CSE306 Computer Graphics - Assignment 2

Francisco Moreira Machado

June 16, 2024

1 Introduction

In this project we provide a free-surface 2D fluid solver in `C++` using semi-discrete optimal transport and Gallouët Mérigot scheme. Notably, we implement the following features:

- Suntherland-Hodgman polygon clipping algorithm.
- Algorithm to find Voronoï diagram using Voronoï Parallel Linear Enumeration.
- Acceleration of Vonoroï diagram algorithm with KD-tree nearest neighbor search.
- Algorithm to find Power-diagram.
- Semi-discrete optimal transport in 2d to control cell areas of Power-diagram cells via optimization of weights. Implemented with L-BGFS.
- Semi-discrete optimal transport fluid simulator with free surfaces.

In addition to this, we also provide a color matching algorithm via sliced optimal transport and a parametrization algorithm via Tutte embedding.

2 Suntherland-Hodgman

We implement the Suntherland-Hodgman algorithm for polygon clipping in file `polygon.cpp` and function `Polygon sutherland_hodgman(Polygon &subject, Polygon &clipper)`. This function is implemented by using `Polygon Polygon::clip(Vector u, Vector N)` method of a `Polygon`, which clips the owner class with the line defined by point `u` and normal `N`, and returns the new clipped polygon.

3 Vonoroï and Power diagrams

The implementation of both Vonoroï diagrams and Power diagrams are included in `polygon.cpp`. Indeed, our algorithm for Vonoroï diagrams simply calls the function for Power diagrams after settings all weights equal to 0. This decision was made for ease of testing and consistency. The functions are named exactly as the diagram they solve. They have the following signatures:

```
std::vector<Polygon> venoroi(std::vector<Vector> &points,
                             const Polygon &default_shape);
std::vector<Polygon> power_diagrams(std::vector<Vector> &points,
                                    const double *weights,
                                    const bool use_air,
                                    const Polygon &default_shape)
```

Both functions take a set of points and a default shape which represents the bounding box that we start the clipping from. Additionally, `power_diagrams` also takes as inputs a pointer to an array of weights and a boolean indicating if we use an extra weight for air, which will be needed for the fluid simulation.

Moreover, the clipping routine is very similar to the one of Suntherland-Hodgman, heavily using `Polygon::clip`.

`power_diagrams` handles both the case of "normal" diagrams and fluid diagrams. In the latter case, if further crops the cells that would be obtained by a regular power diagram by circles of radius $\sqrt{w_i - w_{air}}$.

We accelerate the implementation by using a KD-tree to query kNN of points to clip the cells, rather than the straightforward method to clip by considering all pairs of points. The use or not of the KD-tree can be easily toggled by a flag `ENABLE_KDTREE` in the start of file `polygon.h`

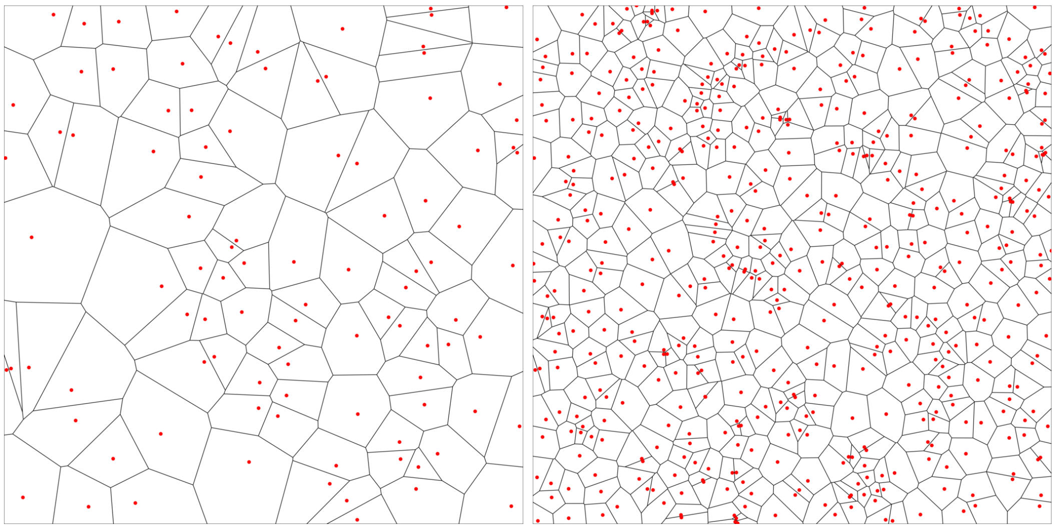

Figure 1: **Vonoroï Diagrams:** We have two examples of diagrams, the left one with 100 randomly picked points in the square, and the right one with 500 randomly picked points. The diagram with 100 points took around 0.010s with KD-tree enabled and 0.015s without it. The one with 500 points took around 0.028s with KD-tree enabled and around 0.075 without it.

4 Semi-discrete Optimal Transport

We provide the implementation of semi-discrete optimal in 2D between a set of Dirac weights and a uniform density in `ot.cpp`. Indeed, the objective of this optimization is to find suitable weights $(w_i)_{i=1}^n$ for the points $P_1, \dots, P_n$ that we generate a power diagram from, such that the areas of each cell of the diagram have area close to $(\lambda_i)_{i=1}^n$ (we assume that $\sum_{i=1}^n \lambda_i = \text{Area}(default\_shape)$).

For this optimization, we use the library liblbfgs for the L-BFGS otimization algorithm. The main function of `ot.cpp` wraps around this library and returns the Power diagram with the optimized weights:

```
std::vector<Polygon> semidiscrete_ot(std::vector<Vector> &points,
                                     const double *lambda,
                                     const double desired_volume)
```

The function takes a vector with the points we run the optimization on, as well as an array of λ_i containing the desired areas. There is an extra option: `desired_volume` in this signature which is going to be used by the fluid simulation only. It indicates how much of the total volume we want to be comprised of fluid rather than air.

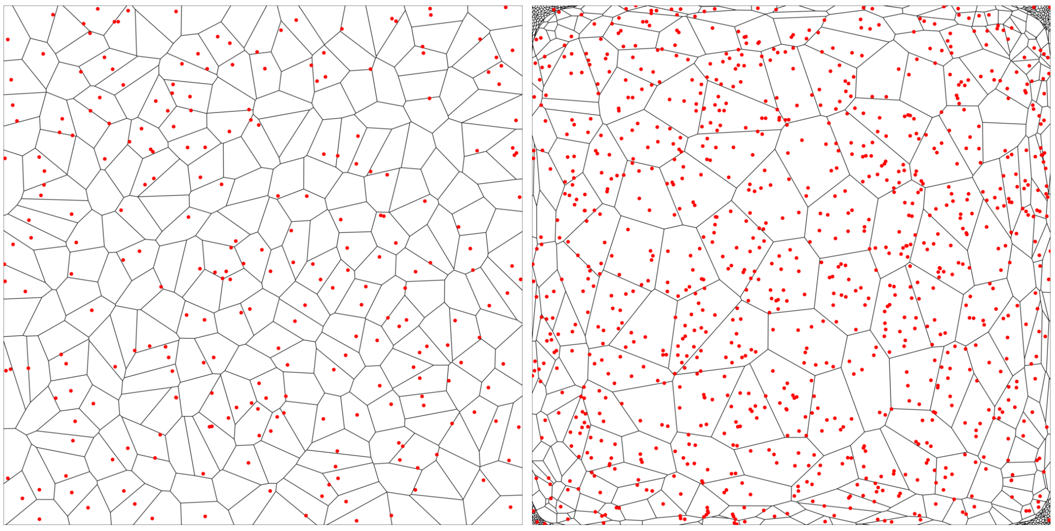

Figure 2: We have two examples of optimized power diagrams. The left one has 250 randomly generated points and we set $\lambda_i = 1/250$. The right one has 1000 randomly points picked and we set $\lambda_i \propto \exp(-\|y_i - C\|/0.02)$ where C is the center of the square. The one with 500 points took around 1.60s with KD-tree enabled and around 2.80s without it. The right diagram took around 106.7s with KD-tree optimization enabled.

5 Fluid Simulation

Our approach consists of using the Galloüt Mérigot scheme, which exploits the optimal transport from uniform to Dirac we discussed in the previous section. We make each particle have a weight m_i, a position p_i, and a velocity v_i. Additionally, we include air as a single fluid particle that takes all the space not taken by the other fluid particles. As said before, our semi discrete optimal transport and power diagrams already support this single extra air particle.

Our goal is to simulate both the position and the shape of the fluid particles. The scheme thus consists taking small time-steps to simulate it. At each step, we first computing the shape of the fluid particles at the current time via optimal transport. This is followed by computing for each fluid particle an update in its position and velocity following Euler's scheme. Let us use C_i represents the cell i-th cell of the Power diagram with the optimized weights as described before. For this, we simulate a time-step dt and do the following updates

$$F = \frac{Centroid(C_i) - p_i}{\epsilon^2} + m_i\vec{g}$$
$$v_i = v_i + \frac{dt}{m_i}F$$
$$p_i = p_i + dtv_i$$

We then render the fluid by exporting the position and shape of the fluid after each time step. We include a video of the fluid in the github repo. Just for a last benchmark, to render a video with 100 fluid particles for 100 frames, it took 45s with KD-tree enabled and 70s with it disabled.