École Polytechnique, Bachelor of Science, CSC3S006EP (Computer Graphics)

Project Report: Free Surface 2D Fluid Solver

Luca-Andrei Fechete[a,1]

[a] *École Polytechnique*

Abstract—This report details the C++ implementation, logic and results of a free surface 2D fluid solver, completed as the second project for the Computer Graphics course at École Polytechnique, Paris.

1. Introduction

The features implemented as part of this project are the following:

- Voronoï diagram using Voronoï Parallel Linear Enumeration (Sec. 4.3.3, lab 6) with Sutherland-Hodgman polygon clipping algorithm (Sec. 4.2, lab 6).
- Extending this to a Power diagram (Sec. 4.4.3, lab 7)
- Optimizing the weights of the power diagram using LBFGS (Sec 4.4.4, lab 7)
- Gallouet-Mérigot incompressible Euler scheme that prescribes each fluid cell area and intersects fluid cells with a disk, and add a spring force from each fluid particle to their Laguerre's cell centroid. (Sec. 5.4, lab 8).

The following sections will explore more in depth the project's general setup along with each feature, its implementation, and the obtained results.

2. General Structure

This project's implementation can be found in the source directory which includes the source files and the headers of the classes used. The linking and the external package management was done using CMake with a template structure obtained from GitHub [1].

- `Vector`: This class was used for the computations involving two-dimensional vectors. It is defined using two or three coordinates and includes methods related to basic vector operations.
- `OptimalTransport`: This directory incorporates the optimal transport functionalities implemented with the help of the `liblbfgs` library.
- `ParticleSimulation`: This class was used to define the functionality behind the free surface fluid simulator and includes the Gallouet-Mérigot step and some utilities used for convenience.

3. Sutherland-Hodgman Polygon Clipping

In the project the polygon clipping algorithm was implemented based on the pseudo-code from the lecture notes. The intersection point P between the segment $[A, B]$ and the (infinite) line through points u and v was found in the following manner. In particular, $P = A + t\,(B-A), t = \frac{\langle u-A, N\rangle}{\langle B-A, N\rangle}$, where $N = (v_y - u_y,\ u_x - v_x)$ is a normal vector to the line (u, v). Note that if $t < 0$ or $t > 1$, the infinite line does not intersect the finite segment $[A, B]$; in our algorithm this case cannot occur because we only compute intersections when the endpoints lie on opposite sides of the half-space. The `inside` test determines on which side of the clip edge a point P lies. It returns true exactly when $\langle P - u,\ N\rangle \le 0$, where u is any point on the clip edge. The final algorithm can be seen at 1. The results of this algorithm can be been in Figure 1.

4. Voronoi Parallel Linear Enumeration

The Voronoi PLE algorithm can be found at 2. In the computation of the Voronoi-edge intersection, we are given a segment with endpoints A and B and two sites P_i and P_j. We first compute the midpoint $m = \frac{P_i+P_j}{2}$ and the direction vector $d = P_i - P_j$. We seek the parameter t such that the point $X = A + t\,(B - A)$ lies on the perpendicular

Algorithm 1 Polygon Clipping (Sutherland–Hodgman)

```
function PolygonClipping(subjectPolygon, clipPolygon)
    clippedPolygon ← subjectPolygon
    for i ← 0 to |clipPolygon.vertices| − 1 do
        outPolygon ← empty polygon
        u ← clipPolygon.vertices[i]
        v ← clipPolygon.vertices[(i + 1) mod |clipPolygon.vertices|]
        for j ← 0 to |clippedPolygon.vertices| − 1 do
            cur ← clippedPolygon.vertices[j]
            prev ← clippedPolygon.vertices[(j − 1) mod |clippedPolygon.vertices|]
            intersection ← Intersect(prev, cur, u, v)
            if Inside(cur, u, v) then
                if not Inside(prev, u, v) then
                    outPolygon.vertices.append(intersection)
                end if
                outPolygon.vertices.append(cur)
            else if Inside(prev, u, v) then
                outPolygon.vertices.append(intersection)
            end if
        end for
        clippedPolygon ← outPolygon
    end for
    return clippedPolygon
end function=0
```

bisector of P_iP_j, i.e. satisfies $(X - m)\cdot d = 0$. Substituting X gives $((A + t(B - A)) - m)\cdot d = 0$, hence $t = \frac{(m-A)\cdot d}{(B-A)\cdot d}$. If $0 \le t \le 1$, then $X = A + t(B - A)$ is the intersection point on the segment; otherwise no valid intersection exists (and the routine returns A).

The predicate `voronoi_inside` tests whether a point P lies in the half-space of the bisector containing P_i by evaluating the sign of $(P - m)\cdot(P_j - P_i)$. It returns true exactly when $(P - m)\cdot(P_j - P_i) < 0$. The results of this algorithm can be seen in Figure 2a.

5. Weighted Voronoi Parallel Linear Enumeration

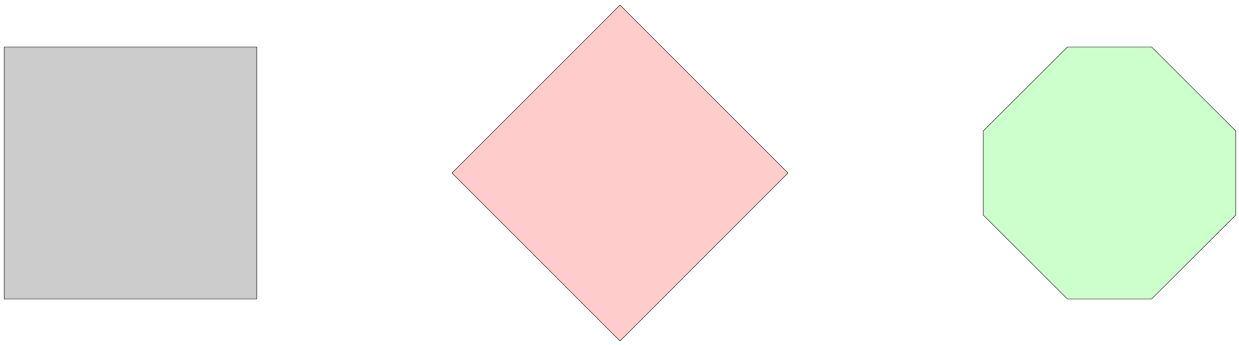
The weighted version of the previous algorithm used an extra array of weights and the substitution of the middle point between two points P_i, P_j by $M' = \frac{P_i+P_j}{2} + \frac{w_i - w_j}{2\|P_i - P_j\|^2}(P_j - P_i)$. The results of this algorithm using linear weights (linear in term of the index) can be seen in Figure 2b.

6. Semi-Discrete Optimal Transport

This algorithm was implemented with the help of the `liblbfgs` library. The implementation was focused around the `evaluate` function which computes the objective function. The goal was to maximize this objective function given by:

$$g(W) = \sum_i \int_{\mathrm{Pow}_W(y_i)} \left(\|x - y_i\|^2 - w_i\right) f(x)\,dx + \sum_i \lambda_i w_i.$$

In practice, the function f was constant and the computation of the integral was done using triangulation and the fact that by fixing one vertex and then iterating over every consecutive ones from the rest of vertices, we can compute the integral over the polygon as a sum of integrals over the resulted triangles with the integral over a triangle

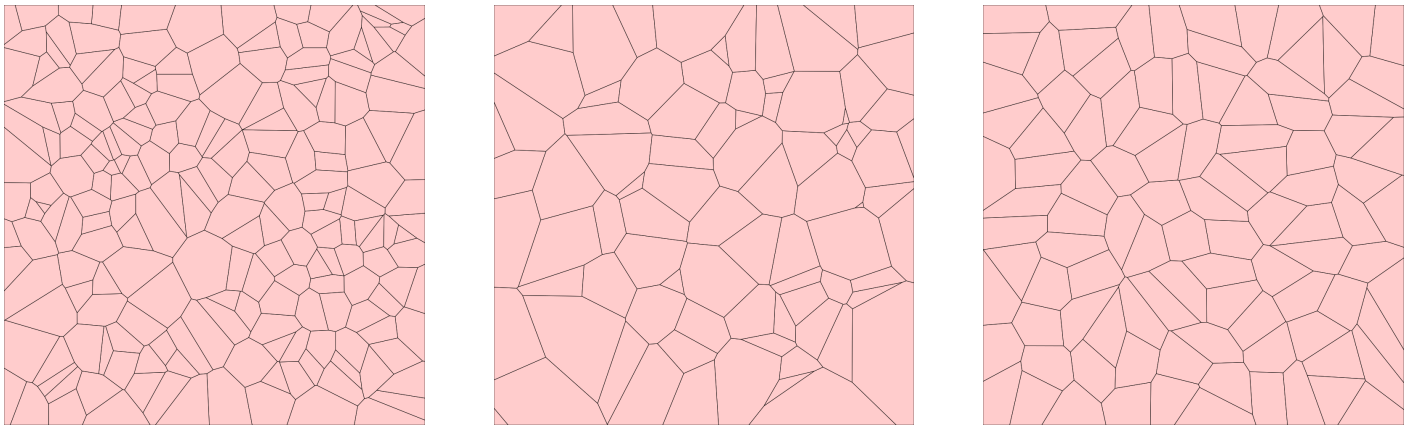


(a) Subject Polygon: the original polygon to be clipped.

(b) Clip Polygon: the convex region defining each clipping edge.

(c) Clipped Result: the intersection after applying the algorithm.

Figure 1. Visualization of the polygon-clipping process. (**a**) Subject polygon, (**b**) clipping polygon, (**c**) resulting clipped polygon computed by the Sutherland–Hodgman algorithm.



(a) Unweighted Voronoi diagram generated by the PL algorithm.

(b) Weighted power diagram with the default weights from `voronoi_pd`.

(c) Final power diagram after refining weights via L-BFGS optimization.

Figure 2. Comparison of diagramming methods. (**a**) unweighted Voronoi cells from the PL algorithm, (**b**) initial weighted power diagram produced by the weighted version of the Voronoi Parallel Linear Enumeration, and (**c**) optimized power diagram after weight adjustment using L-BFGS.

being given by:

$$\int_T \|P - P_i\|^2 \, dP \;=\; \frac{|T|}{6} \sum_{1 \le k \le l \le 3} \langle c_k - P_i,\; c_l - P_i \rangle$$

The final result can be seen in Figure 2c. The optimization takes less than a second for 100 points and around $15 - 20$ seconds for 1000 points.

7. Free Surface 2D Fluid Solver

For simulating fluid I made use of the Gallouet-Mérigot step and of the semi-discrete optimal transport. The implementation follows exactly the pseudo-code from the lecture notes. The results can be seen in the repository of the project at `images/svg_files_final/my_fluid_animation_2500.svg` and `images/svg_files_final/my_fluid_animation_5000.svg`. The first one contains 2500 air particles and 700 fluid particles and it took around 15 minutes to finish. The second one contains 5000 air particles and 1500 fluid particles and it took around 25 minutes to fully render. The number of frames for both is 50 with a dt= 0.03 and an $\epsilon = 0.05$. If you run the code as is, the result of this algorithm would be an svg with 500 air particles and 150 fluid particles that renders in approximately $3 - 4$ minutes.

8. Reproductibility Features

In order to reproduce the majority of the results, there are a number of features in the `main.cc` file. In order to run the first project, `PROJECT` must be set to 1. First of all, in order to reproduce the image with the three spheres, the `SETTING` definition at the beginning of the file should be set to 1. In order to reproduce the image with the spherical light source, the flag `light_source_sphere` should be set to true. Also the optimization strategy for the ray-mesh intersection can be chosen between BVH, BB or no optimization.

In order to run the second project, `PROJECT` must be set to 2. Then a series of functions, that can be found in the `renderer.cc` file, will start running. There are functions that test the Sutherland-Hodgman algorithm, the Voronoi Parallel Linear Enumeration along with its weighted version, the Semi-Discrete Optimal Transport and finally the fluid simulation with the Gallouet-Mérigot step functionality.

All of the features are implemented using the indications and pseudocode from the lecture notes.

Difficulties. My work on the project began with familiarizing myself with the assignment requirements and the fundamental principles of fluid simulation, including polygon clipping, Voronoi cells, fluid simulation techniques, etc. The main difficulties came up when learning to use the `liblbfgs` library and its features as well as when trying to go from the discrete case to the continuous case in the context of fluid simulation, in the end, only implementing the discrete

Algorithm 2 Compute Voronoi cells via parallel linear enumeration (PLE)

```
function VoronoiPLE(sites)
    cells ← empty list
    for i ← 0 to |sites| − 1 do
        P_i ← sites[i]
        {Start with the bounding unit square}
        cell ← Polygon({(0,0),(1,0),(1,1),(0,1)})
        for j ← 0 to |sites| − 1 do
            if j = i then continue
            end if
            P_j ← sites[j]
            outPolygon ← new empty Polygon
            for k ← 0 to |cell.vertices| − 1 do
                cur ← cell.vertices[k]
                prev ← cell.vertices[(k − 1) mod |cell.vertices|]
                X ← VoronoiIntersect(prev, cur, P_i, P_j)
                if VoronoiInside(cur, P_i, P_j) then
                    if not VoronoiInside(prev, P_i, P_j) then
                        outPolygon.vertices.append(X)
                    end if
                    outPolygon.vertices.append(cur)
                else if VoronoiInside(prev, P_i, P_j) then
                    outPolygon.vertices.append(X)
                end if
            end for
            cell ← outPolygon
        end for
        cells.append(cell)
    end for
    return cells
end function=0
```

case.

References

[1] franneck94, *Cppprojecttemplate*. [Online]. Available: https://github.com/franneck94/CppProjectTemplate.