesn't seem to work with our water mesh. We calculate the normals seemingly correctly for each vertex so we're not sure why it isn't working, but we are happy with how it looks with Flat Shading still.

Image Artifacts









Pressure Multiplier 1.8

Target Density 2

Bounce Falloff 0.4

X-Positive 3

X-Negative 10

Y-Positive 3

Y-Negative 10

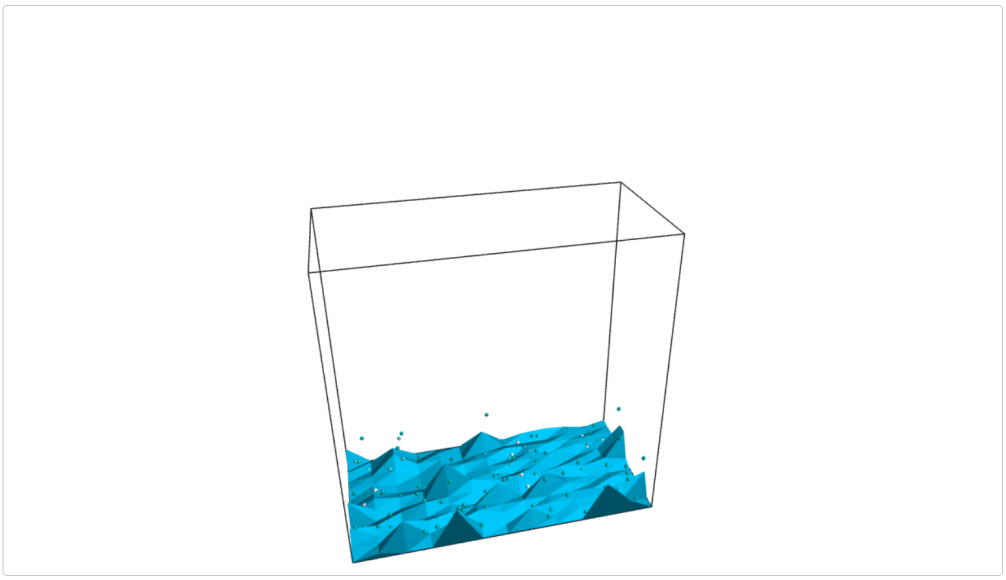
Z-Positive 3

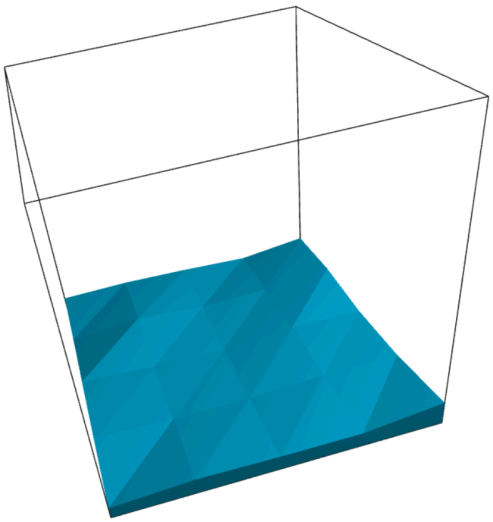
Z-Negative 3

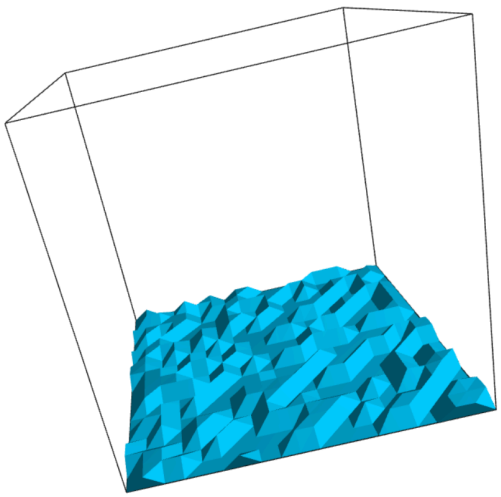
Mesh Subdivisions 12

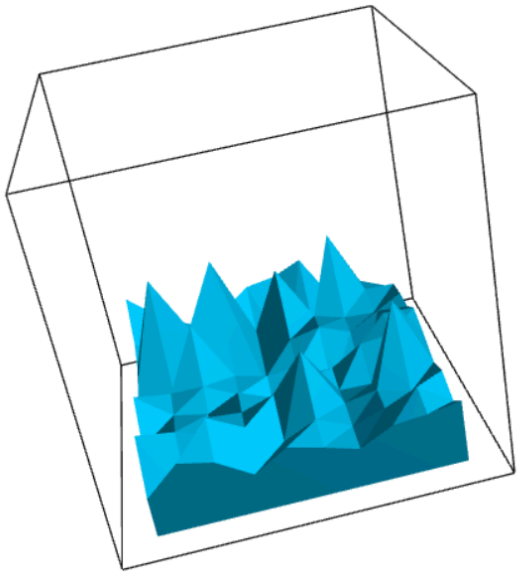
Render Particles ☑

Render Water Mesh ☐


Pressure Multiplier
1.8
Target Density
2
Bounce Falloff
0.4
X-Positive
3
X-Negative
10
Y-Positive
3
Y-Negative
10
Z-Positive
3
Z-Negative
3
Mesh Subdivisions
12
Render Particles
Render Water Mesh







Issues Encountered

- When rendering the spheres, we need to set the renderpass as one renderpass, otherwise the time efficiency is too slow. To accomplish this, we assigned each particle an index and rendered them with a position based on that index.
- We tried various methods to move particles, one of which was trying to update the positions in the shader using deltaTime. This proved unsuccessful because the shader doesn't remember the previous position of the particle.
- When passing in our transformation matrices to the vertex shader, we needed to be able to index it, and when we were indexing into it we miscalculated the vertices by a magnitude of 3, which caused many issues for us until we realized that we weren't doing / 3.
- We weren't able to get smooth shading enabled for our Water Mesh as mentioned above in the "Completions" section.

References:

- https://www.youtube.com/watch?v=rSKMYc1CQHE
- https://github.com/davidwparker/programmingtil-webgl/tree/master/0078-3d-sphere
- https://github.com/SebLague/Fluid-Sim/blob/main/Assets/Scripts/Sim%203D/Spawner3D.cs
- https://matthias-research.github.io/pages/publications/sca03.pdf
- http://www.ligum.umontreal.ca/Clavet-2005-PVFS/pvfs.pdf
- https://web.archive.org/web/20140725014123/https://docs.nvidia.com/cuda/samples/5_Simulations/particles/doc/particles.pdf