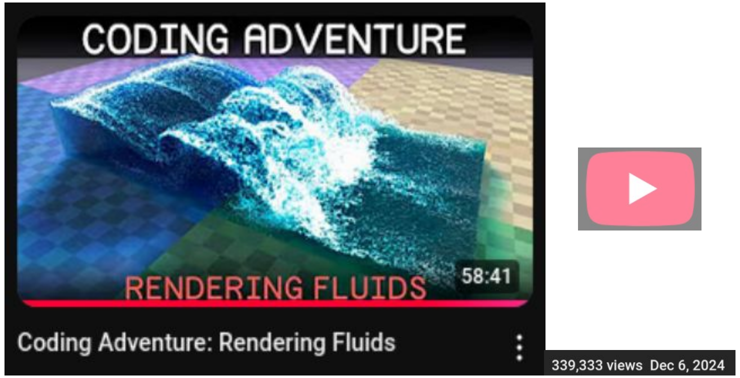
Real-Time Fluid Simulation and Shading Final Presentation

Noah Quigley-Hobson
hobsonn@purdue.edu

Where we last left off

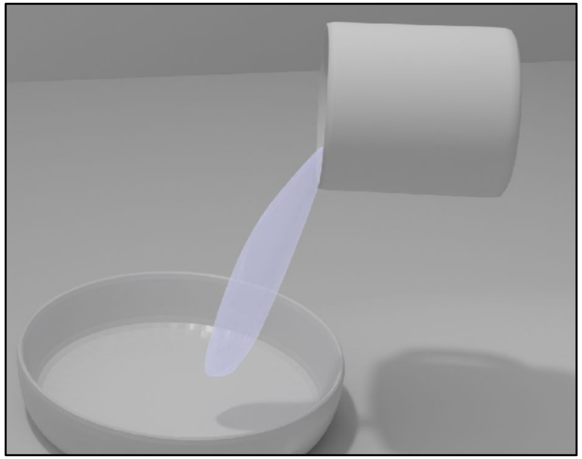
- I selected a combination of fluid simulation + ray marching
- The idea was to do this *before* Sebastian Lague, but...



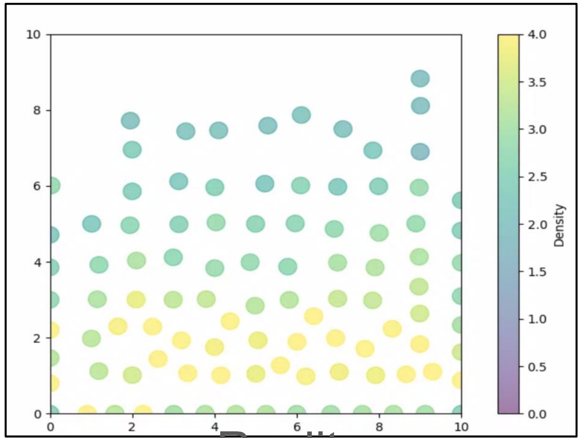
... and his is **a lot** better than mine. Screenshots will be used throughout these slides to convey what's possible.

Where my project stands

- The goal was to beat the Blender Render in both time and visual fidelity
- Pain points:
  - Navier-Stokes
    - Implementation
    - Repeatability
    - Optimization
  - Rendering
    - Implementation
    - Optimization



Expectation



Reality

Navier-Stokes

Navier Stokes Equation gives a good approximation of fluid motion

a	**Acceleration**
p	**Pressure**
ρ	**Density**
v	**Viscosity**
u	**Neighborhood**
g	**Gravity**

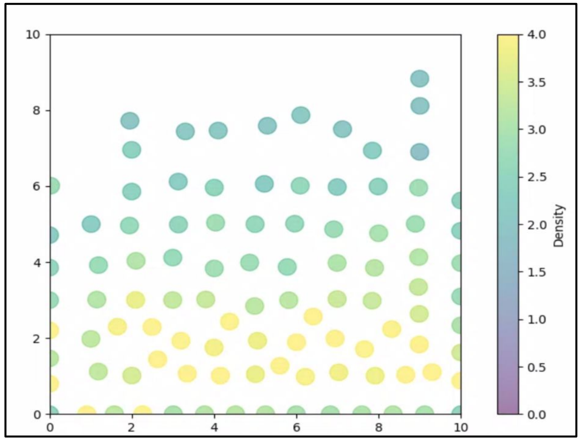
(mostly this part)

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

Apply once per timestep to integrate motion

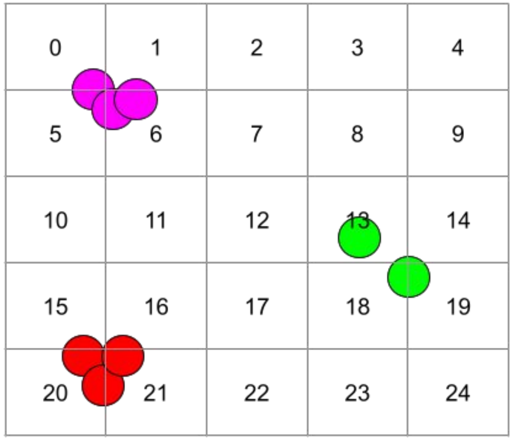
Navier-Stokes Python implementation

2D Implementation in Python was easy enough using Matplotlib for rendering



```
# Simulation loop
for step in range(NUM_STEPS):
        particles, indices, hashes = hash_and_sort(particles)
        densities = compute_density(particles, indices, hashes, SMOOTHING_LENGTH)
        forces = compute_forces(particles, indices, hashes, densities, velocities)

        # Update velocities and positions
        velocities += TIME_STEP * forces / densities[:, None]
        velocities *= 0.8
        particles += TIME_STEP * velocities

        # Enforce boundaries
        enforce_boundaries(particles, velocities, DOMAIN_SIZE)

        # Update visualization
        scatter.set_offsets(particles)
        plt.draw()

(actual code is in the project download as main.py)
```

Speedup for Navier-Stokes



On the previous slide, note the use of indices and hashes...

- Spatial hashing maps each particle to a lattice square
- Particles in neighboring squares affect each other
- Particles further do not

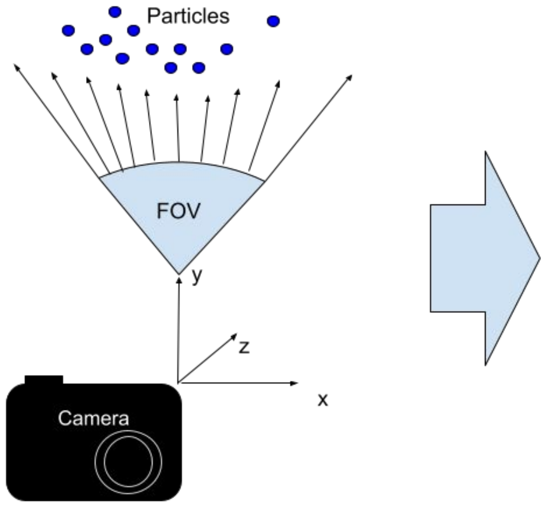
This makes the O(n^2) simulation much faster, since there are few neighboring particles (particles / squares, per particle)

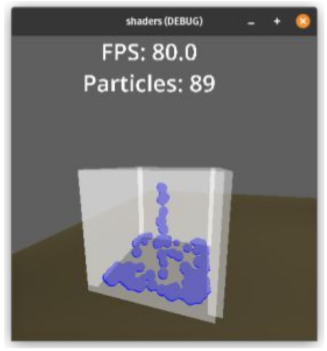
(still O(n^2), but much better)

Spatial hashing *can also speed up rendering*, but I didn't have time to implement it

Ray marching

By passing in the camera coordinates and FOV to our ray-marcher, we can combine ray-marching with traditional rasterized graphics





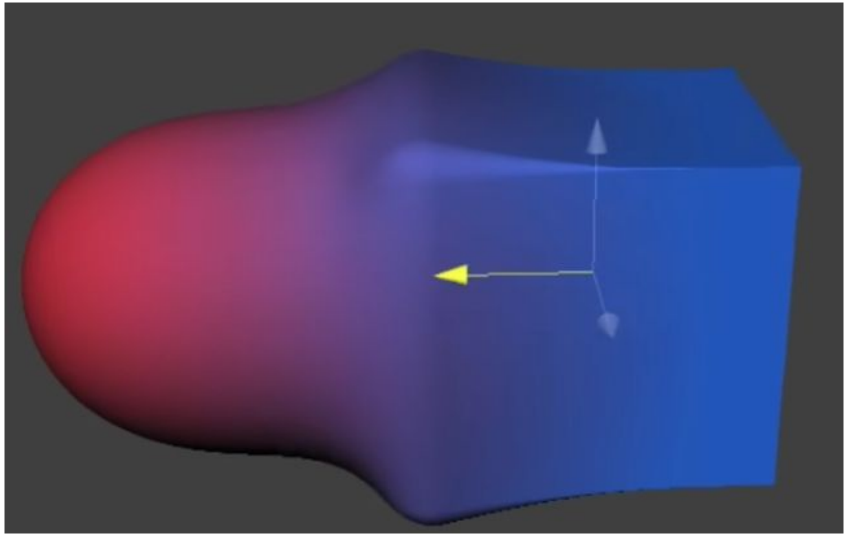
FPS: 80.0

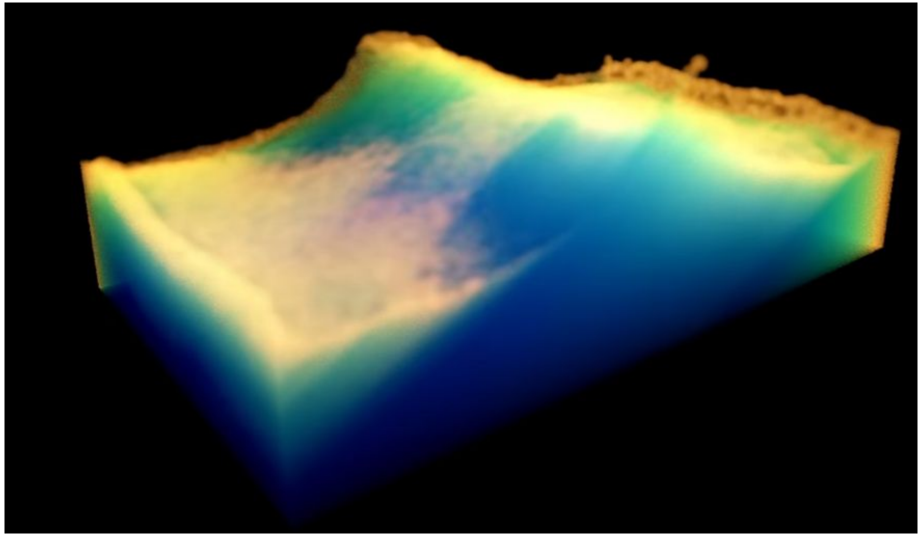
Particles: 89

Color.rgba

Ray marching benefits

- The ability to smooth objects together in a fluid-like way
- Faster than ray-tracing (sometimes)

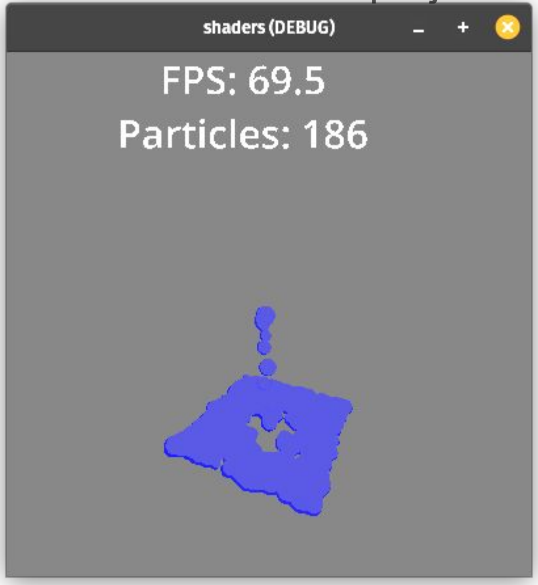




Demo: Everything together (WIP)

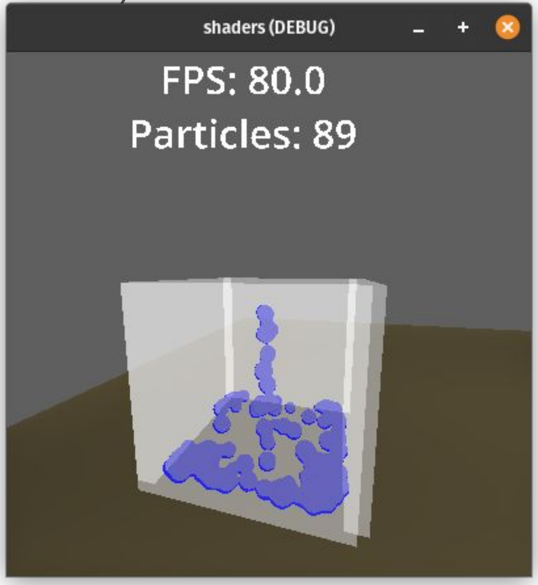
Combining fluid simulation with ray marching, we start to see fluid-like behavior

(this is where I left the project last time)


shaders (DEBUG)

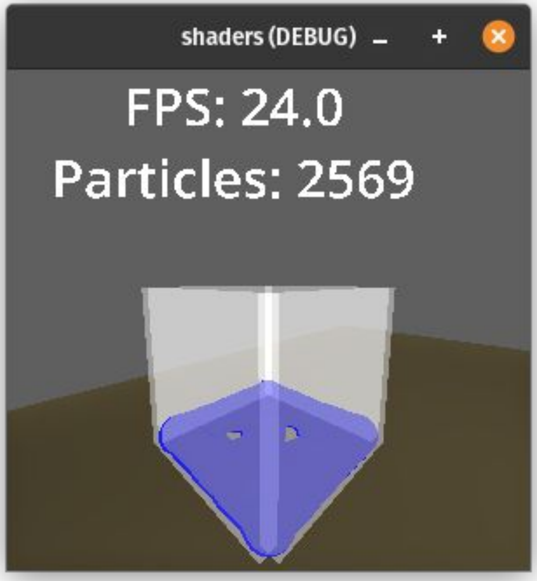
FPS: 69.5

Particles: 186


shaders (DEBUG)

FPS: 80.0

Particles: 89

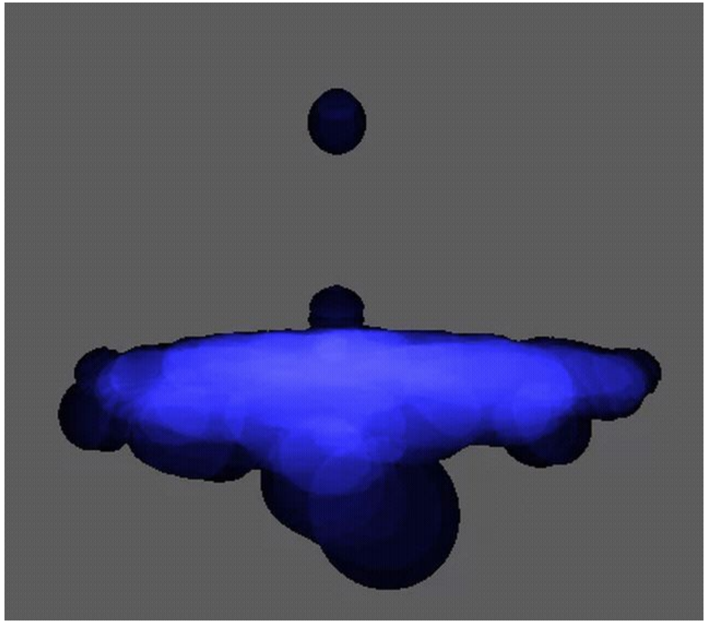

shaders (DEBUG)

FPS: 24.0

Particles: 2569

What I’ve done since the fast-forward

- Optimized 2D Navier-Stokes in Python
- Fine-tuned starting parameters
- Improvements in final rendered image in Godot
    - (but also some regressions)



Final thoughts

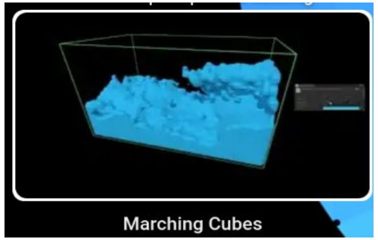
There's a lot more I wish I had time to do

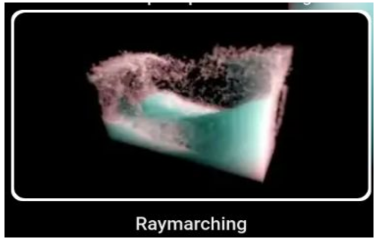
- Optimizations all around
- Improvements to rendering
- More features to game-ify the experience

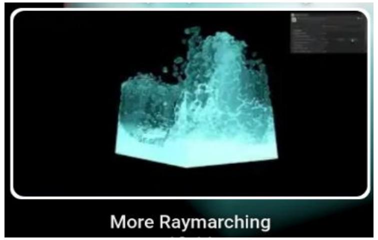
- Maybe not have used Godot because I spent too much time learning how to use it instead of C/C++ or Python
  - and I relied too much on Godot's entity management system

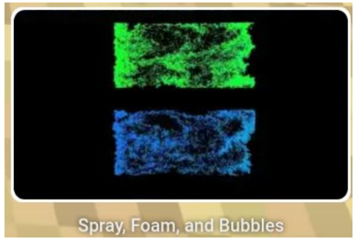
A taste of what's possible with ray marching, from Sebastian Lague

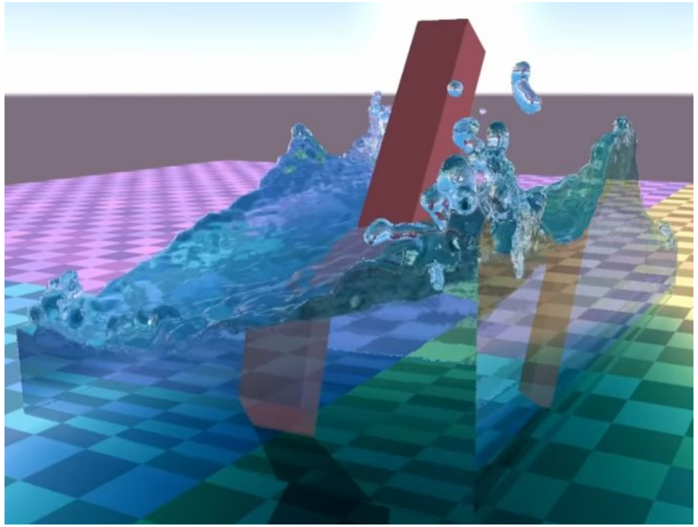
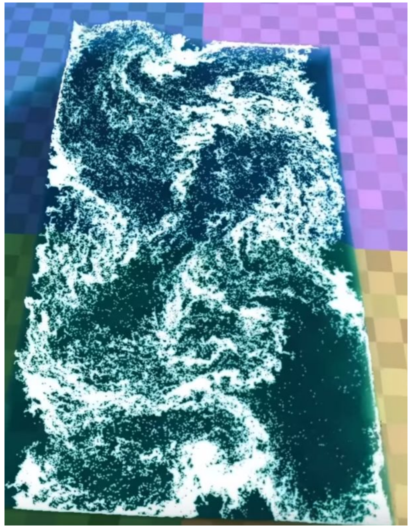
His video came out the day before this project was due, but here's some highlights









(this last image doesn't actually use ray marching, but it is possible)

The key to making this happen was **optimization and planning**

Final demo

As of right now, this is what my code looks like in Godot: