COMS 4995 Project Proposal

Zach Schuermann (zvs2002)

November 22, 2019

Goals

This project's chief goal is to create a parallelized smoothed particle hydrodynamics (SPH) solver in Haskell; its implementation will manifest in a two-dimensional fluid simulation. My twin brother has already implemented a simple 2D SPH solver in C++ [1] – my goal (as any good brother) will be to implement a faster, parallelized version in a **much** more elegant language. This project will attempt to produce a simulation similar to the screen capture in Figure 1.

Background

Fluid simulation belongs to a rather popular subset of computer graphics as its use is found throughout gaming, simulation, and animation. The solver in this project will be an pseudo-implementation of the solver described in Müller's "Particle-Based Fluid Simulation for Interactive Applications."[2] This solver is a Lagrangian (particle-based) method (as opposed to Eulerian, which is grid-based). Briefly, to understand the problem we hope to solve, some math:

$$\rho_i = \rho(\mathbf{r}_i) = \sum_j m_j \frac{\rho_j}{\rho_j} W(|\mathbf{r}_i - \mathbf{r}_j|, h) = \sum_j m_j W(|\mathbf{r}_i - \mathbf{r}_j|, h) \tag{1}$$

$$\mathbf{F}_i^{pressure} = -\nabla p(\mathbf{r}_i) = -\sum_j m_i m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right) \nabla W(|\mathbf{r}_i - \mathbf{r}_j|, h) \tag{2}$$

$$\mathbf{F}_i^{viscosity} = \eta \nabla^2 \mathbf{u}(\mathbf{r}_i) = \eta \sum_j m_j \frac{|\mathbf{u}_j - \mathbf{u}_i|}{\rho_j} \nabla^2 W(|\mathbf{r}_i - \mathbf{r}_j|, h) \tag{3}$$

The solver works by applying a smoothing kernel to perform a weighted sum of neighboring particle contributions to fluid dynamics such as pressure, viscosity, etc. If we omit surface tension, (see stretch goals) and for each particle in the simulation sum over every *other* particle, this naive implementation will yield a runtime of $O(N^2)$.

Design

The overall design will rely on an unoptimized, feature-sparse method of SPH to "solve" the state of the simulation at every timestep before rendering. The desired simulation will involve the classic "dam break" simulation in which a block of fluid is dropped into a rigid container. In the conventional imperative approach, the state of the simulation is stored as a vector of particles which are traversed at every iteration of the simulation to produce the animation. In Haskell, the implementation will rely on recursing over a list of particles to functionally transform the current state into future states of simulation. Simple rendering side-effects will be handled likely via monads (when in doubt, monad it out?) and rely on preexisting Haskell libraries for OpenGL bindings. In order to parallelize the solver, one can think of splitting the particles similar to splitting the document in parallelized word-count. This can be trivially split and subsequently parallelized. One interesting path that I plan to investigate is various means of parallelization through dividing "work" spatially and temporally, as you could potentially parallelize through time with some careful bookkeeping.

[1] GitHub for his implementation: https://github.com/cerrno/mueller-sph/blob/master/src/main.cpp

[2] http://matthias-mueller-fischer.ch/publications/sca03.pdf

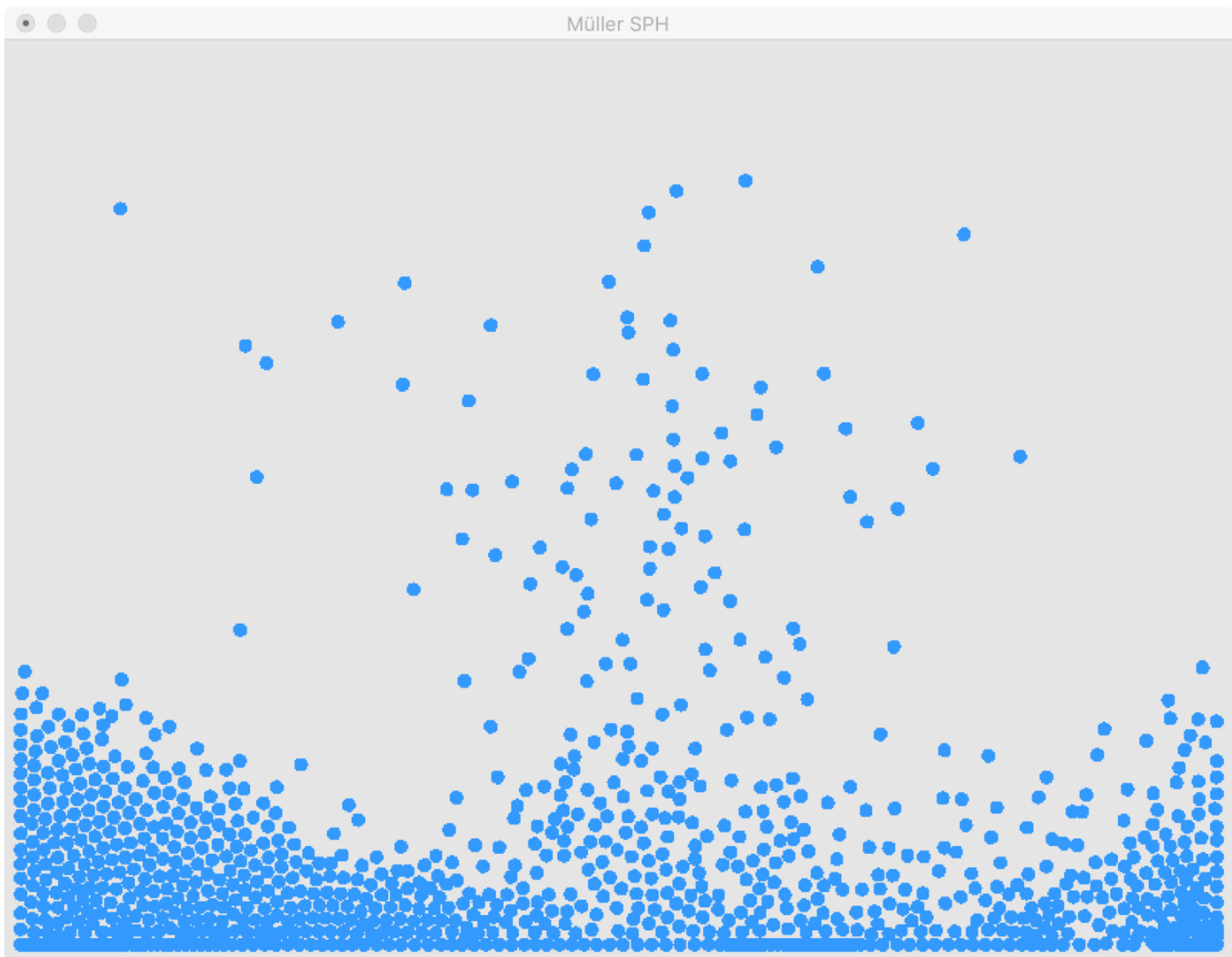

Figure 1: An OpenGL-rendered 2D particle simulation[3]

Stretch Goals

If time allows, the following goals will be included:

1. Expand solver to include surface tension
2. Move to grid-based solver, and ignore particles outside of kernel's radius of support (yielding runtime complexity of $O(N)$)