Jules Lahmi

Computer Graphics

Project 2

1. Introduction

This project implements multiple algorithms in computational geometry and fluid simulation using Voronoi diagrams. We start with the classic Sutherland-Hodgman polygon clipping algorithm, and then extend it to construct Voronoi diagrams using the Voronoï Parallel Linear Enumeration algorithm. We then introduce weighted Voronoi diagrams by incorporating power distances. We then add semi-discrete optimal transport and the Gallouet-Merigot algorithm for geometric optimization. Finally, an attempt is made to simulate fluid mechanics using these geometric tools, though this part remains incomplete.

To test each part of the project, uncomment the associated part of the main function in main.cpp.

2. Sutherland-Hodgman Algorithm

We first try to implement the Sutherland-Hodgman algorithm, clipping a subject polygon with a convex auxiliary polygon. We implement the algorithm as it is described in section 4.2 of the lecture notes, taking as input the polygon and the convex clipPolygon, with ordered vertices. To check for the intersection, we create an auxiliary Intersect function that takes as parameters two lines (as 4 points), and returns their intersection using cross product. This returns the point of intersection, even if it is outside of the lines (further than the points taken as input), which will be very useful for our Voronoi diagram. To check if a vertex is inside an edge, we again use a cross product and compare with 0. This produces very good results and works well, as we can see in Figure 1, and will be able to be adapted easily into a Voronoi Diagram.

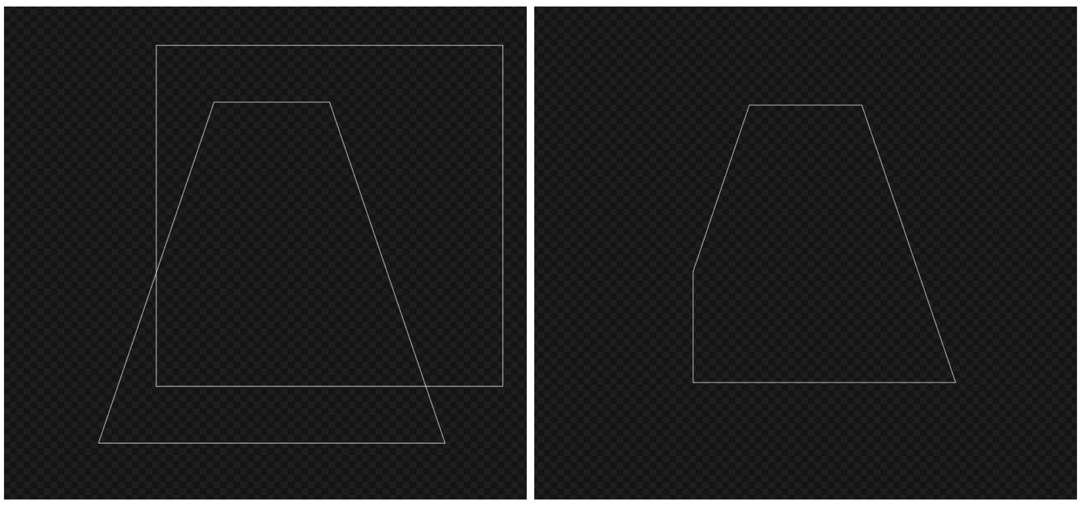

Figure 1 : Sutherland-Hodgman Clipping Algorithm

3. VPLE (Voronoï Parallel Linear Enumeration)

VPLE adapts the idea of Sutherland-Hodgman to compute a Voronoï cell of a point in an array by iteratively clipping a bounding polygon against bisectors formed between the point itself and all other points. Thus instead of using a fixed convex clipping region (starting as the [0,1] box), the clip edge at each step is a bisector between Pi and Pj. We then simply keep the part closer to Pi than Pj. The Voronoi cell for a given site is then obtained as the intersection of all these half-planes.

To modify the *SutherLand* function to work with an edge instead of a polygon, we simply input the large polygon (the box) and the vectors Pi and Pj. The function then computes the bisector as the middle of the two points + the normal, and we use two points on this bisector for the edge. We can then simply use these two points as the edge of the convex polygon.

Since the function *Sutherland_ VPLE* computes this for a single point in an array and does not modify the array, we have no concurrency issues and can simply compute the polygon for each point in parallel using the *VPLE* function, iterating over the array of points.

But as seen in the Lecture Notes, most points do not affect the Voronoi cell, so we first sort all points by euclidean distance to the point we are iterating on, and if the distance from Pi to Pj is more than twice the distance from Pi to the furthest edge in the current (temporary) Voronoi cell, we can discard the call to *Sutherland_ VPLE* of Pj. This is a bit expensive since we are sorting an array of size n for each point in the array, but it still improves the runtimes from 0.7 to 0.1 seconds for 1000 bodies. This is the case because the sorting of the list becomes our main source of computation. I did not manage to use the nanoflamm library to implement this, but what I did improved runtime nonetheless.

We can see the results with 1000 bodies in Figure 2.

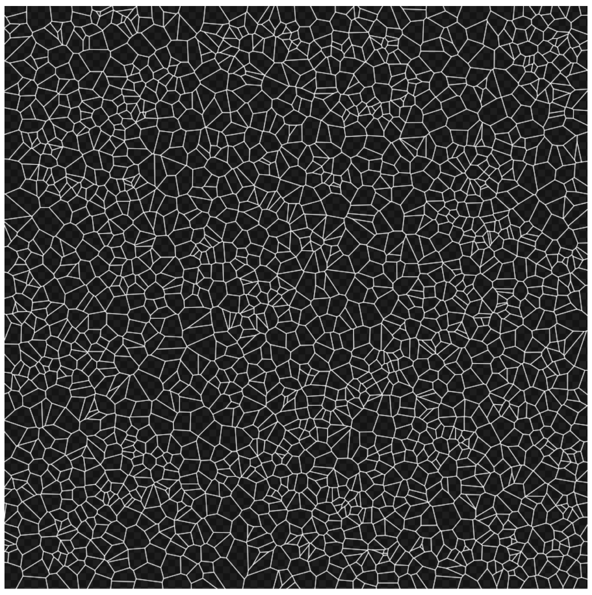

Figure 2 : Voronoi Algorithm with 100 bodies

4. Weighted VPLE

To add weights to our VPLE, we assign a weight wi to each site Pi, which modifies the notion of distance. Instead of using Euclidean distance squared, we use the power distance as described in the lecture notes. Then the bisector is no longer equidistant from the two points but depends on the weights of the two points.

The first change we make to *SutherLand_ VPLE* to include weights is to add a weight parameter, the same size as the array of points. Inside the function for a single point, we simply change the value of the median to the modified formula using Power distances as described in the lecture notes.
Then regarding the *weighted_ VPLE* function, instead of disregarding all points too far by euclidean distance, we disregard all points too far by power distance.
Weighted VPLE adapts Sutherland-Hodgman to use these new bisectors and ensures cells are computed with respect to the power distance. We can also parallelize the same way as the *VPLE* function.

We can see the results of this weighted VPLE using gaussian weights centered around (.5, .5) in figure 3. This is a bit different from the image in the lecture notes, but still looks satisfying. The issue is that cells are being stretched towards the center instead of being stacked towards the exterior so center ones can take more space. This may be an issue with my Gaussian.

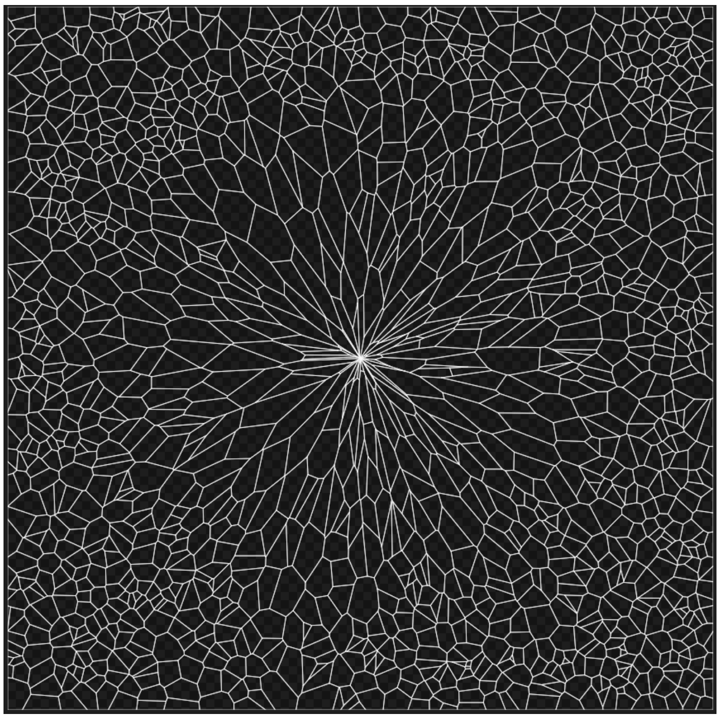

Figure 3 : Gaussian Weights Power Diagram

5. Semi-discrete Optimal Transport

Our goal is now to construct the power diagram according to the weights given. and return the gradient of g and its opposite using the *evaluate* function and the lbfgs library. We give this function an *EvaluateData* struct containing the points, original box and lambdas (population density that we set constant). To do the calculations, we create auxiliary functions to compute the integral of a polygon relative to a point, and the area of a polygon.
Although my auxiliary functions and my maths within the *evaluate* function itself look correct and corresponding to what was in the lecture notes, I still struggled with the lbfgs library, and could hardly debug my code.
I feel like this evaluate function should be working, but when inputting the results in a .txt file, they are not satisfying. I will be testing with the Fluid Mechanics implemented later.

6. Gallouet-Merigot Algorithm

I will now implement the Gallouet-Merigot step algorithm. To do this, we first call *evaluate* using the lbfgs library. We then iterate over each point in the array, take its Voronoi cell (in power distance) and compute its centroid. This was a bit complicated, and still relies of the fact that the vertices of the convex Voronoi polygon are 'ordered'
Finally, we can complete the algorithm using the string force as is explained in the lecture notes. This algorithm uses our previous *evaluate* function, so I find it again hard to test, but when simulating multiple steps of this algorithm and creating a .svg file before and after we see some clear differences. The issue is that I have trouble seeing if these changes are correct or are somehow random.

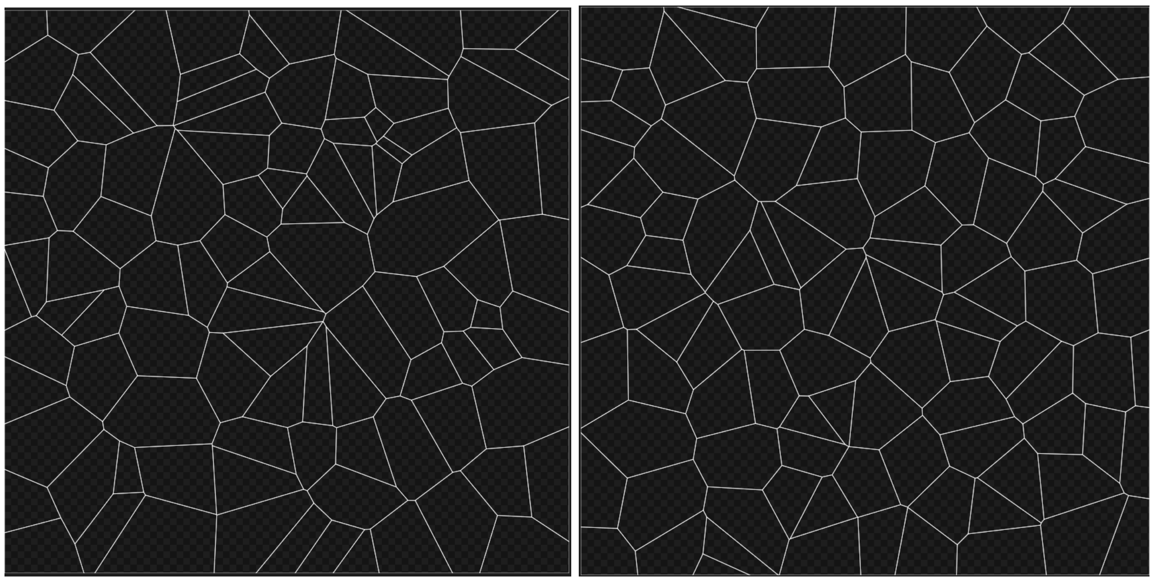

Figure 4 : 100 bodies Voronoi Diagram Before and after 100 steps of Gallouet-Merigo

7. Fluid Mechanics

Using the Gallouet-Merigot algorithm at each step, the goal is now to compute force mechanics.
Unfortunately, I created a loop function to compute fluids, but did not manage to make it work properly..
The non-functional version of my fluids code is in the fluids.cpp file.
It contains all the functions I tried to implement. I suppose some of them, or at least parts of them are correct.
It includes :

- A version of Gallouet-Merigot that also takes air as a parameter
- A version of the *weighted_ VPLE* that attempts to include fluids and air
- A version of *evaluate* that attempts to include fluids and air
- A complete *run_fluid_simulation*

As I was struggling a lot, parts of these incomplete algorithms were found in online resources.

8. Course Feedback

Overall, I really enjoyed the course, especially the first project. I would say that to improve it, maybe creating small weekly quizzes (maybe ungraded even) would be good to make sure students are keeping up with the material, because when it was time to work on the projects, I was sometimes a bit lost, especially for the last lab of each project (labs 4 & 8).
The lectures/td were well planned, but maybe with not enough guidance I would think (although maybe it was on purpose to make us more autonomous).