Cours de programmation séquentielle

Triangulation de Delaunay

1 Buts

— Triangulation d'un nuage de points.
— Implémentation de l'algorithme de Bowyer–Watson.
— Utilisation des structures de vecteurs.

2 Énoncé

La triangulation de Delaunay permet de créer un ensemble de triangle qui recouvre l'envelope convexe d'un nuage de points. Elle a comme propriété de maximiser les angles aigus des triangles faisant partie de la triangulation.

Le but de ce travail est d'implémenter l'algorithme de Bowyer–Watson pour effectuer une triangulation de Delaunay pour une liste de points en deux dimensions, $\mathcal{P}$. Les détails de l'algorithme se trouvent dans les slides sur cyberlearn. Pour implémenter l'algorithme, nous allons discuter brièvement des structures de données dont vous aurez besoin.

2.1 Les `point_2d`

Vous devez définir une structure `point_2d`

```
typedef struct {
    double x, y;
} point_2d;
```

ainsi que toutes les fonctions pour leur manipulation (ces fonctions sont très similaires aux fonctions pour les `point_3d`). En particulier, il faut avoir une fonction permettant de calculer si un point, `p0`, est dans le cercle circonscrit d'un triangle. Si les sommets d'un triangle sont définis par les points, `p1`, `p2`, et `p3`, la fonction déterminant si `p0` est à l'intérieur du cercle circonscrit de centre `pc` et de rayon au carré `rsqr` avec une tolérance `epsilon` est donnée par

```
static bool is_inside_circum_circle_points(point_2d p0,
    point_2d p1, point_2d p2, point_2d p3,
    point_2d *pc, double *rsqr, double epsilon)
{
    double fabs_y1y2 = fabs(p1.y-p2.y);
    double fabs_y2y3 = fabs(p2.y-p3.y);
```

```
    /* Check for coincident points */
    if (fabs_y1y2 < epsilon && fabs_y2y3 < epsilon)
        return(false);

    if (fabs_y1y2 < epsilon) {
        double m2 = - (p3.x - p2.x) / (p3.y-p2.y);
        double mx2 = (p2.x + p3.x) / 2.0;
        double my2 = (p2.y + p3.y) / 2.0;
        pc->x = (p2.x + p1.x) / 2.0;
        pc->y = m2 * (pc->x - mx2) + my2;
    } else if (fabs_y2y3 < epsilon) {
        double m1 = - (p2.x-p1.x) / (p2.y-p1.y);
        double mx1 = (p1.x + p2.x) / 2.0;
        double my1 = (p1.y + p2.y) / 2.0;
        pc->x = (p3.x + p2.x) / 2.0;
        pc->y = m1 * (pc->x - mx1) + my1;
    } else {
        double m1 = - (p2.x-p1.x) / (p2.y-p1.y);
        double m2 = - (p3.x-p2.x) / (p3.y-p2.y);
        double mx1 = (p1.x + p2.x) / 2.0;
        double mx2 = (p2.x + p3.x) / 2.0;
        double my1 = (p1.y + p2.y) / 2.0;
        double my2 = (p2.y + p3.y) / 2.0;
        pc->x = (m1 * mx1 - m2 * mx2 + my2 - my1) / (m1 - m2);
        if (fabs_y1y2 > fabs_y2y3) {
            pc->y = m1 * (pc->x - mx1) + my1;
        } else {
            pc->y = m2 * (pc->x - mx2) + my2;
        }
    }

    double dx = p2.x - pc->x;
    double dy = p2.y - pc->y;
    *rsqr = dx*dx + dy*dy;

    dx = p0.x - pc->x;
    dy = p0.y - pc->y;
    double drsqr = dx*dx + dy*dy;

    return ((drsqr - *rsqr) <= epsilon);
}
```

Les points en deux dimensions seront stockés dans une structure vecteur [1].

2.2 Les `i_triangle`

Afin de représenter les triangles, il ne faut pas utiliser directement les structures `triangle` que vous avez déjà implémentée. Il est en fait plus efficace d'utiliser

1. Au sens de ce que nous avons implémenté au TP sur le sujet il y a quelques semaines.

les indices des points dans le vecteur `vec_points` contenant la liste de tous les points de notre triangulation. Ainsi, `i_triangle` est la structure

```
typedef struct {
    int p1, p2, p3;
} i_triangle;
```

où `p1`, `p2`, et `p3` représentent les indices des sommets du triangle dans du vecteur de `point_2d`[2] que nous appellerons `vec_points`. Ainsi vous n'avez besoin de stocker que 3 entiers, plutôt que de réallouer à chaque fois la mémoire pour chaque sommet de triangle. Chaque triangle sommet du triangle peut être accédé avec

```
// t est un i_triangle
point_2d p1 = vec_points[t.p1];
point_2d p2 = vec_points[t.p2];
point_2d p3 = vec_points[t.p3];
```

2.3 Les `i_edge`

De façon similaire, les arêtes des triangles, dont vous avez besoin dans l'algorithme de Bowyer–Watson, sont également uniquement deux entier, qui représentent l'indice dans le vecteur `vec_points` des sommets de l'arête. Ainsi, la structure `i_edge` est donnée par

```
typedef struct {
    int p1, p2;
} i_edge;
```

et chaque sommet de l'arête se retrouve avec

```
// e est un i_edge
point_2d p1 = vec_points[e.p1];
point_2d p2 = vec_points[e.p2];
```

Les arête de la triangulation devront être stockées dans un vecteur de `i_edge`[3].

2.4 La triangulation

La triangulation n'est rien d'autre qu'un vecteur de `i_triangles`. Elle s'obtiendra avec la fonction

```
void triangulate(vec_point_2d points, vec_i_triangle triangles);
```

2.5 Le super triangle

Le super triangle n'est pas une structure de donnée à proprement parler, mais il requiert un traitement spécial. En effet, pour le créer, il faut rajouter trois points "fictifs" au vecteur des points que vous allez trianguler. Ces points ont les coordonnées suivantes:

2. Au sens de ce que nous avons implémenté au TP sur le sujet il y a quelques semaines.
3. Au sens de ce que nous avons implémenté au TP sur le sujet il y a quelques semaines.

```c
double dx = xmax - xmin; // valeur max de x - min de x des points
double dy = ymax - ymin; // valeur max de y - min de y des points
double dmax = (dx > dy) ? dx : dy;
double xmid = (xmax + xmin) / 2.0;
double ymid = (ymax + ymin) / 2.0;

point_2d p0 = point_2d_create(xmid - 20 * dmax, ymid - dmax);
point_2d p1 = point_2d_create(xmid, ymid + 20 * dmax);
point_2d p2 = point_2d_create(xmid + 20 * dmax, ymid - dmax);
```

où `xmin`, `xmax` sont les valeurs minimale, respectivement maximale, de la coordonnée `x` du vecteur de points. De façon similaire, `ymin`, `ymax` sont les valeurs minimale, respectivement maximale, de la coordonnée `y` des points de la triangulation.

Ces points doivent être retirés de la liste, une fois la triangulation finie. N'oubliez pas d'enlever les triangles contenant les sommets du super-triangle de la triangulation quand celle-ci est terminée.

2.6 Optimisation de l'algorithme

Optimiser un code ne se fait qu'une fois que le code fonctionne. En effet, avoir un code très optimisé mais faux est totalement inutile. On se sert de l'implémentation de référence pour vérifier que le code optimisé donne toujours **exactement** le même résultat que le code de référence.

La complexité de l'algorithme de Bowyer–Watson tel qu'il est implémenté ici est de $\mathcal{O}(N^2)$, où N est le nombre de triangles. En effectuant la recherche des points se trouvant dans les cercle circonscrits, on peut grandement améliorer l'efficacité de l'algorithme (il devient $\mathcal{O}(N \cdot \log N)$).

Néanmoins, on peut effectuer une optimisation à faible coût pour accélérer l'insertion des points en triant le vecteur `vec_points` selon la coordonnée `x`. Vous devrez donc implémenter l'algorithme de tri de votre choix afin de trier les points par ordre croissant selon leur coordonnée `x`.

2.7 Remarque

Si vous n'avez pas réussi à implémenter les vecteurs une autre solution est d'utiliser des tableaux pour toutes les structures de données. Mais cela ne simplifiera pas votre tâche, loin de là.

3 Tests

Afin de tester que votre triangulation marche, il faudra écrire l'écrire dans un fichier `STL`[4]. Il faudra donc,

Commencez par tester votre triangulation sur trois points pris au hasard et vérifiez que vous obtenez bien le triangle. Puis, vous pouvez tester sur la liste de points suivante:

4. Vous avez déjà écrit la librairie qui permet d'écrire des triangles dans un fichier `STL`.

```
point_1 = (79.692045, 177.291743)
point_2 = (105.876045, 15.998743)
point_3 = (-249.549955, 146.060743)
point_4 = (214.277045, 40.505743)
point_5 = (-246.241955, 164.004743)
point_6 = (182.411045, 146.041743)
point_7 = (78.422045, -238.258257)
point_8 = (149.081045, 152.309743)
point_9 = (112.333045, 88.411743)
point_10 = (-116.498955, -186.731257)
```

Le fichier `STL` que vous devriez obtenir si tout se passe bien se trouve sur `cyberlearn` (le fichier ne sera pas forcément identique).