

Rapport du projet de la Génération de maillage pour le calcul scientifique

Jincheng KE

Une présentation du projet sur les résultats obtenus.

Mars 2024

Algorithme de Delaunay

Noyau de Delaunay

Localiser un point P dans \mathcal{T}^n

Pour localiser un point P dans un maillage non structuré, on peut le chercher à partie d'un triangle $K_0 = P_0P_1P_2$, ensuite, on calcule la signe de l'aire signée des triangles PP_1P_2 , P_0PP_2 , P_0P_1P , notées β_0 , β_1 , β_2 . Si tout β est positif, on a alors que le point P est bien dans ce triangle. Si β_i est négatif, alors on peut faire le même processus pour son i^{me} voisin qui peut être retrouvé dans le maillage grâce au tableau de hachage.

(cf. code static inline int localiser(Mesh *msh, int K, double2 P))

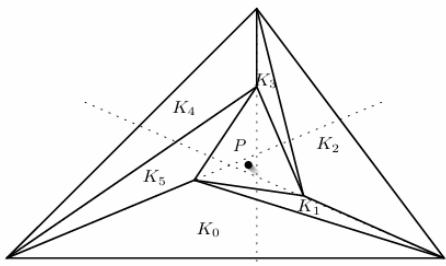


FIGURE 1

Notons que l'algorithme ci-dessus pourrait être bloqué par une boucle infinie. Par exemple l'algo est bloqué pour le FIGURE 1. On a donc besoin d'avoir un aléa lors qu'on choisit le numéro de voisin.

(cf. code static inline int eva_sgn(int* sgn_P))

Calculer la cavité C_P associée à P et retirer les triangles de C_p de \mathcal{T}^n

Un triangle est dans la cavité de point P si son cercle circonscrit contient P. On a déjà le triangle dans lequel P se trouve est bien dans la cavité de point P. On note ce triangle B_P . Pour trouver ensemble de tout triangle dans la cavité de point P, on peut appliquer le BFS ou DFS pour parcourir les voisins de B_P . De plus on peut améliorer cet algorithme par un élagage. On ne parcourt pas les voisins d'un triangle s'il n'est pas dans la cavité de point P.

(cf. code static inline Vector* get_cavity(double2 P, int K_P, Mesh* msh))

Pour retirer les triangles de C_p , il faut qu'on reconnaît les arêtes au bord de C_p et les laisse. Et on supprime les triangles. De plus, on réinitialise le tableau de voisins pour les triangles dans C_p aussi pour ses voisins.

(cf. code static inline int reinit_voi(Vector* cav, Mesh* msh))

Pour améliorer la complexité, on utilise le tableau de hachage.

(cf. code static inline HashTable *hsh_cav(Vector *cav, Mesh *msh))

En considérant notre structure de maillage n'utilise pas la liste d'arêtes. Donc on pourrait simplifier ce processus en fusionner le processus suivant.

(cf. code static inline int Bowyer_Watson(double2 P, DMesh* dMsh, Set* vertex_deleted))

Créer les triangles B_P en étoilant autour du point P

Le nouveau triangle est construit par relier le point P avec un arête au bord de C_p . On alloue le numéro d'un triangle supprimé à ce nouveau triangle s'il y a encore des numéros d'un triangle supprimé non utilisés. Sinon, on alloue le numéro NbrTri + 1. Cet algo est rapide car on a stocké tout numéro de triangle supprimé (La cavité

elle-même). De plus, comme le nombre de triangles est toujours croissant, cet algo ne produit jamais des trous dans la structure de maillage.

Dès qu'on a relié toutes les arêtes au bord de C_p avec point P et a ajouté tous les nouveaux triangles dans maillage, on doit mettre le tableau de voisins à jour pour la prochaine itération. Grâce au tableau de hachage et au fait que le mis à jour est local (il s'agit que les nouveaux triangles est les voisins de triangles de C_p), le mis à jour est rapide.

(cf. code `int msh_neighbors(Mesh *msh)`)

Structure de maillage dynamique

Comme le nombre de triangles et le nombre de sommets sont dynamique lors de la triangulation, on a utilisé la structure `DMesh` afin de faciliter de modifier la taille.

(cf. code `int dMesh_resize(DMesh* dmsh, int NbrVerMax, int NbrTriMax)`)

La triangulation de Delaunay

Les processus au-dessus est une partie de la triangulation de Delaunay.

(cf. code `static inline int Bowyer_Watson(double2 P, DMesh* dmsh, Set* vertex_deleted)`)

Étant donné des points pour générer un maillage avec la triangulation, on a besoin tout d'abord un maillage simple avec que 4 sommets et 2 triangles qui peut contenir tous les points donnés. Ensuite, on ajoute chaque points avec les processus au-dessus. Après avoir ajouté tous les points donnés, on obtient un structure dynamique final.

(cf. code `int Triangulation(double2* Ps, int NbrP, DMesh* dmsh)`)

Notons que le processus de la triangulation de Delaunay n'est pas encore fini. D'un côté, on a besoins d'un structure standard `Mesh` mais pas un structure dynamique `DMesh`. D'autre côté, il faut supprimer les 4 sommets extérieurs et les triangles contentant ces sommets. Ces processus est rapide parce que le numéro des 4 sommets extérieurs est toujours 1, 2, 3, 4. Les opérations restes sont donc de complexité linéaire.

(cf. code `Mesh* toMesh(DMesh* dmsh)`)

Validation

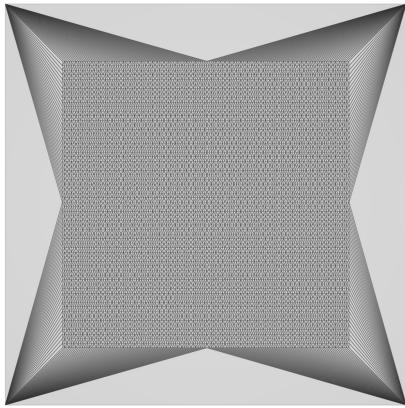


FIGURE 2 – Résultat intermédiaire

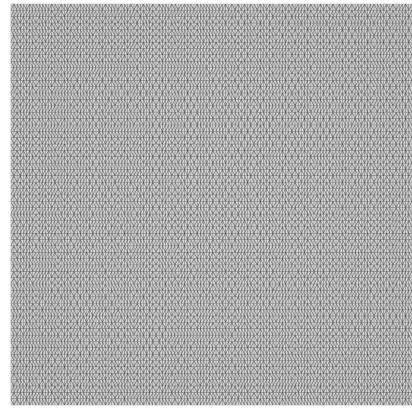


FIGURE 3 – Résultat final

Pour valider notre algorithme de la triangulation de Delaunay, on fait un test de gérerer un maillage uniforme sur un domaine de 5×5 , et de $Nx = 128$, $. La FIGURE 2 est le résultat intermédiaire de structure dynamique, et la FIGURE 3 est le résultat final. On peut observer que les points uniformes sont bien ajoutés dans la domaine. Les triangles sont bien créées et ajoutés dans le maillage.$

Application à la compression/décompression d'images

Une algorithme de la compression

Une algorithme de la compression qu'on a essayée est base sur le contour. Nous conservons plus de points proches du contour pour plus de détails, et moins de points éloignés du contour pour la compression.

Pour trouver le contour, nous pouvons nous référer au filtre de Sobel. Le contour c'est le lieu de gradient grand. On doit alors calculer le gradient sur chaque sommet.

Pour notre maillage triangle, le gradient de chaque élément est donné par la formule de cours 3. De plus, au moyen d'une moyenne pondérée en utilisant la superficie d'élément comme poids, on peut obtenir le gradient de sommets.

(cf. code `static inline double2* photo_gradient(Photo* pht)`)

Ensuite, nous pouvons calculer la norme de gradient pour chaque sommet, et normaliser sa norme dans [0, 1] par :

$$\frac{G - G_{\min}}{G_{\max} - G_{\min}}$$

Ensuite, étant donné un paramètre de la compression λ dans (0, 1), on fait un seuillage. Le sommet est préservé si sa valeur normalisée de gradient est supérieure que λ . Sauf que les 4 sommets aux 4 coins de la photo soient toujours préservés pour assurer la forme rectangle de la photo compressée. On aura donc le taux de compression $\eta_c = \frac{N_c}{N_H}$ est plus petit si le paramètre λ est plus grand.

(cf. code `cPhoto* compress_1(Photo* pht, double level1)`)

Finalement, on peut avoir notre photo compressée qui contient N_c positions de points 2D et N_c valeurs de gris. Si on stocker les valeur par le type `float`, alors cela fait $3*N_c$ de taille de `float`. Et pour la photo standard avec N_H pixels, la taille est N_H de la taille de `float`. De ce fait, la compression est significatif seulement si $\eta_c < \frac{1}{3} \approx 33.3\%$.

Algorithme de la décompression

Pour décompresser la photo compressée, on applique l'algorithme de la triangulation de Delaunay.
(cf. code `Photo* decode_cPhoto(cPhoto* cpt)`)

Validation

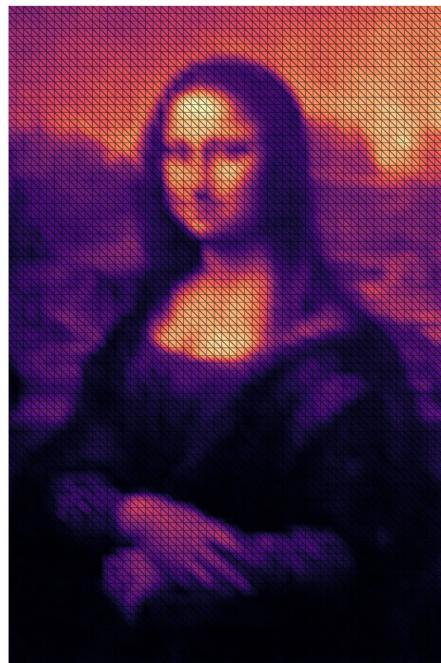
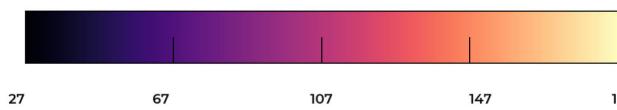


FIGURE 4 – Photo initiale en faible résolution

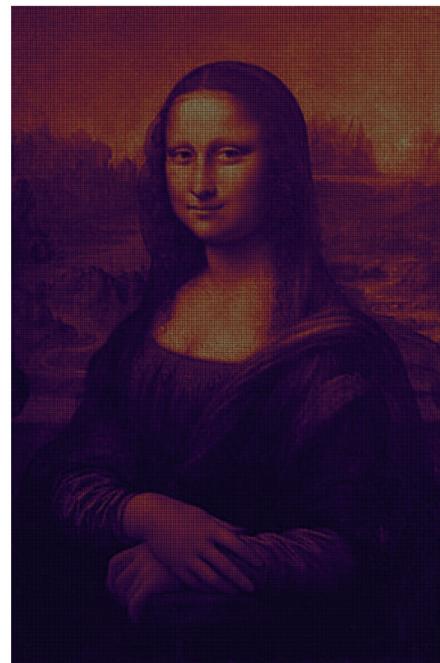
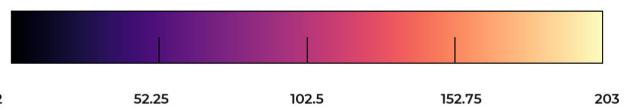


FIGURE 5 – Photo initiale en haute résolution

On va tester notre algorithme de la compression sur deux photos en résolution différent. La FIGURE 4 est une photo en faible résolution avec seulement 5104 pixels. La FIGURE 5 est une photo en haute résolution avec 127160 pixels.

Test sur la photo en faible résolution



FIGURE 6 – $\lambda = 0.2$

```
size_init = 5104
level = 0.200000
size_compressed = 826 ~= 16.183386 %
1
```

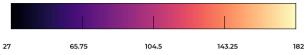


FIGURE 8 – $\lambda = 0.175$

```
size_init = 5104
level = 0.175000
size_compressed = 970 ~= 19.004702 %
1
```



FIGURE 10 – $\lambda = 0.15$

```
size_init = 5104
level = 0.150000
size_compressed = 1147 ~= 22.472571 %
1
```

FIGURE 7 – $\lambda = 0.2$



FIGURE 9 – $\lambda = 0.175$



FIGURE 11 – $\lambda = 0.15$



FIGURE 12 – $\lambda = 0.125$

```
size_init = 5104
level = 0.125000
size_compressed = 1377 ~= 26.978840 %
1
```

FIGURE 13 – $\lambda = 0.125$

FIGURE 14 – $\lambda = 0.1$

```
size_init = 5104
level = 0.100000
size_compressed = 1691 ~= 33.130878 %
1
```

FIGURE 15 – $\lambda = 0.1$

Les FIGURES au-dessus sont les résultats de la compression de la photo en faible résolution avec le paramètre de la compression $\lambda = 0.2, 0.175, 0.15, 0.125, 0.1$. On a le taux de la compression η_c pour chaque paramètre est environ 16%, 19%, 22%, 27%, 33%. On a pour $\lambda = 0.1$, la photo compressée préserve la plus part de détails de la photo initiale. Mais à cause de $\eta_c \approx 33\% \approx \frac{1}{3}$, on ne peut pas économiser bien l'espace de stockage. Pour $\lambda = 0.2$, la photo préserve une partie de détails de la photo initial. On peut voir clairement le contour. Mais la photo n'est pas suffisamment détaillée. De plus, le taux de compression $\eta_c \approx 16\%$. On économise environ 50% d'espace de stockage. Si l'on considère à la fois la qualité de l'image compressée et le taux de compression, le paramètre $\lambda = 0.15$ est un bon choix. Dans l'ensemble, la compression n'était pas très perceptible car l'image initiale n'était pas assez détaillée.

Test sur la photo en haute résolution

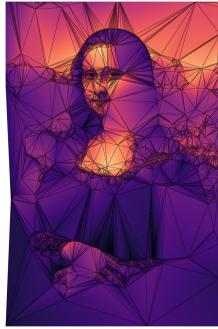


FIGURE 16 – $\lambda = 0.2$

```
size_init = 127160
level = 0.200000
size_compressed = 4695 ~= 3.692199 %
1
```

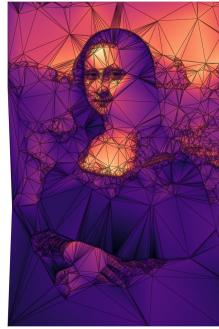
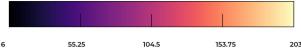


FIGURE 18 – $\lambda = 0.175$

```
size_init = 127160
level = 0.175000
size_compressed = 6311 ~= 4.963039 %
1
```

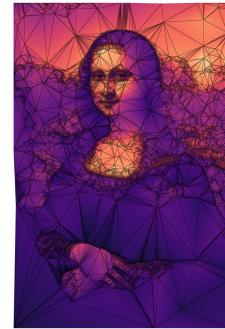


FIGURE 20 – $\lambda = 0.15$

```
size_init = 127160
level = 0.150000
size_compressed = 8699 ~= 6.840988 %
1
```

FIGURE 17 – $\lambda = 0.2$

FIGURE 19 – $\lambda = 0.175$

FIGURE 21 – $\lambda = 0.15$



FIGURE 22 – $\lambda = 0.125$

```
size_init = 127160
level = 0.125000
size_compressed = 12627 ~= 9.930009 %
1
```

FIGURE 23 – $\lambda = 0.125$

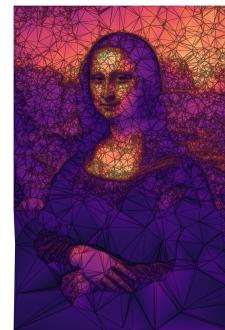


FIGURE 24 – $\lambda = 0.1$

```
size_init = 127160
level = 0.100000
size_compressed = 19281 ~= 15.162787 %
1
```

FIGURE 25 – $\lambda = 0.1$

Les FIGURES au-dessus sont les résultats de la compression de la photo en haute résolution avec le paramètre de la compression $\lambda = 0.2, 0.175, 0.15, 0.125, 0.1$. On a le taux de la compression η_c pour chaque paramètre est environ 3.7%, 5.0%, 6.8%, 9.9%, 15%. On a pour $\lambda = 0.1$, la photo compressée préserve beaucoup des détails de la photo initiale, et avec un taux de compression $\eta_c \approx 15\%$ qui nous permet d'économiser environ 55% d'espace de stockage. Et pour $\lambda = 0.125$, la photo ne perd pas beaucoup de détails, et le taux de compression $\eta_c \approx 9.9\%$ nous permet d'économiser environ 70% d'espace de stockage. Donc $\lambda = 0.125$ et $\lambda = 0.1$ sont tous des bons choix. Mais si on considère l'empreinte mémoire lors de décompression, c'est mieux de choisir $\lambda = 0.125$, ou d'améliorer l'algo de la décompression, car les 16 Go de mémoire vive ont été presque utilisés pendant la décompression.