Assignment 2: Project Report

John Levy
June 20, 2023

I. Introduction

This report will present an overview of a fluid simulation project developed during labs 6-7-8 of the CSE306 – Computer Graphics course. First, we will discuss the overall concepts used to create the simulation process before moving to the specifics of the implementation, running through a short description of all the supporting classes.

The goal of the free-surface 2D fluid solver is to simulate the behavior of fluid particles subjected to incompressible Euler's equations.

To do so, we start by implementing the Sutherland-Hodgman polygon clipping algorithm to be able to generate Voronoï diagrams. These are easily extended to Power diagrams which represent the basic blocks of the fluid particles. We optimize the weights using the LBFGS solver and use the obtained solution in the Gallouet-Mérigotto scheme that prescribes fluid and air areas as well as the spring force the particles are subject to.

II. Overview of the Implementation

The Voronoï diagram class (renamed Power diagram afterwards) consists of a set of points and a set of weights associated with each of these points. Each point represents a center around which we will create a Polygon instance. Polygons are stored in a vector. Lastly, in the final version, we have a separate Polygon instance called a disk that we use to find disk to disk intersections when computing the voronoï cell of each particle during the fluid simulation.

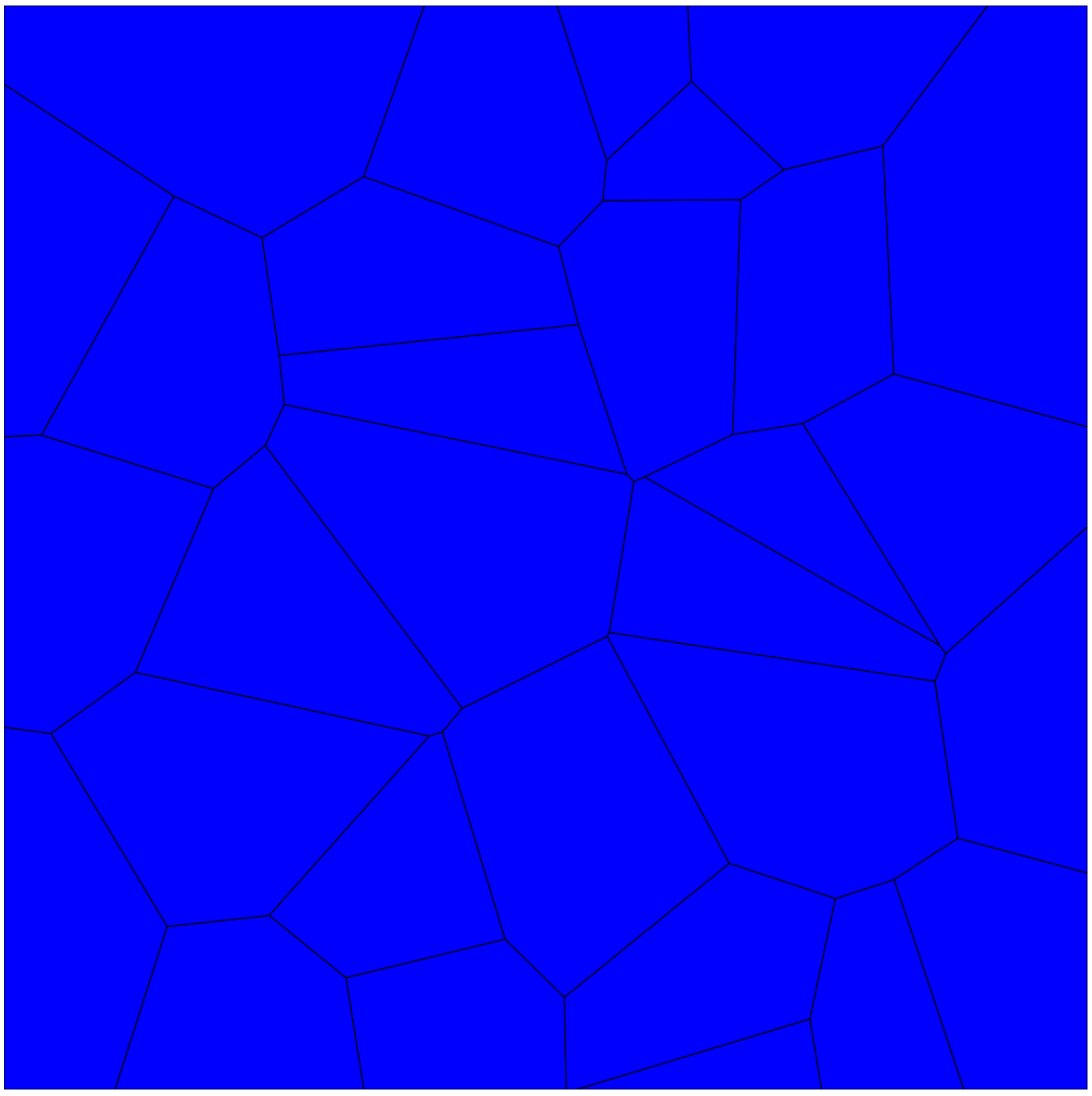
The Optimal Transport (OT in the implementation) class is the actual solver. It computes the solution using the LBFGS optimizer. The energy function we try to maximize is made up of cell area, the integrated square distance and the lambda added up for each cell (lambdas represent the volume the fluid particle takes in function of its input size, in our case, all particles have the same random size).

III. Backbone classes

<u>Polygon class:</u> In its most basic interpretation, a Polygon consists of a set of points (two dimensional vectors) representing the polygon's vertices. This basic class allows us to implement polygon specific functions needed for the computation of the energy. Those functions consist of a cell area calculator, a centroid calculator (note that we proceed using Laguerre's definition of a cell centroid), and the integrated square distance between a cell and another given point P. This function is particularly useful when computing the distance between the particles during the fluid simulation.

<u>Power diagram class:</u> As mentioned before, the power diagram is crucial to the implementation of the fluid solver. Indeed, in addition to the objects it contains, it is also in

this class that we implemented the clipping functions based on the Sutherland-Hodgman algorithm. The clipping by bisector method allows us to cut a cell based on the bisectors of the angles between its vertices, which we use when computing the voronoï cell during the solution computation. The second implementation of this algorithm is present in the clip by edge function, which works with the exact same principle, simply using the polygon’s edges rather than angles to separate the cell. This change was required when shifting the project towards the fluid solver, as fluid particles are represented by disks, and we cannot compute the intersection between two circles without considering the crossing of their respective edges. Finally, the most time-consuming function of this class is the compute method as it computes the voronoï cell for each point of the solution. Since this function is called many times by the solver we accelerated its runtime by implementing it in parallelism fashion.



Voronoï diagram, 30 cells

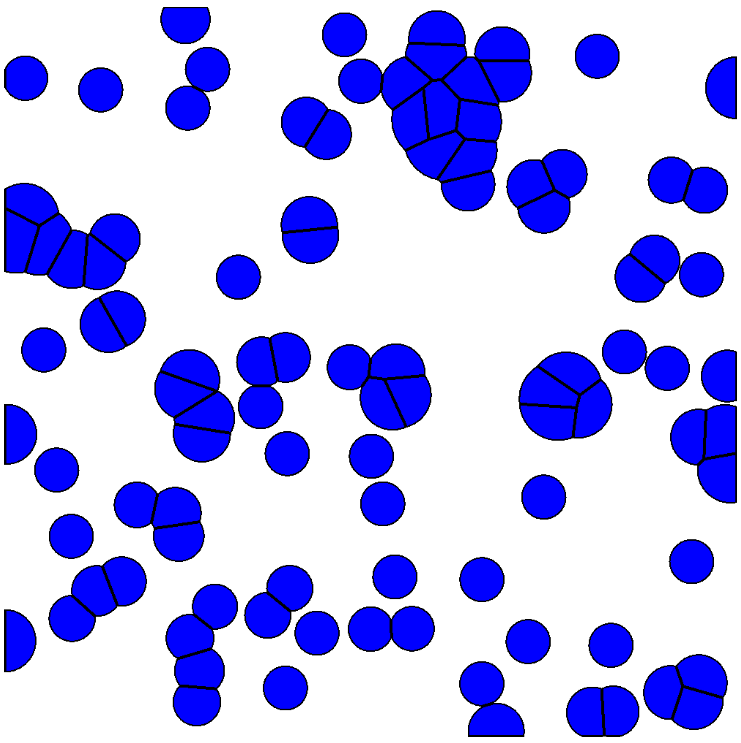
OT solver class: This class performs the optimization of the energy function by calling the LBFGS solver. We use the incompressible Euler scheme to solve the Gallouet-Mérigot model and assign the values derived from the physical differential equations to fluid cell and air areas. The components needed for the model are computed through the previously detailed classes so that the code is short, more efficient and less prone to implementation errors. Note that since LBFGS is a minimization optimizer we must use the negative value of the energy in order to maximize the function.

IV. Fluid class

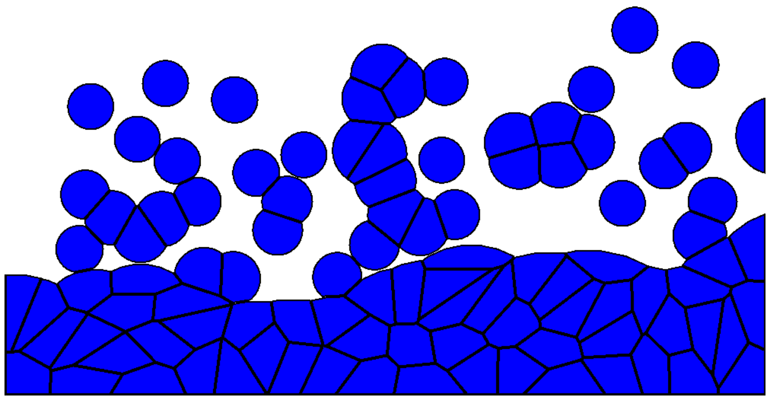
In order to run the fluid simulation, we encapsulate all of the above into a fluid class. This class contains a vector of points representing the particles (i.e. the centroids of the circles shown in the animation, randomly assigned), and a vector of vectors consisting of their initial velocities (all initialized to 0 on all axes). The full simulation is made up of individual steps computed from time t_0 to t_1000, where we consider the time step to simply be one. At each step, and for each particle we do the following:

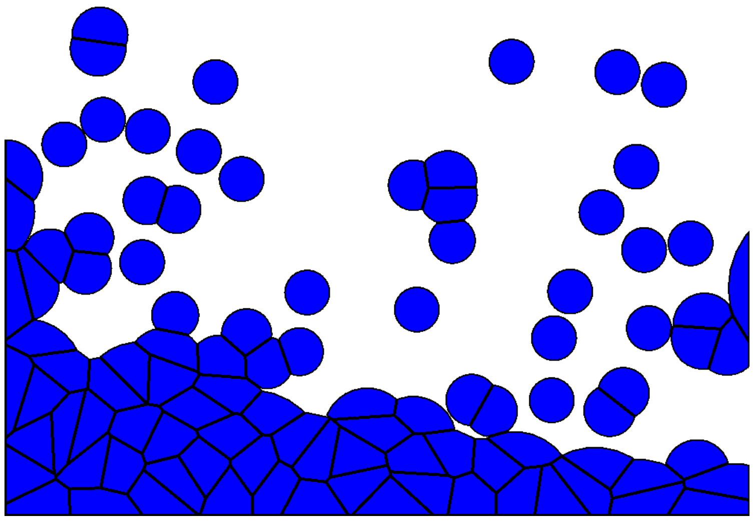
- update the velocity with the forces that apply, that is gravity and the spring force.
- update the particle by shifting its position by the newly computed velocity.
- Save the voronoï diagram of the current solution as the current frame.

Below we show three examples of frames at time t_0, t_640 and t_1000.



State of the fluid particles at the beginning of the simulation.
Particles are randomly initialized and shown as blue disks, floating in air (white area)


State of the fluid particles at time 640.
Due to gravity particles have uniformly fallen to the bottom of the frame but some are starting to bound back up.


State of the fluid particles at the end of the simulation.
The particles have fallen back to the ground after having bounced back from it as they were subject to gravity again. The accumulation in the left corner can be explained by the fact particles crossed paths meaning some where pushed in this direction.

The entirety of the steps is computed on average in 210 seconds (3 minutes, 30 seconds), which is quite efficient compared to an implementation not using parallelism in the voronoï cell computation.

To obtain a satisfying simulation, we combined all the images in a short video using the FFmpeg library (with a frame rate of around 60 fps ~ 17 seconds for 1000 frames) called animation.mp4 that can be found in the git repository of the project. When watching this video, we conclude that our solver works as expected. The particles are all firstly attracted towards the bottom of the frame (but not to one side specifically) as the biggest force acting on them is gravity. Lastly, whenever particles intersect with each other or touch the ground (bottom of the frame) they bounce back in the opposite direction according to Newton’s third law of motion.