École Polytechnique

CSE306 - Geometry Processing - Final Report

Author:

Liam Loughman

Academic year 2023/2024

1 Introduction

In this project, we attempted to develop a computational fluid dynamics simulation to model the behavior of incompressible fluids. We started by implementing the Sutherland-Hodgman polygon and, using this, the Voronoï Parallel Linear Enumeration. Next, we implemented the Power diagram and optimized its weights using the LBFGS algorithm. The final part of our project aimed to implement the de Gallouet-Mérigot scheme for simulating incompressible fluids. This method adjusts the size of each fluid cell and involves interactions with a disk shape, including a spring force tied to each fluid particle. Unfortunately, we were unable to complete this part of the project, it was too hard and we ran out of time.

We ran our ray tracer on a lab machine through SSH. This machine has an Intel® Xeon® W-1270P CPU with 8 cores at 3.80GHz. All rendered images are saved in .svg format. The domain is defined within a [0, 1].

2 Sutherland-Hodgman Polygon Clipping Algorithm

The Sutherland-Hodgman polygon clipping algorithm is used in our project to determine the intersection of a subject polygon with another polygon. The algorithm processes each edge of the clipping polygon sequentially and computes the intersection with the subject polygon's edges.

The function $intersect$ computes the intersection between a line segment from the subject polygon and an edge of the clipping polygon. It uses a vector normal to the clipping edge to determine the parameter t which, if between 0 and 1, indicates that the intersection point lies on the segment. If t is within this range, the function returns the intersection, computed using linear interpolation.

The $inside$ function checks if a point is inside the clipping polygon using the dot product between the vector normal to the edge and the vector from the edge's start point to the point in question.

The $clipPolygon$ function iteratively constructs a new polygon. For each edge of the clipping polygon, it builds the output polygon by traversing the vertices of the subject polygon. It adds vertices to the output polygon if they lie inside the clipping area. If an edge of the subject polygon crosses the boundary of the clipping polygon, the intersection point is calculated and added to ensure the new polygon precisely matches the clipped area.

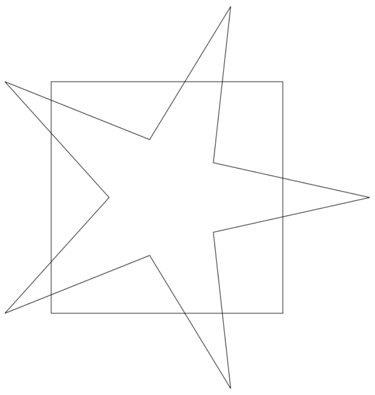
Figure 1: Before Clipping.

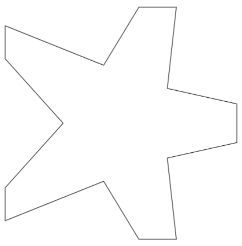
Figure 2: After clipping.

3 Voronoï Parallel Linear Enumeration

The *voronoi* function handles the construction of the Voronoï diagram. Each point in the dataset is compared against all others to determine the polygon (Voronoï cell) that bounds its region by continuously clipping the polygon using the Sutherland-Hodgman clipping algorithm.

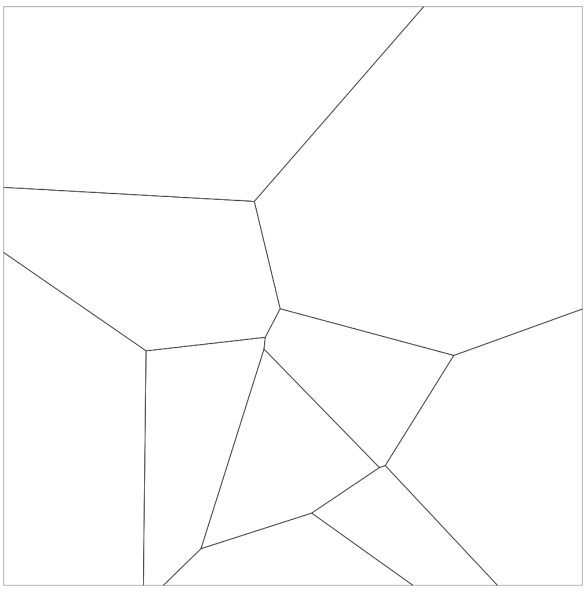
Figure 3: Voronoi Diagram for 10 points. Time for rendering: 0.033 seconds.

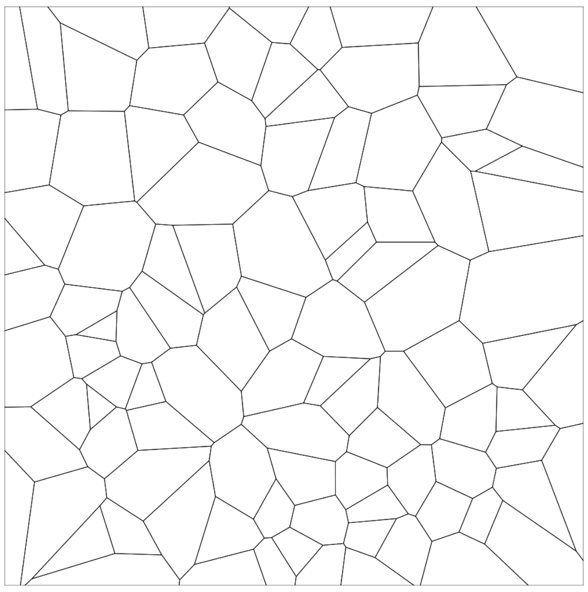
Figure 4: Voronoi Diagram for 100 points. Time for rendering: 0.042 seconds.

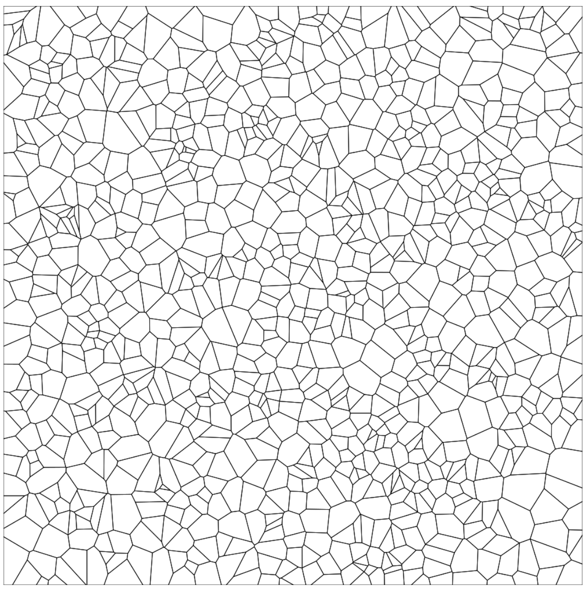
Figure 5: Voronoi Diagram for 1000 points. Time for rendering: 0.091 seconds.

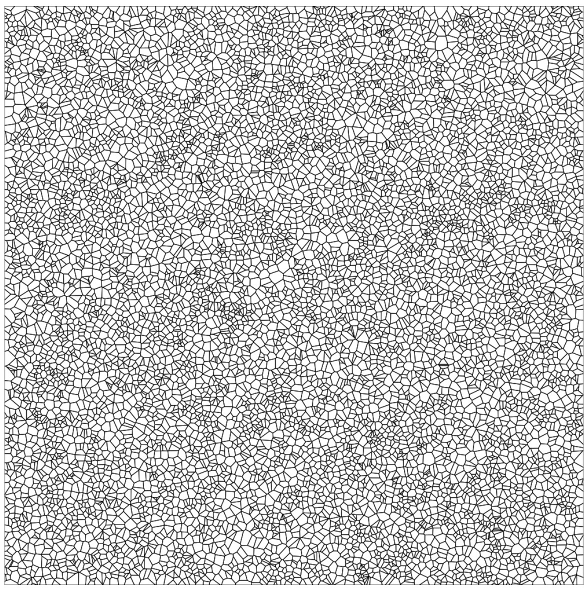
Figure 6: Voronoi Diagram for 10000 points. Time for rendering: 5.235 seconds.

4 Power Diagram

The Power Diagram, which is an extension of the Voronoï diagram, is implemented to handle weighted points. This method is crucial when each point's influence is not uniformly distributed but depends on some weight.

The $power\_diagram\_intersect$ and $power\_diagram\_inside$ functions follow the same structure as the $voronoi\_intersect$ and $voronoi\_inside$ functions. However, it adds an additional step to factor in the weights of the points.

The $power\_diagram$ function, for each pair of points, clips the current polygon (initially the entire bounding polygon) against the boundary defined by these points' weight-adjusted midline. This process is repeated for each point against all others, updating the polygon as necessary by adding new vertices or clipping edges to match the power diagram boundaries.

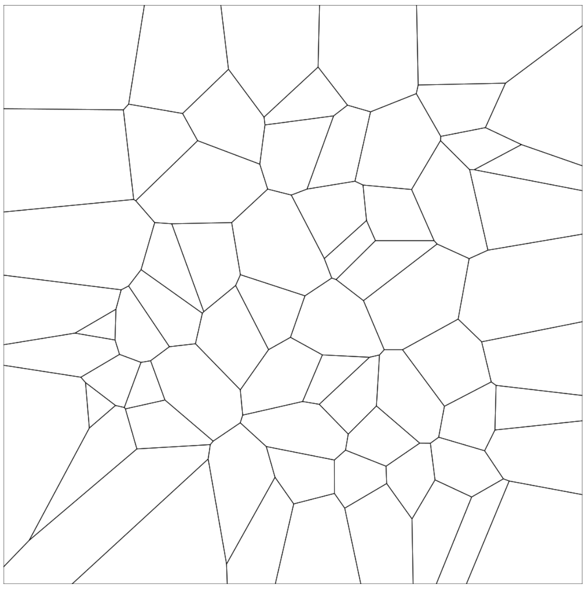
Figure 7: Power diagram with 100 points. Time for rendering: 0.023 seconds.

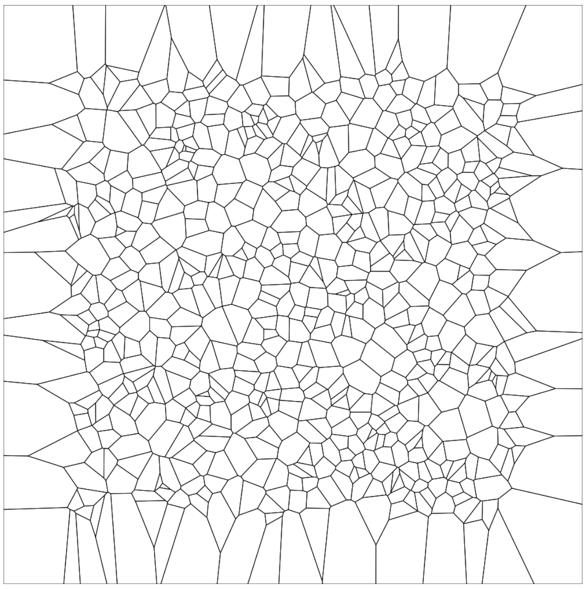
Figure 8: Power diagram with 1000 points. Time for rendering: 0.083 seconds.

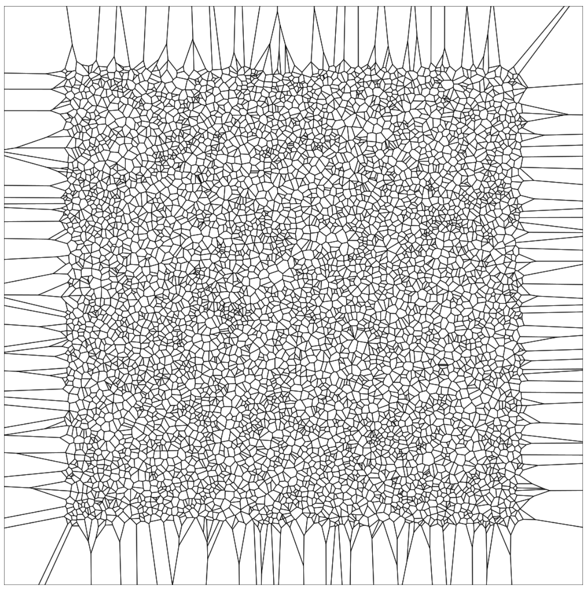
Figure 9: Power diagram with 10000 points. Time for rendering: 4.031 seconds.

For these renders, the weights for the points are set based on their positions. Points that are close to the edges of the space (with x or y coordinates either less than 0.2 or greater than 0.8) are given a weight of 0.7. Points that are centrally located are given a weight of 1.

5 Semi-Discrete Optimal Transport with LBFGS Optimization

We used the LBFGS library to optimize the weights in the Power Diagram. We followed the sample code provided with the library and implemented the $evaluate$ function based on the lecture notes.

The $evaluate$ function takes the current set of weights and computes both the objective function value (fx) and the gradients (g) which are necessary for the LBFGS algorithm. Specifically, for each point in our dataset, the function computes the Power Diagram which partitions the space based on the point locations and their respective weights using the current weights, sums the integrated squared distance between the cell vertices and the point, and computes how much changing that weight would change the area of the corresponding cell in the Power Diagram. This gradient is used by the LBFGS algorithm to adjust the weights in the search for an optimum.

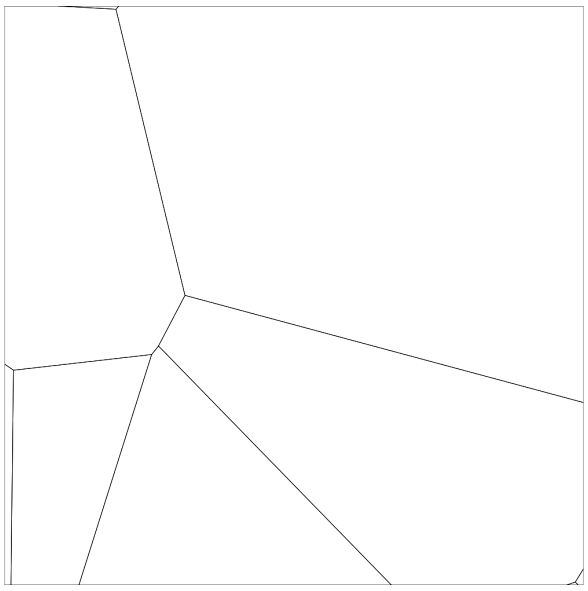
Figure 10: Optimal transport diagram with 10 points. Time for rendering: 1.523 seconds.

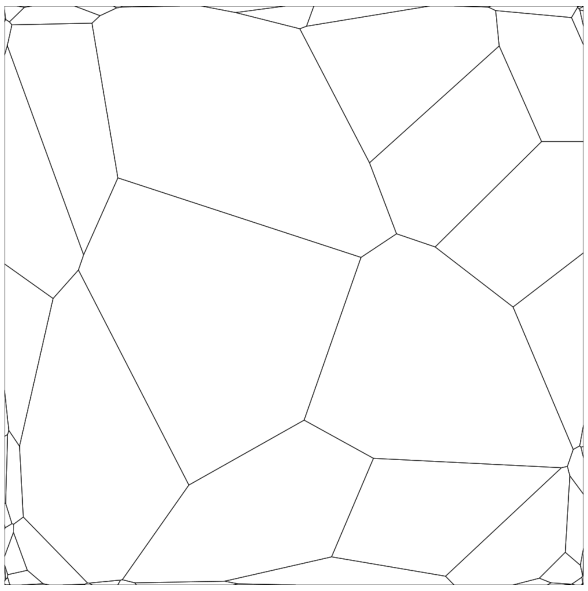
Figure 11: Optimal transport diagram with 100 points. Time for rendering: 13.531 seconds.

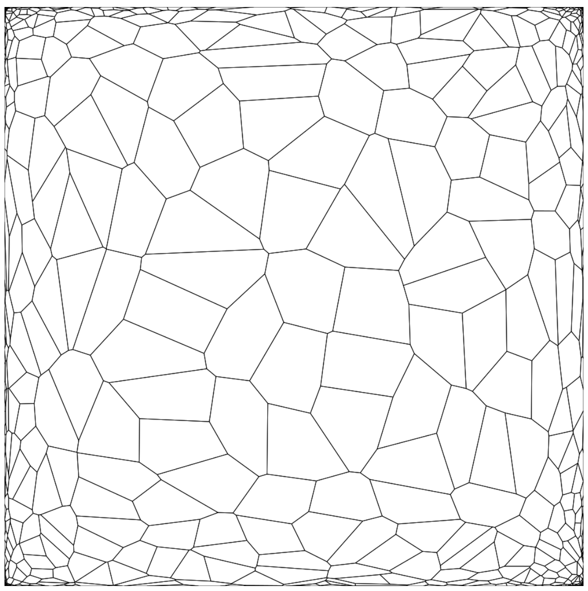

Figure 12: Optimal transport diagram with 1000 points. Time for rendering: 2 minute(s) and 31.507 seconds.

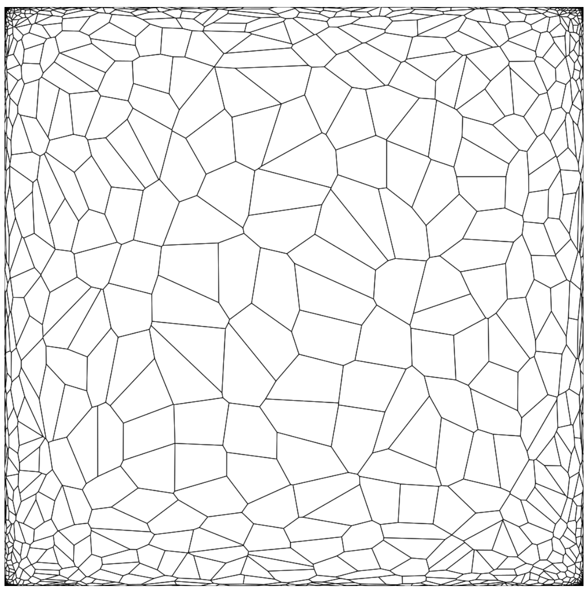

Figure 13: Optimal transport diagram with 2000 points. Time for rendering: 10 minute(s) and 45.606 seconds.