Triangulation de Delaunay
Rapport de projet de programmation
L3 Informatique, parcours Science Informatique

Alexandre Drewery, Julien Duron et Odile Radet
1A département informatique
ENS Rennes et ISTIC, Université Rennes 1

26 octobre 2018

Résumé

Compte rendu du projet, codé en Ocaml, de Programmation 1 de première année informatique de l'ENS de Rennes. Ce projet consiste en l'étude et l'implémentation de la triangulation de Delaunay.

Introduction

Le projet est basé sur l'étude de la triangulation de Delaunay.

Le problème est le suivant : on cherche à trianguler un ensemble de points, de façon à ce qu'aucun des points ne soit dans l'intérieur strict du cercle circonscrit d'un des triangles (voir figure en introduction). Expérimentalement, ces triangulations tendent à favoriser l'apparition de triangles les plus équilatéraux possibles.

Nous avons proposé une implémentation basique d'un algorithme permettant de déterminer une triangulation de Delaunay pour un ensemble de points chosis aléatoirement (section1.1) avant de nous concentrer sur l'amélioration de l'expérience utilisateur. Afin de mieux illustrer la prédominance des équilatéraux, nous avons colorié la triangulation (section 1.2). Une extension a également été réalisée pour permettre à l'utilisateur de modifier la figure (section 1.3.1) ou de naviguer à travers cette dernière (section 1.3.2).

L'objectif premier du projet était d'acquérir une expérience de la modularisation intensive d'un programme et du travail en groupe sur du code. Une attention particulière a été portée à l'optimisation de l'algorithmique et du calcul. De part nos choix d'extensions, nous nous sommes également penchés sur la programmation réactive. La section 2 vise à mettre en lumière les causes et les conséquences de certains de nos choix.

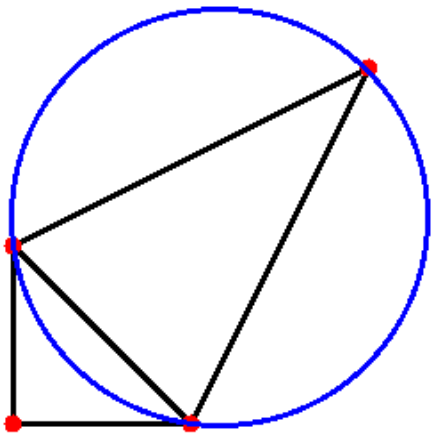

FIGURE 1 – Triangulation à 4 points : le cercle circonscrit à l'un des triangles a été tracé.

1 Contribution

1.1 Algorithme de triangulation

Pour obtenir une triangulation de Delaunay de notre ensemble de points, nous considérons l'approche itérative suivante. Supposons construite une triangulation de Delaunay $T = (S, A)$ où S est un ensemble de points, et A l'ensemble des arêtes les reliant. Si p est un point n'ap-

partenant pas à S, qui se situe dans l'envelope convexe de S, alors en notant C l'ensemble des points des triangles dont l'intérieur du cercle circonscrit contient p, et $Conv$ son enveloppe convexe, nous avons $T' = (S', A')$, une triangulation de Delaunay si $S' = S \cup p$, et $A' = (A \backslash \text{Conv}) \cup \{c, p | c \in C\}$.[1]

Sachant cela il faut, pour trianguler un ensemble fini de points S ne contenant pas deux points de mêmes coordonnées, le placer dans l'intérieur d'un convexe déjà triangulé. Nous ajoutons donc quatre points, sommets d'un rectangle contenant tous les points de S dont la triangulation de Delaunay est connue d'avance. Dorénavant, nous considérerons toujours des ensembles de points contenant les sommets de ce rectangle.

Notons que cette approche termine bien, car le nombre de triangles considérés est toujours fini, tout comme le nombre de points, donc le calcul de S' et A' termine toujours, or il est effectué exactement $\text{card}(S)$ fois.

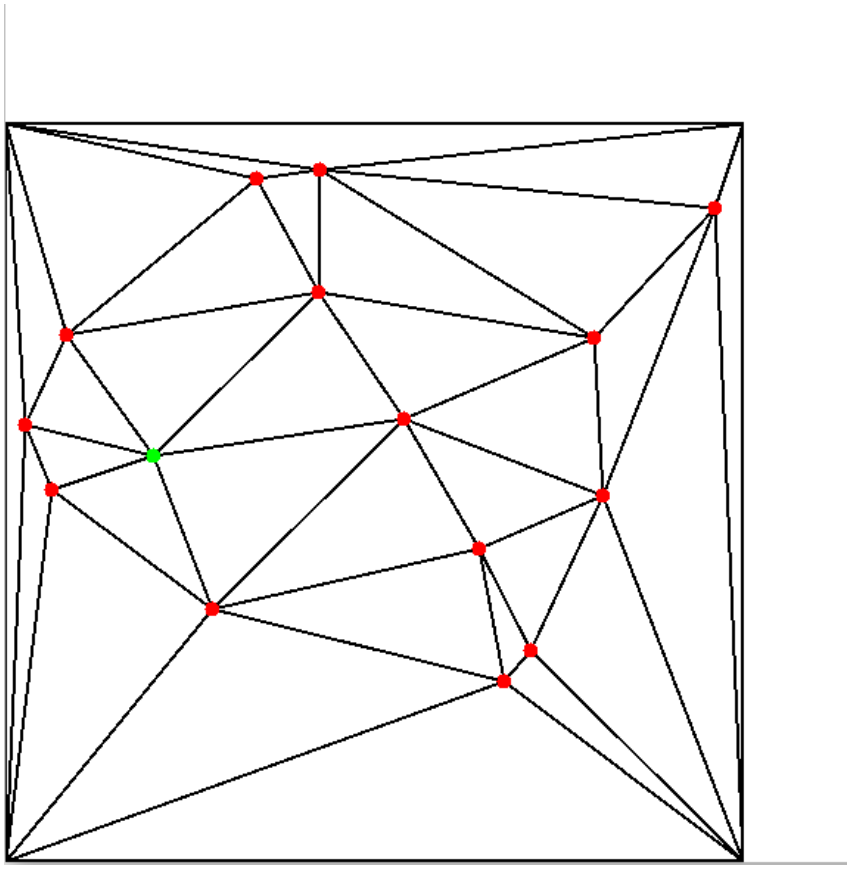

FIGURE 2 – Résultats expérimentaux pour 15 points.

1.2 Coloriage de la triangulation

Dans l'optique d'une meilleure expérience utilisateur, notre première extension visait à colorier les triangles en fonction de leur forme.

Nous avons mis en place une fonction qui à un triangle associe un triplet d'entier interprétable comme une couleur RGB. Ces entiers sont calculés à partir de la différence entre le

1. Énoncé du projet

plus grand angle et le plus petit angle du triangle. Les fonctions d'affichage ont également été modifiées afin de permettre le tracé de triangles en couleur.

L'utilisation de cette fonctionnalité conduit à une figure en teintes de bleu et de violet, les triangles les plus équilatéraux (différence entre le plus grand et le plus petit angle faible) étant affichés en bleu (voir figure en 1.2).

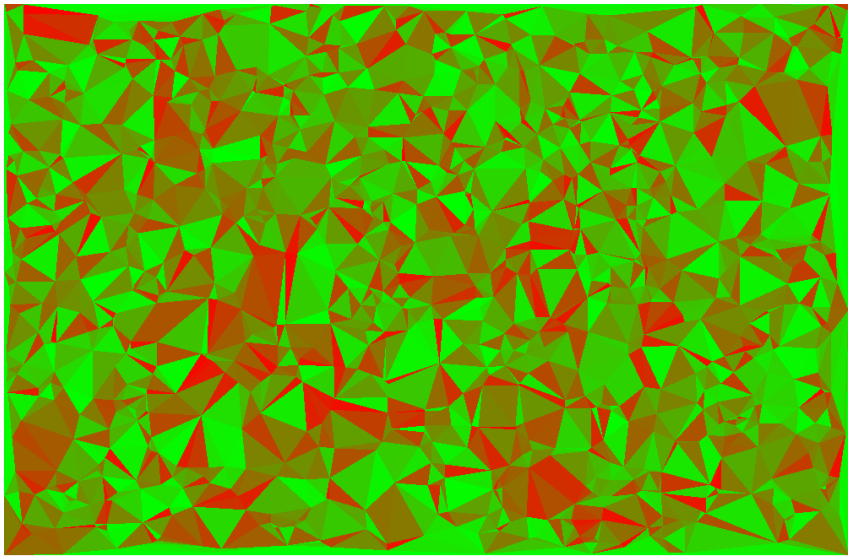

FIGURE 3 – Résultats expérimentaux de triangulation à 1000 points coloriée. Les couleurs ont été modifiées pour un meilleur rendu noir et blanc. Les "meilleurs" triangles apparaissent en vert.

1.3 Interactivité utilisateur

1.3.1 Modification de la triangulation

Le but premier de l'interaction est de pouvoir déplacer un point p de la triangulation tout en s'assurant que celle-ci est toujours de Delaunay en l'actualisant. Une approche simple peut être de recalculer entièrement la triangulation pour chaque nouvelle position de p. Cela implique que p ne soit jamais confondu avec un autre point. Mais cette approche nécessite de nombreux calculs pour chaque changement de position, ce qui rend l'interaction lente.

La biliothèque Graphics de Ocaml présente des fonctions adaptées à la programmation réactive : notre programme attend donc une entrée utilisateur (sous la forme d'un clic), et se charge alors de rechercher le point le plus proche de la position de la souris. Par la suite ce point peut-être déplacé et relaché à un emplacement au choix de l'utilisateur : le programme attend alors un relâchement du bouton de la souris.

Dans l'interaction telle qu'elle est conçue l'utilisateur choisit un unique point à déplacer à chaque action. Il s'agit donc de pouvoir actualiser une triangulation d'un ensemble de points dont un seul bouge.

Pour ce faire, considérons S notre ensemble de points, de positions distinctes deux à deux, et p appartenant à S, le point mouvant. Alors si $T = (S \setminus \{p\}, A)$ est une triangulation de Delaunay et si p est dans l'enveloppe convexe de S, l'algorithme présenté en 1.1 permet alors

de trouver une triangulation de Delaunay en ajoutant p à la triangulation T. Cela permet de n'avoir qu'un passage dans la boucle itérative présentée précédement, en supposant que l'on ait accès à T.

Pour calculer T, on exécute l'aglorithme de la section 1.1 sur $S \setminus \{p\}$, puis on garde T en mémoire tant que le déplacement a lieu, et nous renvoyons au module graphique une bonne triangulation pour chaque nouvelle position de p, si tant est que p n'est confondu avec aucun autre point et est dans l'enveloppe convexe de $S \setminus \{p\}$.

En modifiant très légèrement la fonction d'attente d'entrée utilisateur, il a été possible de lui faire détecter la pression d'une touche. Sous l'hypothèse que le curseur de la souris se trouve dans l'enveloppe convexe, nous pouvons ajouter un nouveau point à la triangulation sur pression de la touche N par simple application de l'algorithme de la section 1.1.

1.3.2 Navigation dans la triangulation

L'objectif premier de la navigation est de pouvoir se déplacer dans la figure (grossir ou réduire la triangulation et s'y mouvoir selon les deux directions du plan), en particulier quand celle-ci comporte un grand nombre de points.

La détection de l'entrée utilisateur n'a demandé que quelques modifications de notre fonction précédente qui recevait déjà les entrées claviers.

Nous avons mis en place un ensemble de nouveaux types :

```
type point_tp : {x : int; y: int};;
type triangle_tp = {p1 : point_tp; p2 : point_tp; p3 : point_tp};;
```

La triangulation n'est plus calculée qu'une seule fois au début, puis tous les points et les triangles sont stockés dans ces structures de données et transmis au module Draw pour le tracé.

À chaque entrée utilisateur, le module Calculs modifie chaque point et chaque triangle en fonction de la commande par translation et homothétie sur les coordonnées entières.

2 Discussion

2.1 Efficacité de l'algorithme

L'algorithme que nous utilisons est identique à celui de Bowyer et Watson à ceci près que nous considérons un rectangle comme triangulation de départ, et que nous ne retirons aucune arête à la fin. La complexité en est donc du même ordre, car il suffit de placer notre rectangle initial (et donc tous les autres points) dans un triangle plus grand, et d'appliquer l'algorithme de Bowyer et Watson ce qui nécessite strictement plus de calculs que notre algorithme, puisque nous faisons exactement la même chose, mais avec moins de triangles dans la triangulation. Or l'algorithme de Bowyer et Watson a une complexité en $O(n^2)$ [2], donc le notre à une complexité en $O((n-3)^2) = O(n^2)$.

2. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1029.3746&rep=rep1&type=pdf page 7

Pour illustrer ce fait, nous avons d'ailleurs utilisé R afin de tracer le nombre d'appels totaux à la fonction "in_circle" (qui s'exécute en temps constant), qui vérifie si un point appartient au cercle circonscrit d'un triangle en fonction de val, qui vaut le nombre de points au carré (voir figure en 2.1).

Cet algorithme est loin d'être le plus efficace, on trouvera dans la litérature plusieurs améliorations possibles comme le fait de trier les points afin de faire moins d'appels à "in_circle" inutiles [3].

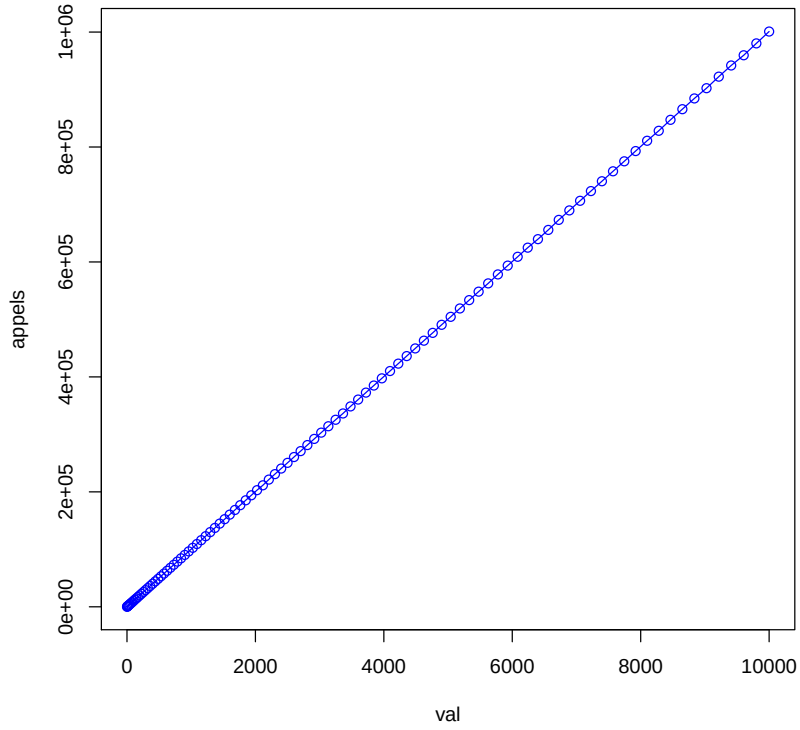

FIGURE 4 – Nombre d'appels à la fonction in_cicle en fonction du nombre de points de la triangulation. Résultats expérimentaux, courbe tracée avec un script R.

2.2 Extensions

2.2.1 Coloriage : sémantique

Le choix du caractère plus ou moins équilatéral des triangles comme critère de coloriage met en lumière la régularité de la triangulation.

La différence entre l'angle maximal et l'angle minimal d'un triangle n'était bien évidemment que l'une des méthodes que nous pouvions employer : en paticulier, nous aurions pu nous intéresser à l'écart-type des trois angles du triangles, mais nous ne l'avons pas fait.

3. `http://paulbourke.net/papers/triangulate/`

2.2.2 Modification : cas limites

Comme nous l'avons dit en section 1.3.1, il existe des restrictions lors du déplacement d'un point sur sa position pour que notre algorithme puisse fonctionner. Celui-ci doit se trouver dans l'enveloppe convexe des autres points (donc en particulier ne pas être un point constituant l'enveloppe convexe), et il ne doit pas avoir la même position qu'un autre point de la triangulation.

Pour gérer ces deux cas de figure, nous avons choisit de rendre inamovibles les sommets de notre rectangle initial. De plus, si l'utilisateur choisit de faire sortir sa souris du rectangle imposé, la position du point sera bornée dans le rectangle, l'empêchant de sortir du convexe. Enfin si le point est lâché non loin d'un autre point (distance strictement positive choisie arbitrairement) alors le programme remet le point à sa position initiale. De même, si l'utilisateur tente d'ajouter un point dangereusement proche d'un autre déjà existant, le programme refuse la commande.

Ces choix présentent l'avantage principal de garantir la validité de la figure une fois les modifications terminées : à la fin de chaque modification, ou bien nous disposons d'une figure valide car nous nous trouvons dans les conditions d'application de l'algorithme, ou bien nous avons tenté une action illégale et nous revenons à l'état précédent.

Néanmoins, dans le cas du glisser-déposer, rien ne garantit la validité de la figure tant que le point n'a pas été posé. En pratique, l'utilisateur se trouve effectivement devant une triangulation de Delaunay tant qu'il ne passe pas sa souris sur un autre point. Nous avons tenté d'implémenter une zone de sécurité inaccessible à la souris autour de chaque point, mais cette sécurité supplémentaire existait au détriment de la fluidité de l'interaction. Nous avons donc jugé que garantir la validité a fortiori était un compromis acceptable.

2.2.3 Navigation : structure de donnée

Le choix d'utiliser une structure de points à afficher sur lesquels les calculs sont faits a pour avantage principal de simplifier les calculs : quelque soit la situation actuelle à l'écran, la même opération est toujours effectuée sur tous les points en réponse à une commande donnée.

Cette idée était un premier jet d'implémentation, inspiré par le fait que c'est le module Draw qui récupère l'entrée utilisateur. Néanmoins, elle a plusieurs défauts majeurs : perte de précision au dézoom dûe à l'emploi d'entiers dans les calculs, résultats aberrants suite à un zoom trop important, etc. Ceci conduit à des figures qui, quoi qu'esthétiques, ne sont plus valides (voir figure en 2.2.3). Pour corriger ce problème, il eût par exemple fallu que le module Calculs eusse géré une conversion type intrinsèque-type d'affichage à chaque nouvelle commande de l'utilisateur.

De plus, cette extension n'est en l'état actuel pas compatible avec la précédente : il eût fallu que nous eussions été capables de replacer l'action de l'utilisateur sur la figure dans le contexte d'un zoom particulier sur une partie précise de la triangulation. Ce problème se résout de la même façon que le précédent, à l'aide d'un passage systématique par le type intrinsèque au sein du module Calculs quand le module Draw reçoit une commande.

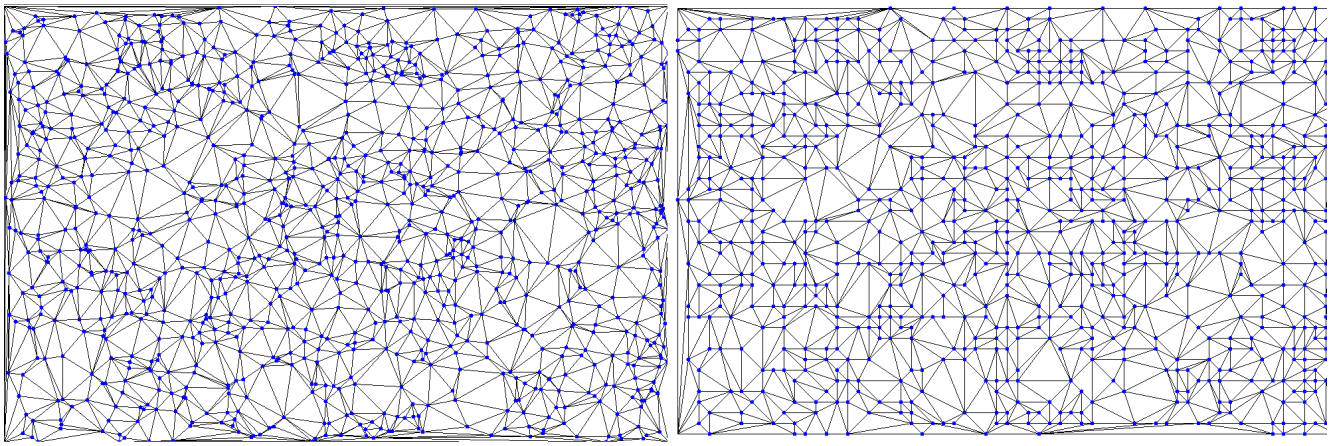

FIGURE 5 – Résultats expérimentaux pour 1000 points avant et après une compression par dézoom abusif.

Conclusion

Nous avons implémenté une triangulation de Delaunay et une interface utilisateur partielle et imparfaite autorisant quelques actions sur la figure.

Plusieurs idées généales auraient pu permettre une amélioration significative de notre programme : en particulier, partitionner la triangulation aurait permis d'accélérer les calculs sur tous les plans : recherche du point le plus proche pour la modification, recherche des triangles dont le cercle circonscrit contient un point pour la triangulation, etc.

D'autres part, la suppression de point aurait également pu être implémentée. L'ajout de cette dernière et la compatibilité modification-navigation auraient complété l'interface utilisateur.