Géométrie algorithmique et maillages
Rapport projet de triangulation Delaunay

Aurélien Chemier 10908892
aurelien.chemier71@gmail.com
Romane Lhomme 11006689
romane.lhomme@gmail.com

5 Janvier 2015

Table des matières

1 Introduction

L'objectif de ce projet, réalisé dans le cadre de l'unité d'enseignement *"Géométrie algorithmique et maillages"* à l'Université Claude Bernard Lyon 1, était de générer la triangulation de Delaunay d'un terrain en $2D\frac{1}{2}$ à partir de points tirés aléatoirement en utilisant le principe du *godness of fit* (ou affinage).
La triangulation est produite à partir d'insertions itératives des points dans le maillage.

Le principe du *godness of fit* consiste à insérer les points selon une priorité dépendant de la distance courante entre le maillage et le terrain. Ainsi, on insère en priorité les points les plus éloignés du maillage.

La triangulation de Delaunay consiste à vérifier, pour chaque triangle créé, que seuls les points présents dans le triangle circonscrit à ce triangle sont ceux qui le définissent. Lorsque l'on constate une anomalie, le maillage est modifié localement pour y remédier.

Afin de mener à bien ce projet, il nous a été demandé d'utiliser le language C pour implémenter les algortihmes et la bibliothèque graphique OpenGL afin d'avoir un aperçu des résultats. Nous avons également utilisé le gestionnaire de version Git afin de travailler facilement en collaboration.

2 Utilisation du programme

Les paramètres par défaut d'utilisation du programme peuvent être modifiés via la ligne de commande.

- L'affichage du résultat dans une fenetre OpenGL doit être demandé avec l'argument $-a$ pour le squelette et $-t$ pour les triangles pleins et colorés.
- Le nombre de vertices est de 50 par défaut. Il peut être modifié avec l'argument $-n$ suivi du nombre de vertices voulu.
  Exemple : $./delaunay - n100$
- Une triangulation naïve a également été implémentée. L'argument $-b$ permet de sélectionner cette triangulation.
- Les différents critères d'arrêts pour la triangulation doivent être sélectionnés en ligne de commande :
  - Le nombre de facettes à créer peut être modifié avec l'argument $-f$ suivi du nombre de facettes voulues. Par défaut toutes les facettes sont créés.
    Exemple : $./delaunay - f50$

  - $-s$ modifie la valeur du seuil de distance minimale pour qu'un vertex puisse être inséré. Par défaut la distance minimale est à 0. Ce seuil doit être exprimé sous forme pourcentage par rapport à la distance maximale et doit être compris dans l'intervale $[0; 1[$.
    Exemple : $./delaunay - s0.2$ insérera les vertices donc la distance à la facette leur correspondant est supérieure à 20% de la distance maximale possible.

- L'aide peut être retrouvée avec l'argument $-h$.

Les différentes options peuvent être cumulées à l'exception de l'aide qui arrête le programme.

3 Structures de base

Afin de pouvoir mettre en place un système de triangulation à partir de vertices, nous avons dû créer un certain nombre de structures de base.
Si vous souhaitez avoir plus d'informations que celles données dans cette section, vous pouvez consulter la documentation Doxygen du projet.

3.1 Vertex

Le Vertex est la structure de base du projet, il permet de représenter les points qui formeront les sommets des simplexes.

Listing 1 – la structure Vertex

```
typedef struct _Vertex
{
    double coords[DIM];
    struct _Vertex *suivant;
} Vertex;
```

- *coords* est le tableau des contenant les coordonnées du vertex. Sa taille correspond au nombre de dimensions dans lequel le vertex est défini ; dans le cadre de ce projet *DIM = 3*.

- *suivant* est un pointeur sur un autre vertex. Il est utilisé pour la gestion des vertices dans la file de priorité de la structure *Delaunay* (voir 3.4).

3.2 Simplexe

La structure Simplexe permet de représenter les triangles.

Listing 2 – la structure Simplexe

```
typedef struct _Simplexe
{
```

```
    const Vertex *sommets[3];
    Vertex *listeVertex;
    Equation e;
    struct _Simplexe *voisins[3];
    double distanceMax;
    int indiceDansFile;
    struct _Simplexe *precedentPile;
    time_t marqueurTemps;
} Simplexe;
```

- *sommets* contient les pointeurs sur les trois vertices qui composent le triangle. Ces trois sommets sont systématiquement ordonnés dans le tableau de manière à être définis dans le sens trigonométrique. Si tel n'était pas le cas, plusieurs fonctions et procédures n'auraient pas les résultats escomptés.

- *listeVertex* est un pointeur sur le premier vertex de la liste des vertices qui appartiennent à ce simplexe mais n'ont pas encore été insérés. Le premier élément de cette liste est le vertex le plus éloigné du plan défini par le simplexe. On peut accéder aux autres éléments de la liste grâce à l'attribut *suivant* de la structure *Vertex*. Si il n'y a plus de vertex à insérer dans le simplexe, la liste pointe sur *NULL*.

- *e* contient les paramètres de l'équation définissant le plan formé par le simplexe. Elle est calculée à la création du simplexe et est de la forme suivante :
$$a * x + b * y + c * z + d = 0$$

- *voisins* contient les pointeurs sur les voisins du simplexe. Ils sont ordonnés de la manière suivante : le voisin en position i correspond au simplexe géographiquement opposé au sommet i ($i \in [0; DIM - 1]$).

- *distanceMax* contient la distance entre le plan formé par le simplexe et le vertex le plus éloigné de ce plan. Si la liste de vertices est vide, la distance est de -1. Cette distance entre un vertex v est le plan est calculée grâce à la formule suivante :
$$|| - \frac{a * v.x + b * v.y + d}{c}| - |v.z||$$
avec
  - a, b, c et d les paramètres de l'équation du plan.
  - $v.x$, $v.y$ et $v.z$ les coordonnés du vertex.

- *indiceDansFile* stocke comme son nom l'indique la position du simplexe dans la file de priorité (voir 3.3).

- Les éléments *precedentPile* et *marqueurTemps* sont utilisés dans le cas de la triangulation de Delaunay. Lorsque de nouveaux simplexes sont créés, ils sont ajoutés à la pile afin de pouvoir vérifier si il faut faire des échanges de diagonales. *precedentPile* pointe sur le simplexe précédant dans la pile et *marqueurTemps* permet d'indiquer si le simplexe a déjà été traité dans l'itération courante.

3.3 File de Priorité

La file de priorité permet de trier les simplexes en fonction de leur attribut *distanceMax*. Plus celle ci est élévée, plus le simplexe est prioritaire. Cette file de priorité est basée sur celle developpée en TP mais a été adaptée pour les simplexes.
Elle se met à jour automatiquement à chaque ajout ou suppression d'élément ; une fonction est également disponible pour la mettre à jour lorsqu'un simplexe a été modifié (après un échange de diagonale par exemple).

Listing 3 – la structure FileSimplexe

```
typedef struct
{
   Simplexe ** file;
   int nbElements;
   int nbElementsCourant;
} FileSimplexe;
```

- *file* est un tableau de pointeurs sur des simplexes. Ce tableau est de taille $nbElements + 1$ car le premier élément d'une file de priorité se trouve à l'indice 1.
- *nbElementsCourant* correspond au nombre de simplexes dans la file et donc au nombre de facettes créées.

3.4 Delaunay

Cette structure a été créée afin de regrouper les éléments nécessaires à la triangulation et de simplifier le passage des paramètres des fonctions et procédures.

Listing 4 – la structure Delaunay

```
typedef struct
{
    FileSimplexe *filePrioriteSimplexe;
    Vertex *tableauVertex;
    int nbVertex;
    int nombreFacetteMax;
    double distanceMin;
} Delaunay;
```

- *filePrioriteSimplexe* est un pointeur sur la file de priorité qui contient tous les simplexes.
- *tableauVertex* est un tableau qui contient tous les vertices qui sont créés lors de l'initialisation puis qui seront associés à des simplexes. Ce tableau est de taille *nbVertex*.
- *nombreFacetteMax* et *distanceMin* sont les variables qui permettent de gérer les cas d'arrêts de la triangulation (voir 5.1).

4 Initialisation

Avant de commencer la triangulation, une étape d'initialisation est nécessaire afin de tirer aléatoirement les coordonnées des vertices et de créer manuellement les deux premiers simplexes. Cette étape permet également d'initialiser la structure *Delaunay*.

Le tableau de vertices est alloué en mémoire de manière à pouvoir contenir autant de vertices que le paramètre renseigné par l'utilisateur en ligne de commande. Les quatres premiers vertices correspondent au quatres sommets du carré unitaire initial ($(0,0)$, $(1,0)$, $(1,1)$ et $(0,1)$). Les coordonnées en x, y et z de tous les autres vertices sont définis selon une loi aléatoire uniforme

$$x, y \in [0; 1[; z \in [0; H\_MAX[$$

Il faut ensuite, à son tour, allouer la file de priorité. Par définition, le nombre maximum de simplexes pouvant être créés est égal à $2 * (n-1) - x$ avec :

- n le nombre de vertices.
- x le nombre de vertices composant l'enveloppe convexe.

Dans notre cas $x = 4$ ce qui nous donne au plus $2n - 6$ simplexes. Comme nous ne gérons pas de conteneur dynamique, on alloue autant de place que nécessaire pour stocker tous les simplexes dans la file de priorité.

Les quatres premiers vertices forment les deux premiers simplexes qui seront insérés dans la file : $((0,0),(1,0),(1,1))$ et $((0,0),(1,1),(0,1))$. Les voisins des simplexes sont initialisés à *NULL* lorsque qu'il s'agit des bords du carré unitaire.
Les vertices restants sont répartis dans les deux simplexes. Un vertex est à l'interieur d'un simplexe si sont projeté sur le plan $z = 0$ est à l'intérieur du projeté du simplexe sur ce même plan.

Enfin, il faut initialiser les variables qui gérent les cas d'arrêt.
Si aucune limite du nombre de facettes n'est passée en paramètre, la variable *nombreFacetteMax* est alors égale à la taille maximum de la file de priorité. Sinon, c'est la valeur passée en paramètre qui sera stockée dans cette variable.
La variable *distanceMin* est initialisée à 0 par défaut ; cela signifie que tous les vertices seront insérés car il n'y a pas de distance minimale à respecter. Si une valeur est passée en paramètre, elle est multipliée par *H_MAX* avant d'être stockée dans la structure.

À la fin de l'initialisation, tout est prêt pour que la triangulation se fasse de manière automatique.

5 Triangulation

Pour la triangulation, nous avons procédé en deux étapes. Dans un premier temps, une triangulation naïve a été implémentée. La gestion des triangles avec l'algorithme de Delaunay n'a été mise en place que lorsque la première triangulation fonctionnait.

5.1 Cas d'arrêt

La triangulation peut s'arrêter en fonction de deux paramètres qui sont passés par l'utilisateur. Ces deux paramètres ne sont pas mutuellement exclusifs. Si aucun paramètre n'est fourni, la triangulation s'arrête lorsque tous les vertices on été insérés.
Voici les deux cas d'arrêt :

- On s'arrête si l'on atteint le nombre maximum de facettes demandé par l'utilisateur.
  Etant donné qu'à chaque itération on crée trois simplexes et qu'on en supprime un, il est possible que le nombre final de facettes soit légèrement supérieur à celui renseigné par l'utilisateur.
- On s'arrête lorsque la valeur de l'attribut *distanceMax* du simplexe en tête de la file de priorité est inférieure à la distance minimale demandée par l'utilisateur. Si cette distance est égale à 0, tous les vertices seront insérés.

Dans tous les cas, l'algorithme insère les vertices dans la triangulation tant qu'aucun des cas d'arrêt n'est satisfait.

5.2 Triangulation de base

5.2.1 Algorithme

Données $fileS$ *la file de priorité des simplexes contenant les deux simplexes initiaux*
Résultats $fileS$ *la file contenant tous les simplexes créés*
while *(fileS.premier.distanceMax > distanceMin) && (fileS.nbElementsCourant > nbFacetteMax)* **do**
    simplexe = premierFile;
    vertex = simplexe.vertexLePlusLoin;
    listeVertexTemp = vertex.suivant;
    s1, s2, s3 = nouveaux simplexes à partir de $simplexe$ et $vertex$
    foreach s *in {s1, s2, s3}* **do**
        ajoutDesVoisins(s);
    end
    foreach $voisin$ *in simplexe.voisins* **do**
        changementDeVoisin(voisin, simplexe, nouveauVoisin);
    end
    foreach v *in listeVertexTemp* **do**
        repartitionDansLesNouveauxSimplexes(v);
    end
    ajoutDansLaFile(s1);
    ajoutDansLaFile(s2);
    ajoutDansLaFille(s3);
    suppression(simplexe);
end

Algorithme 1 : Algorithme de triangulation naïve

Lors de la création de chaque simplexe dans cet algorithme, les sommets sont rangés dans l'ordre trigonométrique pour garantir le bon fonctionnement du programme et une certaine simplicité de raisonnement.

5.2.2 Efficacité

Voici, pour un nombre de vertices données, le temps, en secondes, d'exécution de la triangulation (l'affichage n'est pas pris en compte). Les tests ont été effectués sur l'une de nos machines personnelles et peuvent varier d'un ordinateur à un autre.

Nombre de vertices	Temps d'exécution (s)
10	0,004027
100	0,005394
1000	0,007469
10^4	0,03256
10^5	0,306865
10^6	4,173267
10^7	58,279697

L'augmentation de la durée reste globalement linéaire ($o(N)$) ; seule la gestion de la file de priorité s'effectue en $o(\log n)$.

5.3 Triangulation de Delaunay

5.3.1 Algorithme

La triangulation de Delaunay consiste à ajouter à l'algorithme naïf (voir 5.2.1) une boucle qui va contrôler, pour chaque simplexe, qu'il n'y a pas de vertices autres que les sommets de ce simplexe dans son cercle circonscrit.
Après l'ajout des nouveaux simplexes créés dans la file, on les insère également dans la pile qui sera gérée par la boucle suivante.

```
Données p Une pile de Simplexe
while p non vide do
    simplexe = sommetPile(p);
    foreach voisin de simplexe do
        SommetOppose = sommet de voisin qui n'est pas dans simplexe;
        if SommetOppose appartient à Cercle de Simplexe then
            echangeSimplexe(simplexe, voisin);
            retriFile(Simplexe);
            retriFile(voisin);
            ajoutPile(simplexe);
            ajoutPile(voisin);
        end
    end
end
```

Algorithme 2 : Boucle supplémentaire pour la triangulation de Delaunay

5.3.2 Echange de diagonale

La fonction *echangeSimplexe* qui peut aussi être appellée *swap* consiste à changer la diagonale commune de deux simplexes afin que la condition de Delaunay soit respectée. Ainsi deux simplexes ABC et et ACD deviendront ABD et BCD (voir figure 1).

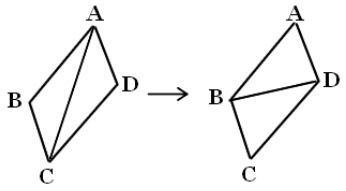

FIGURE 1 – Echange de diagonale

Nous avons essayé de faire un échange de diagonale le plus générique possible afin d'avoir une méthode claire et précise. Par la suite, nous supposerons que les deux simplexes dont nous voulons changer la diagonale se nomment S1 et S2.
Dans un premier temps, nous cherchons à connaître l'indice du sommet de S1 opposé à S2 ainsi que celui du sommet de S2 opposé à S1, que nous appellerons respectivement iS1 et iS2. Nous avons donc la configuration suivante :

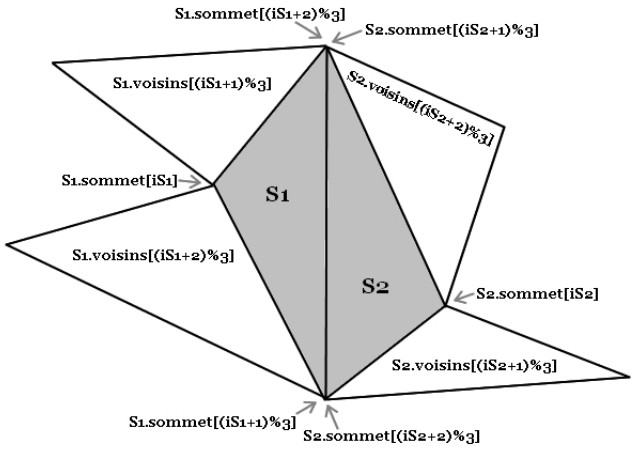

FIGURE 2 – Configuration avant l'échange

Grâce à cette configuration, il nous est possible de créer les nouveaux simplexes aisément, quel que soit l'indice du sommet opposé à S2. Les vertices composant les sommet de S1 sont stockés dans un tableau temporaire ; de même pour les voisins de S1 et les voisins de S2. On stocke également le sommet de S2 opposé à S1.
Tous ces éléments nous permettent de créer les deux nouveaux simplexes en changeant dans les structures de données des simplexes d'origine les valeurs qui correspondent à la configuration suivante :

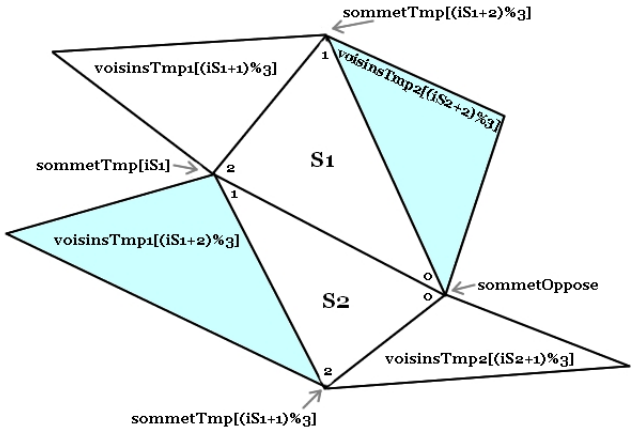

FIGURE 3 – Configuration après l'échange

Une fois que la diagonale a été changée, nous stockons les listes des vertices restants à insérer dans un temporaire puis nous les distribuons dans les deux nouveaux simplexes.
Il faut enfin notifier les deux voisins de S1 et S2 concernés (en bleu sur la figure 3) qu'ils ont changé de voisin.

5.3.3 Efficacité

Voici, pour un nombre de vertices données, le temps, en secondes, d'exécution de la triangulation avec l'échange de diagonale si nécessaire (l'affichage n'est pas pris en compte). Les tests ont été effectués sur l'une de nos machines personnelles et peuvent varier d'un ordinateur à un autre.

Nombre de vertices	Temps d'exécution (s)
10	0,004939
100	0,005378
1000	0,016825
10^4	0,506709
10^5	46,24618

L'ajout de cette boucle augmente de façon significative la complexité de l'algorithme car il n'est pas possible de savoir à l'avance la taille maximale que va atteindre la pile et le nombre de fois que la fonction d'échange de diagonale va être appelée. C'est cette dernière qui augmente la compléxité de l'algorithme et allonge le temps d'exécution du programme car elle nécessite de replacer dans la file de priorité les simplexes qui ont été modifiés.

D'après les résultats que nous avons obtenu, nous pouvons affirmer que le programme ne s'exécute pas en temps logarithmique, contrairement à ce que nous pouvions espérer.

6 Conclusion

L'objectif principal de ce projet, à savoir la triangulation d'un terrain en utilisant le principe de Delaunay et du *godness of fit* a été atteint. En effet, nos simplexes sont correctement créés et l'on peut voir que l'échange de diagonale fonctionne lorsque l'on affiche le résultat dans une fenêtre OpenGL. Nous avons également fourni une méthode de triangulation naïve qui utilise uniquement le principe de *godness of fit*.

La plus grande difficulté que nous avons rencontré a été l'implémentation de la fonction d'échange de diagonale et nous avons du procéder par étapes afin d'obtenir le résultat escompté. Dans un premier temps, nous avons traité les configurations initiales des simplexes cas par cas en fonction de leur orientation. Une fois que cette méthode fonctionnait, nous sommes passés à une implémentation plus générique (voir 5.3.2).