CSE306 - Project 2 Report

Name: Minh Pham

Lecture 6: Geometry Processing - Clipping and Voronoï Diagrams

Key Implementations

1. I started with polygonClip function which receives a subject Polygon and a clipPolygon. The output is a Polygon clipped from the subject Polygon conforming with the shape of a clipPolygon. Figure 1 shows an example test case for this method.
2. Next, I implemented the VoronoiPolygon function, which takes in a set of Points and a specific index of a point and constructs the Voronoi polygon around that point. The starting polygon is the bounding square of the whole surface (0,0) to (1, 1). This is called the subject polygon, which is iteratively updated per iteration. The outer loop loops through every edge between the point *index* with every other point in the pointset, taking the line going through the center of the edge, and clipping the subject polygon by the line. The inner loop checks the intersection between this line with every edge of the subject polygon.
3. The Voronoi diagram is constructed by iteratively performing the Voronoi Polygon function for all points in the point set. OpenMP can be used at this stage as each Voronoi polygon of a point is independent from the other.

Results

The implementation generated Voronoï diagrams for a set of 1000 random points. The Sutherland-Hodgman algorithm clipped the polygons to construct the Voronoï cells. (See Fig 2.)**.** 1000 points Voronoi Diagram takes 0.023 seconds to complete with openMP and 0.15 seconds without openMP.

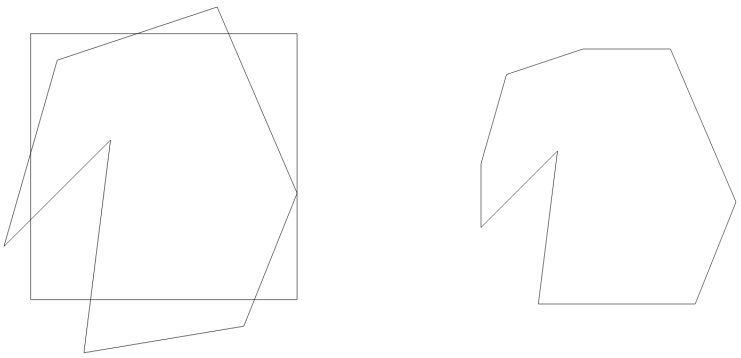


Fig 1. Test Polygon Clip Algorithm. Left: Input. Right: Output.

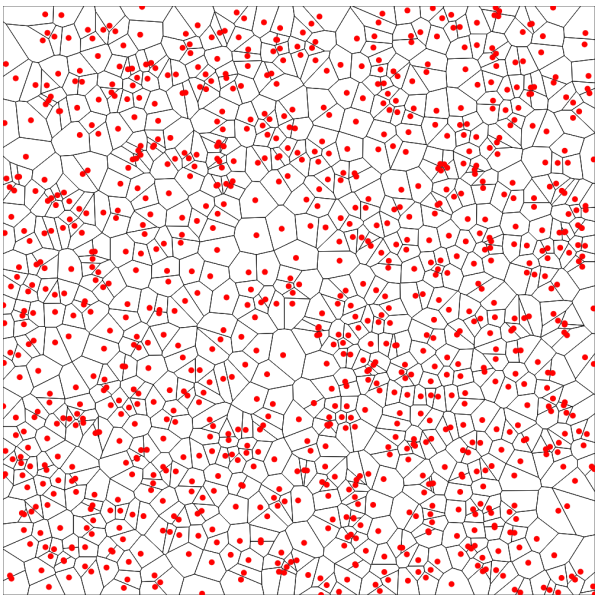
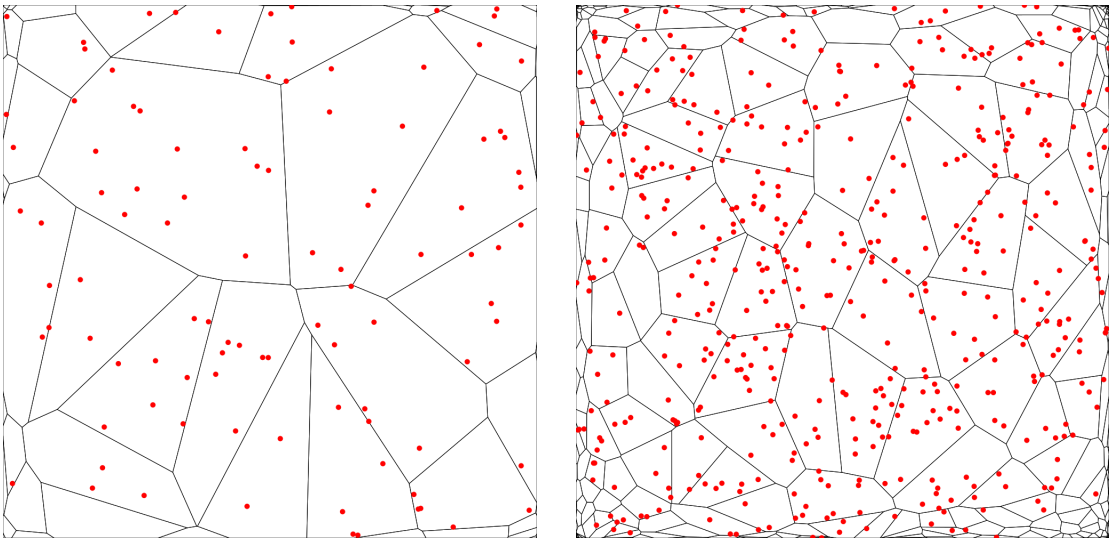


Fig 2. Voronoi Output.

Lecture 7: Geometry Processing - Semi-Discrete Optimal Transport in 2D

Key Implementations

1. Power diagram: Instead of taking the line going through the center of the edge between point *index* with other points, point M is now updated according to additional weights (see formula in lecture notes).
2. The next step is to update the weights to match the requirements (the ratio of the polygon's area). This step requires the integration of the L-BFGS library to optimize an objective function.
3. Before that, I implemented the Polygon *area()* method to calculate the area of the polygon and the *integrate(Vector P)* methods that calculate the integral over the Polygon surface of a given point P inside the Polygon. I used the triangularization integral for this part because the Polygon formula gave me incorrect results. To debug the area functions, I check if the final sum of all polygons is equal to 1.
4. For L-BFGS, I implemented the objective_function class following the cpp sample on their repo. The method run() will call the lbfgs function that will run the optimization routine. The next important function is evaluate(), which needs the value of the objective function (fx) and its derivative (g) with respect to each variable. The computation of fx and g follows the formula in the lecture notes. It is also necessary to negate fx and g as we want to maximize it instead of minimize them.
5. I checked the return code of lbfgs every iteration to make sure they all return 0 correctly.
6. The results are shown in the figure below. The polygons in the center are expanded and the ones on the side shrink.



Power Diagram + L-BFGS optimization
100 points take 0.61seconds to render with OpenMP and O3 compiling flag. 500 points take 19.16 seconds.

Lab 8 Report: Optimizing Voronoi Diagram for Fluid Simulation

Key Implementations

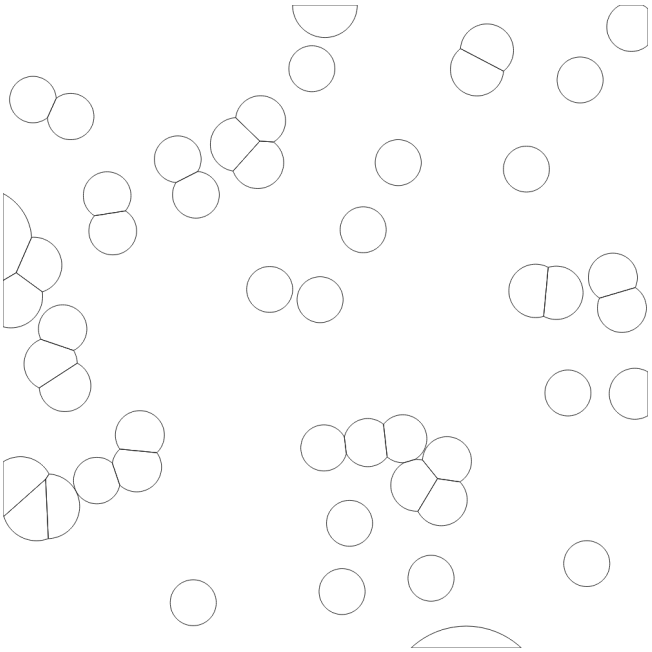
1. Implementing fluid simulation is less complicated because we already have Power Diagram with L-BFGS working.
2. Each polygon now should be clipped with a disk of radius R = *sqrt(weights[i] - weights[air]*). For this, I created a constructDisk method to easily change the number of sample points per disk later on.
3. The evaluation method in the objective_function class should be modified as follows:

```
for (int i = 0; i < n_fluid; i += 1)
{
    // Pow.print()
    g[i] = polygons[i].area() - FLUID / n_fluid;
    // cout << polygons[i].integrate(points[i]) << " " << polygons[i].area() << " " << sum_lambdas << endl;
    fx += polygons[i].integrate(points[i]) - weights[i] * polygons[i].area() + (FLUID / n_fluid) * weights[i];
}
g[n - 1] = (1 - estimated_vfluid) - (1 - FLUID);
fx += weights[n - 1] * ((1 - FLUID) - (1 - estimated_vfluid));
return -fx;
```

4. FLUID is the proportion of fluid volume in the whole surface (between 0 and 1).
5. Finally, I add the physics formula into a GMScheme function that iteratively modify the position of the particle and recompute the diagram.

```
void GMScheme(vector<Vector> &points, vector<Vector> &v, double mass, int max_num_frames = 50)
{
    for (int i = 0; i < max_num_frames; i++)
    {
        objective_function obj(points);
        obj.run(N_POINTS + 1);
        double eps = 0.004;
        double dt = 0.008;
        Vector g = Vector(0, -9.8, 0);

        vector<Polygon> voronoi(points.size());
        double sum_area = 0;
        vector<Vector> new_points(N_POINTS);

        for (int i = 0; i < points.size(); i++)
        {
            voronoi[i] = VoronoiPolygon(points, obj.weights, i);
            double R = sqrt(obj.weights[i] - obj.weights[N_POINTS]);
            voronoi[i] = polygonClip(voronoi[i], constructDisk(points[i], R));

            Vector F_spring = (voronoi[i].centroid() - points[i]) / (eps * eps);
            // PRINT_VEC(F_spring);
            Vector F = F_spring + mass * g;
            v[i] = v[i] + F * dt / mass;
            new_points[i] = points[i] + v[i] * dt;
        }

        points = new_points;

        // printf("Frame %d done!\n", i);

        save_svg_animated(voronoi, "./fluid.svg", i, max_num_frames);
    }
}
```

6. The end result is a smooth, satisfying fluid animated SVG (obviously cannot be seen in the figure below but available on my repo).



Fluid Simulation: 50 Points with p_fluid = 0.2, mass = 200, 200 frames, rendered in 30 seconds.