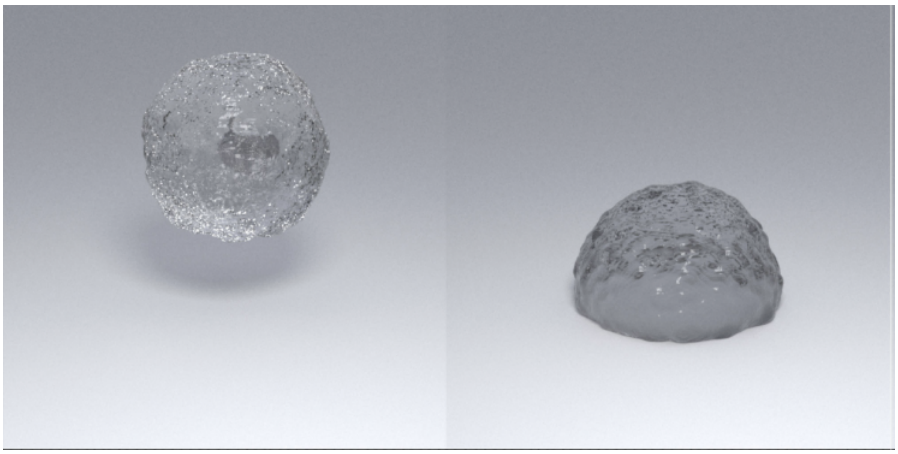
Computer Graphics - Fluid Simulator

Lucas Massot



I. Voronoi Diagrams

I.1. Class outline

In this Fluid simulator implementation, we started with a Voronoi diagram class, that would contain the full diagram. In fact we also defined other classes that would represent objects within this diagram. Here is the outline of these classes :

- **Vector** : an easier way to represent three-dimensional vectors, with a member variable that contains the coordinates. Vectors also provide methods for each vector operation that we use (e.g. addition, dot product).
- **Polygon** : an object that contains a set of vertices (represented as Vectors) that form the polygon. There is also some methods (e.g. area or centroid) for later use when optimizing the diagram.
- **Voronoi-Diagram** : one of the main classes of this implementation, which contains a points member (as a vector of Vectors), a diagram member (as a vector of Polygons). It is on this class that we will build most of the important methods of this first part.

I.2. Regular Clip by Bisector

The principle of this fluid simulation is to first generate randomly points on a 2d map of 1x1, that will become the "centers" of the Polygons in our diagram, in the sense that for each Polygon there will be a point that generates it. In our first implementation we use the function Voronoi-clip. This function takes two points P_o and P_i of our points and a Polygon of the full square, and clips this Polygon to only keep the part closer to Po than Pi before returning it. We define a function compute that uses it over every ordered pair (P_o, P_i) in points^2 such that $P_o \neq P_i$ to finally obtain our 1x1 square partitioned by Polygons which contain all the points closer to the points that generated them. In other words we obtain a Voronoi diagram.

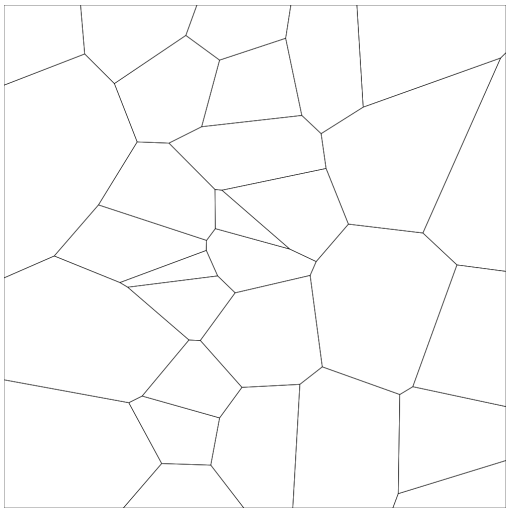


Figure I: Voronoi Diagram with 30 generating points

II. Power diagrams and Optimization

So far in our implementation, we have managed to obtain a square cropping of a Voronoi Diagram. As we mentioned, the Voronoi Diagram uses points to generate Polygons that partition the space by creating cells containing the points closer to the generating point of that cell. However, for a fluid rendering algorithm we might be more interested in a diagram that would partition the space by weights so that each cell has almost the same importance as the other given a criterion. This is called Power Diagram, which works with points and a diagram like a Voronoi diagram, but adds weights to each cell. In addition, we need to define an OptimalTransport class to compute these weights.

II.1. Class outline

- **Voronoi-Diagram** : Since both diagrams are really similar we decided to just upgrade the Voronoi-Diagram class to give it a member weights (as a vector of doubles)
- **OptimalTransport** : This class relies on optimization tools found online and that implement the Limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm to optimize a function. This class as an upgraded Voronoi-Diagram member to optimize its weights as well as methods used to do so (e.g. optimize)

II.2. Weighted Clip by Bisector

There are many ways one can use Power diagram, depending on what they want the partition of the space to rely on. For the sake of this early implementation of Power diagram (restricted to this second part) we will partition the space in cells of similar areas (this is where the member function area of the Polygons becomes useful). As we said, the implementation of the Power diagram relies on the use of weights. These weights are used in the weighted clip by bisector, that we call in our implementation Laguerre-clip. Instead of considering the Euclidean bisector between two points, we consider the Laguerre bisector which is shifted from the regular bisector using the weights. In our case it ensures that we can clip Polygons to have a similar area even though there might be part of the space where some points are closer together than others, which would result in different areas using the basic Voronoi-clip.

II.3. Optimizing the weights

To be able to use our Laguerre-clip we need to have the proper weights, that will ensure that each cell as a similar area. To do that we use our OptimalTransport class, in particular its optimize member function. Paired with evaluate, optimize allows to optimize a given function, that is find the arguments such that the function is minimal or maximal. In our case, since we want the areas to be similar using weights, we want to find the weights argument for which the function

$$g(w) := \sum_i \left(\int_{Pow_w(p_i)} (\|x - p_i\|^2 - w_i) dx + \lambda_i w_i \right) = \sum_i \left(ISD_i(w) - w_i(A_i(w) - 1/n) \right)$$

In our case we want the cells to have similar area, hence the λ_i which represent the target mass of cell i are all the same hence $\lambda_i = 1/n$, which allows us to rewrite this function g into the right-hand side of the equality by defining $ISD_i(w) := \int_{Pow_w(p_i)} \|x - pi\|^2 dx$ which is the integral squared-distance cost and $A_i(w) := \int_{Pow_w(p_i)} dx$ the area of the cell $Pow_w(p_i)$ generated by the point p_i. We then define two methods of Polygon to compute the two quantities. First, we define integral-sq-dist which iterates over triangles formed by the vertices of the Polygon that form a partition and computes the integral of the squared-

distance cost on them before adding together. In fact we have a formula to compute this integral on a triangle which makes it more simple : given a triangle $T = (c_1, c_2, c_3)$

$$\int_T \|x - p_i\|^2 dx = \frac{|T|}{6} \sum_{1 \le k \le l \le 3} \langle c_k - p_i, c_l - p_i \rangle \quad \text{with} \quad |T| := \frac{1}{2}|(c_2 - c_1) \times (c_3 - c_1)|$$

Secondly, we have a member function Area that we have already discussed and that simply relies on this formula $A = \frac{1}{2}|\sum_{i=0}^{N-1} x_i y_{i+1} - x_{i+1} y_i|$ with (x_i, y_i) the i-th vertex of the Polygon

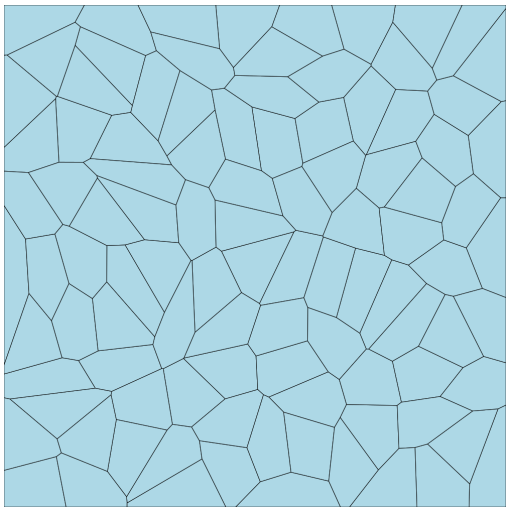


Figure II: Power Diagram with 100 generating points

Once the function g optimized and the optimal weights w computed, we can recompute our diagram with the Laguerre clip and obtain cells with similar area.

III. Fluid simulation

The idea behind our fluid simulation is to have a diagram with particles of fluid submitted to forces such as gravity and that move from one frame to the other according to Newton's laws of motion. In that sense we need to define new classes to better represent this fluid.

III.1. Class outline

- **OptimalTransport** : just like the voronoi diagram has to be updated to be a power diagram, our optimal transport needs to reflect our new goal, hence we have to upgrade

its methods optimize and evaluate. Moreover, we need to add one extra weight w_{air} for the air to the weights of its diagram

- **Fluid** : a class that represents our fluid, with member variables particles and velocities as vectors of Vectors, and its own optimal transport (along with the diagram within the optimal transport). The points of the diagram are the particles of fluid and we implement several methods to allow the simulation of the fluid (e.g. time-step)

III.2. Clip By Edge

So far we have managed to get a diagram with Polygons of similar area thanks to our implementation of the power diagram in part II, however for our fluid we want circular particles, hence we need to clip our Polygons further. We first define a unit-disk member variable in our Voronoi-Diagram as a vector of Vectors as well as N-disk member variable as an int. The latter is used to know by how many vertices we will approximate our circle, and the unit-disk contains N-disk points equally distant from the one on their left and the one on their right and that lie on the unit circle. we then implement a clip by edge in our Suth-Hod-clip that takes the Polygon we want to clip, as well as two points u and v through which the bisector goes to clip our Polygon to only keep whats on one side of the bisector. We iterate over the pairs $(u, v) = r(unit - disk[i], unit - disk[i + 1])$ to clip the Polygon into a circle of radius $r = \sqrt{w_p - w_{air}}$, where w_p is the weight corresponding to the particle in the diagram. We add this clip by edge after our clip by bisector in our compute function to obtain disks of fluid of similar area generated by our particles.

III.3. Update on the OptimalTransport

The overall structure of the OptimalTransport remains the same, however we need to update the optimize and evaluate methods to reflect the presence of air with the fluid. We first define a global variable VOL-FLUID (denoted V_{fluid} that represents the percentage of volume that is fluid (leaving the rest to be air). We can retake our function g from before and modify it

to better fit our goal. Let us first remember g :

$$g(w) = \sum_i \left(\int_{Pow_w(p_i)} (\|x - p_i\|^2 - w_i) dx + \lambda_i w_i \right)$$

we have that again we want similar area so all the lambdas for the fluid should be equal, hence $\lambda_i = V_{fluid}/n$. We also need to account for the air, hence we need to add another factor at the end, which gives us

$$g(w) = \sum_i (ISD_i(w) - w_i(A_i(w) - V_{fluid}/n)) + w_{air}(V_{desired} - V_{estimated})$$

With $V_{desired} = 1 - V_{fluid}$ the proportion of air, and $V_{estimated} = 1 - \sum_p A_p(w)$ the estimated current proportion of air given the weights w.

Using this updated version of evaluate in optimize allows us to keep our similar-area Polygons (close to circles) for each particles, while also ensuring that the total area (the sum of the area of these Polygons) tends to V_{fluid} leaving an area of $V_{desired}$ for air.

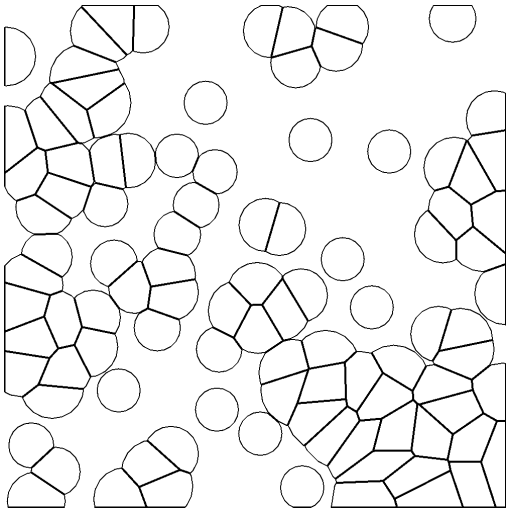


Figure III: Power diagram cliped by circles with a volume for air, generated by 100 particles

III.4. Simulating methods

Finally, we want to be able to change the state of our diagram from one frame to another, hence we need methods that update the velocities and positions of our particles based on

the forces they are subjected to. We define the time-step function, which takes a double dt which represents the time difference between the previous frame and the current frame we are making. We start by optimizing the diagram for the current set up, and then for each particle we assume that they have the same mass M, and we compute the spring force and add in the gravitational pull to compute the all-forces variable, which is a Vector denoted F. we then update the velocity v and the position p of this particle i using all-forces and Newton's laws :

$$v[i] = v[i] + \frac{dt}{M} \times F, \quad p[i] = p[i] + dt \times v[i]$$

We use this function time-step to define themain function of this whole implementation : simulation, which takes an integer pics which is the amount of frame we want in total. Then we iterate pics times by first calling step-time and then saving the current frame before moving to the next one in the next iteration. Calling simulation allows us to get all the frames necessary to create our fluid simulation video !