Willy Jacquet, Yoan Parra

Géométrie algorithmique et maillage

Topologie

Dans cette section, nous présentons la représentation de maillage employée et les considérations à ce propos.

Choix indice/pointeur

La premier choix de conception est la manière d'identifier un élément d'un maillage (triangle, sommet, arête, ...). Ce choix est directement lié aux structures de données sous-jacentes. Les pointeurs ne permettent pas une réallocation des données et conduisent donc à une forte fragmentation des données, ce qui peut gravement nuire aux performances en temps et en mémoire. En revanche, ils permettent un accès direct à la donnée pointée ce que les indices ne permettent pas, cependant ce gain est négligeable comparé aux pertes précédemment évoquées.

En termes de commodité, l'usage des proxies (cf. section Architecture), encapsulant les indices, permet d'obtenir une interface identique à un pointeur. De plus, les éléments qui ne sont pas représentés directement par le maillage (arêtes dans un représentation de maillage "face-vertex") ne peuvent pas être identifié facilement à l'aide de pointeurs.

Les indices permettent également une sérialisation naïve des données.

Nous avons donc opté pour les indices, plutôt que les pointeurs, pour les raisons précédentes. Le premier indice que nous considérons valide est le 0.

Maillage triangle

Nous avons premièrement disposé du maillage triangle, ne conservant aucune donnée d'adjacence, pour le rendu graphique. Celui-ci étant recevable tel quel par la plupart des APIs graphiques, il convient dès lors de télécharger les attributs de sommets dans la mémoire graphique et de fournir les indices de triangles pour effectuer le rendu d'un maillage.

“Face-vertex mesh”

Nous ne considérons que les faces à trois côtés (triangles) bien que ce modèle puisse être adapté à plus. Nous ne permettons, également, pas les faces moindres, c'est-à-dire des points ou des arêtes libres qui n’appartiendraient pas à un triangle.

Dans cette représentation, les informations topologiques disposées sont les suivante:

- pour un triangle:
    - 3 indices de sommet incident,
    - 3 indices de triangle adjacent;
- pour un sommet:
    - 1 indice de triangle incident.

Les indices des sommets sont triés dans le sens trigonométrique d’apparition et les indices de triangles sont arrangés de sorte à ce que l’indice relatif du sommet corresponde à l’indice relatif du triangle adjacent en opposition à ce sommet.

Lorsqu’un triangle n’a pas de triangle adjacent à l’une de ses faces, il stocke son propre indice, permettant ainsi de détecter facilement ce cas.

Invalidation d’éléments

L’invalidation d’élément est une primitive essentielle à de nombreux algorithmes; néanmoins, nous ne maintenons pas une liste d’éléments invalidés et nous ne ré-utilisons pas les emplacements mémoire ceux-ci lors de la création d’un nouvel élément. Ce choix d’implémentation peut s’avérer important lorsqu’il est nécessaire de supprimer et d’ajouter des éléments successivement de nombreuses fois.

Cependant, nous nécessitons tout de même d’une façon de marquer un élément comme supprimé en temps constant. Afin d’éviter complètement un surplus mémoire pour le maintien de cette information, nous l’avons encodé dans les informations topologiques déjà présentes:

Un sommet est invalidé si son indice de triangle incident est égal à -1.

Un triangle est invalidé si n’importe lequel de ses indices de sommets incidents est égal à -1.

Identification des arêtes

Nous avons utilisé trois manières d’identifier une arête:

- 2 indices de sommet;
- 2 indices de triangle;
- 1 indice de triangle et 1 indice de sommet.

On remarque que, pour chaque description, il est possible d'identifier une même arête de deux manières différentes. En effet, les couples de sommets (v0, v1) et (v1, v0) identifient la même arête. Cela peut être exploité pour représenter les arêtes orientées, cependant cette distinction n'est pas toujours voulue. L'identification unique peut être réalisée grâce à une relation d'ordre sur les indices (ex. on ne considère le couple (v0, v1) que seulement si v0 < v1).

La représentation par deux triangles n'est pas pratique; on ne peut que caractériser les arêtes qui sont à l'interface de deux triangles. Cela pose, par exemple, problème pour les bordures d'une triangulation 2D.

Nous avons donc constaté le couple triangle/sommet être la représentation la plus pertinente. Elle reste néanmoins moins générique qu'une représentation sommet/sommet puisqu'elle ne permet pas de représenter trivialement les deux orientations d'une même arête d'un triangle en bordure, c'est-à-dire que cette arête n'est pas partagée par deux triangles.

Sommets infinis

Dans un premier temps nous avions entrepris d'utiliser les sommets infinis.

Ceux-ci requièrent cependant un effort conséquent dans leur maintien: il faut pouvoir marquer un sommet comme tel et introduire des triangles de nature différente car reliés à ce sommet. Dans certains cas, comme pour une bande de N triangle, il faut introduire N + 2 nouveaux triangles, ce qui est difficilement acceptable. Chaque élément doit par la suite être systématiquement testé pour savoir s'il doit être considéré comme appartenant au maillage "réel". Un maillage doit contenir autant de sommets infinis qu'il y a de composantes connexes et de trous.

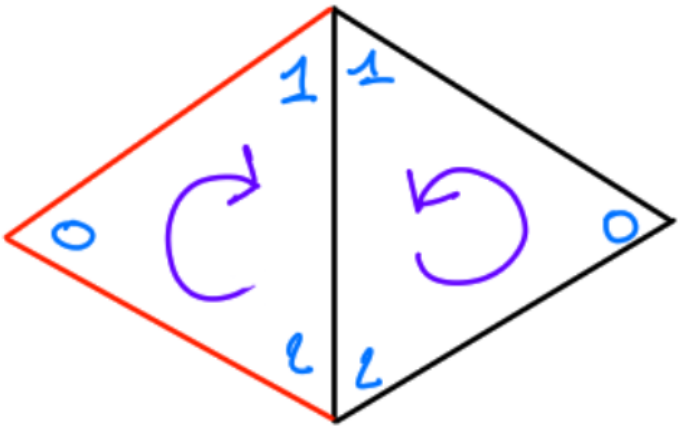
Cela introduit un surcoût en temps et en mémoire, et demande des efforts de conception considérables. Nous ne sommes pas parvenus à en réaliser une implémentation décente.

Ils permettent néanmoins d'implémenter de manière assez élégante certaines primitives comme les itérateurs / circulateurs sur les faces.

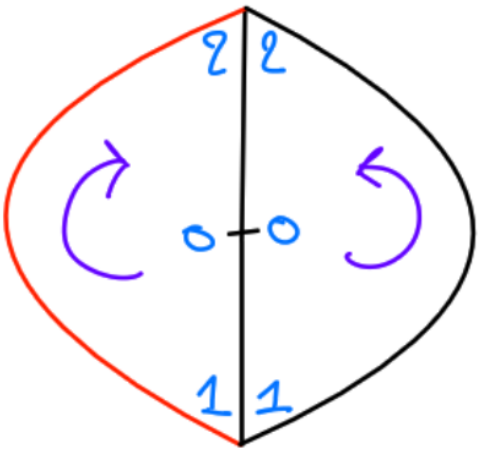
Triangles "réflectifs" (nom temporaire)

Pour pallier aux problèmes précédemment évoqués, nous avons imaginé - mais probablement pas inventé - une méthode permettant de retenir une formulation générique des algorithmes pour un surcoût mémoire nul, ne nécessitant pas d'introduire de nouveaux éléments.

Pour cela on considère que les bordures de triangles (pas de triangle adjacent) mènent au même triangle, mais que l'ordre de parcours est inverse (anti-trigonométrique).



Cela ne semble pas poser de problème si le triangle a 2 ou 3 bordures.



Il suffit dès lors de retenir dans les itérateurs si l'on se trouve dans le "monde miroir" ou non. Cela peut également être interprété comme le dessous ou le dessus d'un triangle. Nous n'avons néanmoins pas eu l'occasion d'explorer cette piste.

Algorithmique

Dans cette section nous présentons les algorithmes mis en œuvre.

Prédicats

Les seuls prédicats dont nous ayons eu besoin sont les prédicats d'orientation par rapport à un segment en 2D, d'orientation par rapport à un triangle en 3D et enfin d'appartenance par rapport à un triangle 2D.

Dû au fait que nous n'utilisons pas une représentation numérique rationnelle, ceux-ci échouent régulièrement et peuvent entraîner des résultats d'algorithmes faux, transgressant les invariants, sur des triangles avec un mauvais "aspect ratio".

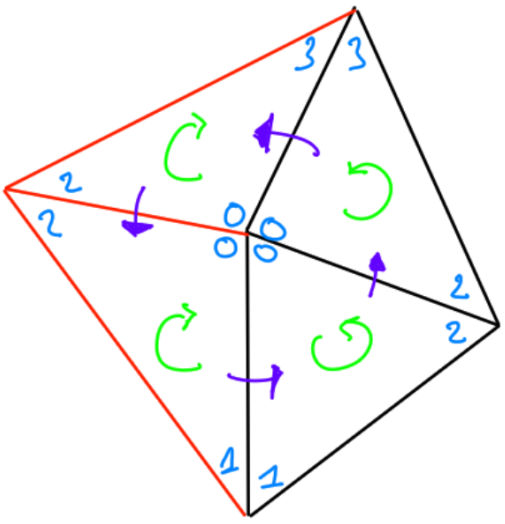
Opérations primitives

Parcours

Nous avons utilisé les portées ("range" dans la langue de Shakespear), généralisant les itérateurs et les circulateurs, à l'aide de la superbe bibliothèque C++ "range-v3" par Eric Niebler.

N'utilisant pas les sommets infinis, mais les triangles "réflectifs", les algorithmes employés sont différents de ceux présentés en cours.

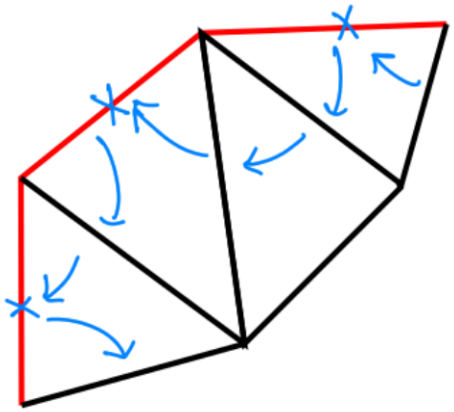
Parcours des faces adjacentes à un sommet



En rouge, les triangles réfléchis.

Le parcours des faces adjacentes

Parcours des arêtes de l’enveloppe convexe



En rouge, les arêtes de l'enveloppe convexe.

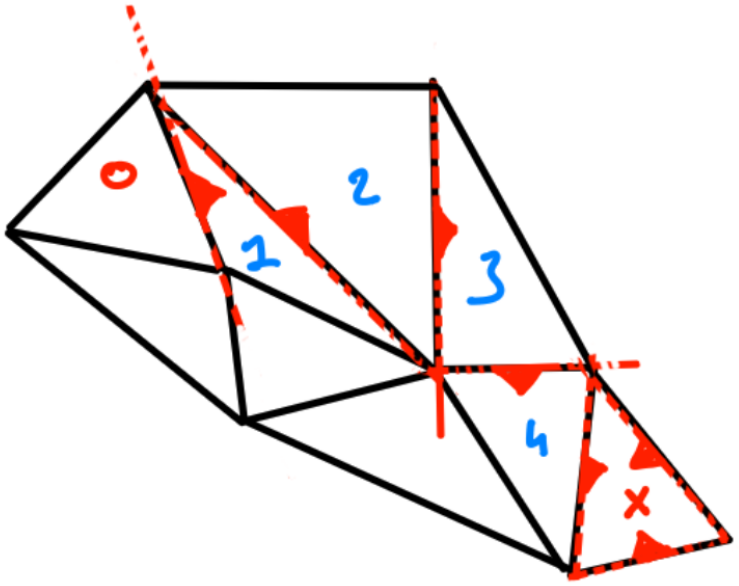
Le parcours des arêtes de l’enveloppe convexe est utilisé pour l’insertion
Cette méthode est moins efficace que si l’on avait utilisé les sommets infinis, cependant les résultats peuvent être retenus et les performances . Nous avons estimé

Opérations notables

Comme mentionné précédemment, nous pensons qu’il est important de fournir une interface préservant les invariants, évitant ainsi à l’utilisateur tout “accident”. En particulier, nous mettons en avant les opérations fréquemment utilisées dans les algorithmes qui suivent.

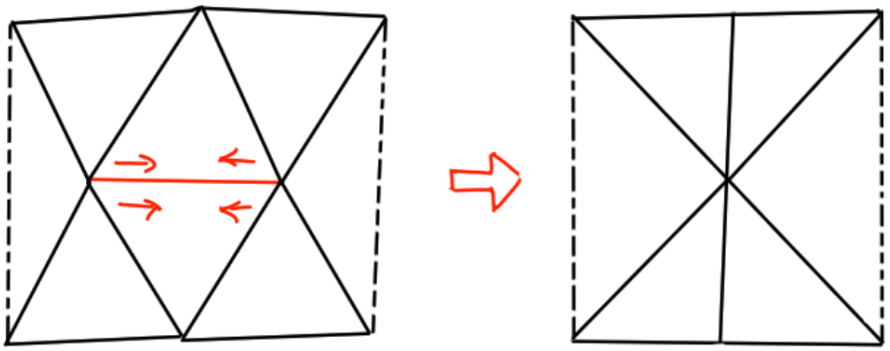
Nous remarquons à nouveau qu’il est possible de découpler totalement ces opérations en termes de géométrie et de topologie. La géométrie n’intervient que pour définir les conditions préalables aux maintien d’invariants et pour la modification des éléments résultants.

Marche 2D



Recherche d'un triangle contenant un point, ou sinon une bordure le voyant, dans une enveloppe convexe à partir d'un autre triangle. Renvoie un triangle.

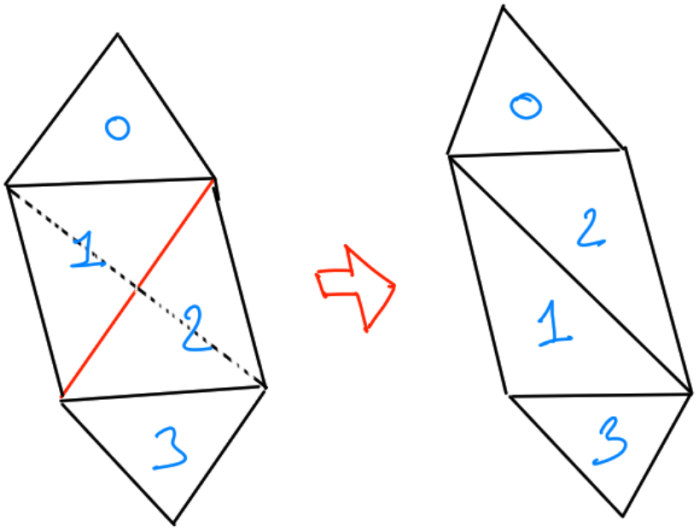
"Edge collapse"



Supprime deux triangles et un sommet. Modifie uniquement la topologie. Renvoie le sommet restant.

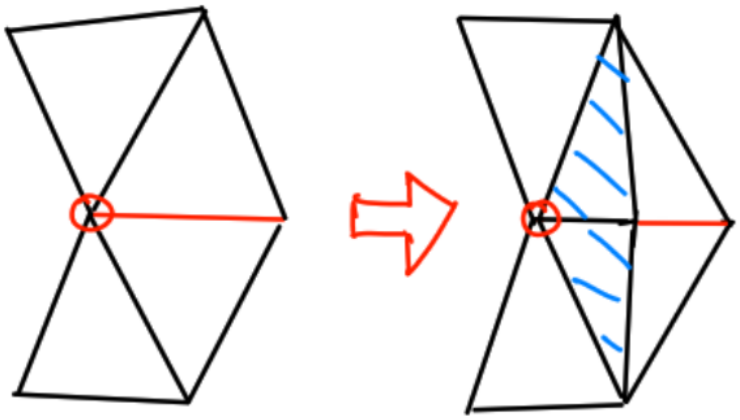
Il est nécessaire de respecter une condition sur la topologie et une autre sur la géométrie pour conserver les invariants. Nous les vérifions dans notre test pour savoir si une arête est localement de Delaunay.

“Edge flip”



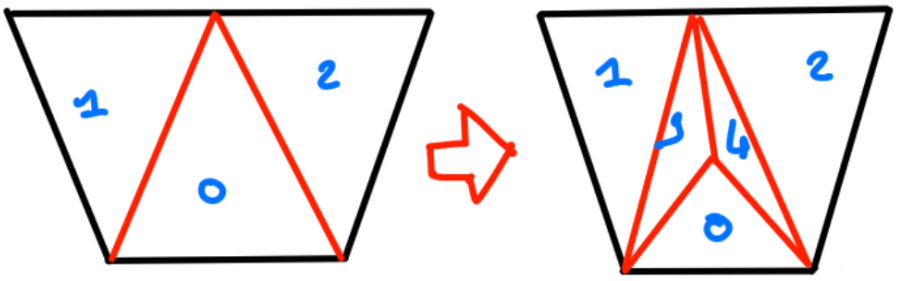
N’invalide, ni ne crée rien: modifie seulement la topologie. Ne renvoie rien.

“Edge split”



Crée deux triangles et un sommet. Renvoie le nouveau sommet.

“Triangle split”



Crée deux triangles et un sommet. Renvoie le nouveau sommet.

Triangulation 2D

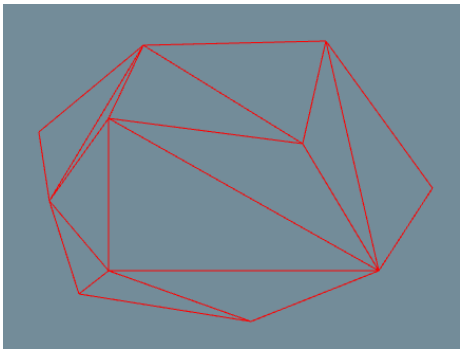
Lawson

Nous avons implémenté l'algorithme de Lawson permettant de rendre une triangulation de Delaunay.

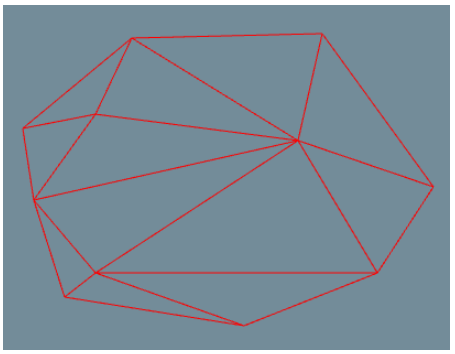
L'insertion à l'extérieur est également moins compliquée sans sommet infini; elle ne nécessite pas d'effectuer des flips sur les triangles reliés à ce sommet, seulement de créer les nouveaux triangles et de les relier à la topologie existante.

Cet algorithme fait usage de la marche au sein d'une triangulation 2D et du flip d'arête. Des contraintes strictes sur la topologie et la géométrie doivent être vérifiées sur une arête avant de la flipper.

Exemple d'une triangulation quelconque sur laquelle l'algorithme sera appliqué:

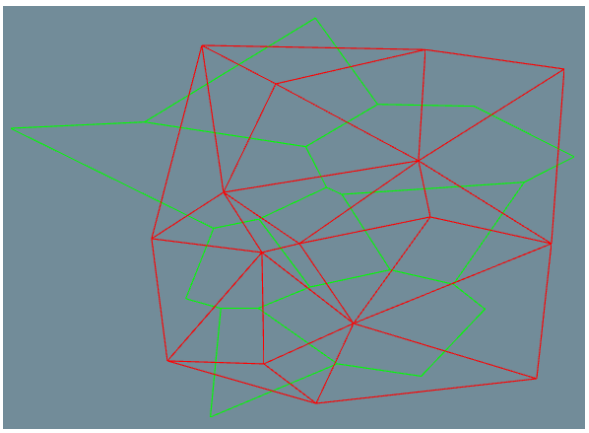


Après application de l'algorithme, on obtient la triangulation de Delaunay suivante:



L'insertion progressive de Lawson est également prise en charge.

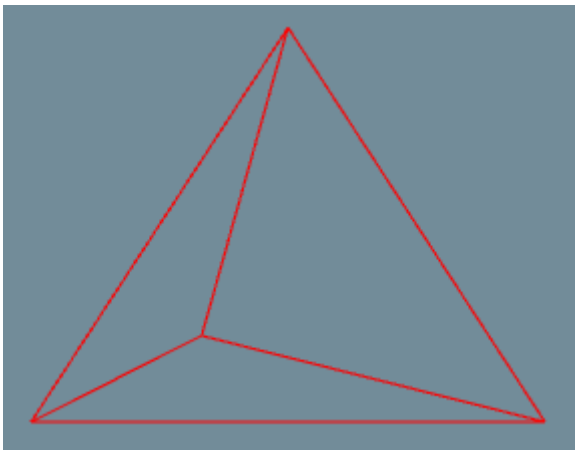
L'interface permet d'afficher le diagramme de Voronoï dual de la triangulation:



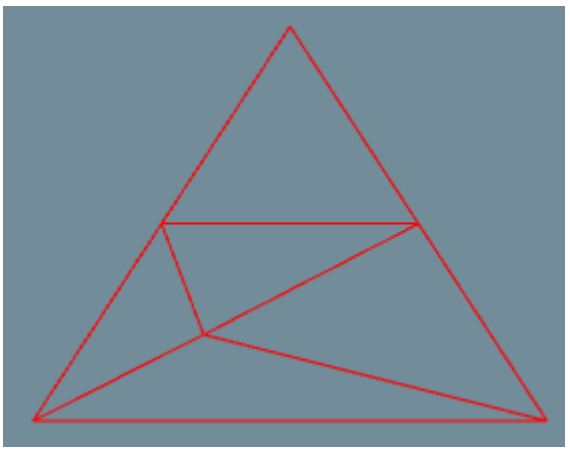
Ruppert

Nous avons implémenté l'algorithme de raffinement de maillage de Ruppert en considérant comme arêtes contraintes les arêtes de l'enveloppe convexe (faute d'une interface permettant de marquer des arêtes comme telles).

Sur l'exemple suivant, nous interdisons les triangles avec angle de moins de 20 degrés:



Après exécution de l'algorithme, on obtient la triangulation suivante:



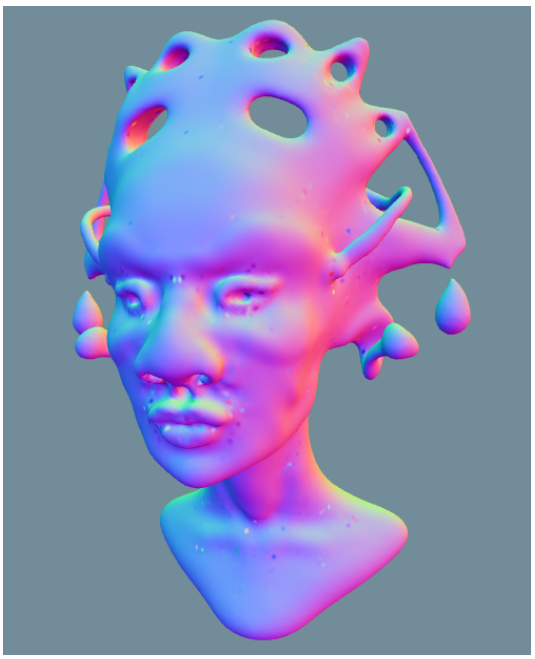
Triangulation 3D

Laplacien

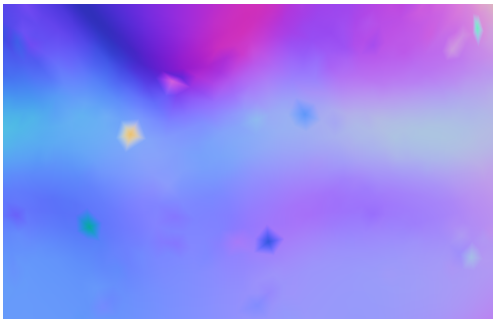
Nous avons implémenté l'approximation des normales et de la courbure moyenne à l'aide du Laplacien.

Sur le modèle "Queen", on obtient les normales (en fausses couleurs) suivantes:

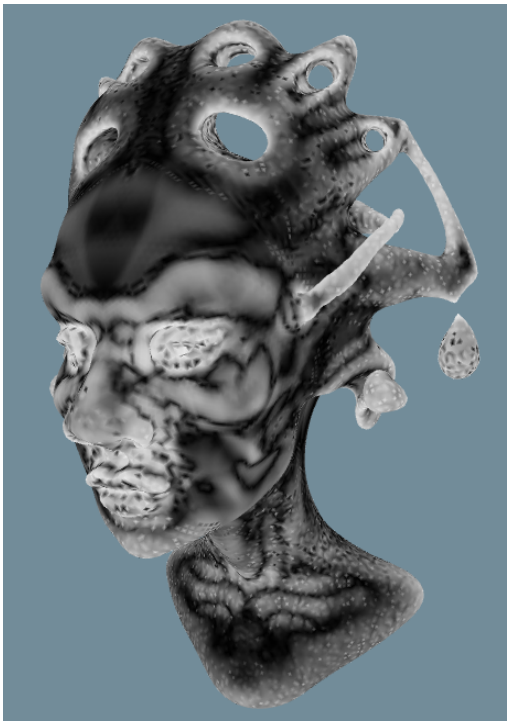
Cependant, on observe que les normales peuvent être dans la mauvaise direction suivant si la courbure est positive ou négative. Après correction, on obtient:



Le Laplacien est très sensible (opération sur un voisinage de taille 1) et peut donc produire des incohérences visuelles:



Le Laplacien permet également de calculer la courbure moyenne, ici, en fausses couleurs:

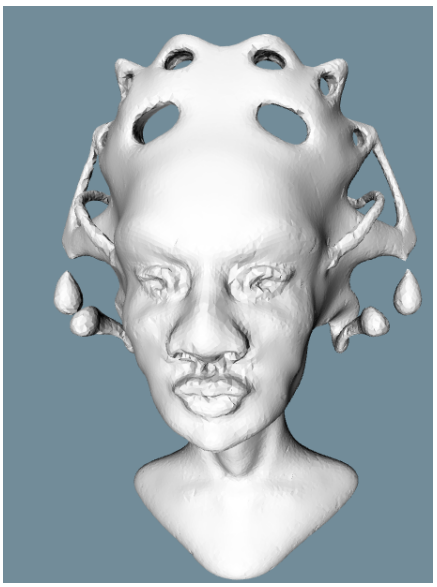


Plus la couleur est claire, plus la courbure moyenne est importante.

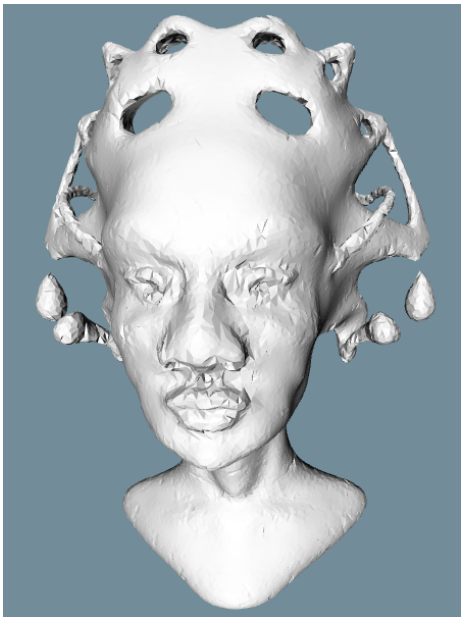
Simplification

À l'aide de l'edge collapse, nous avons implémenté une simplification de maillage aléatoire.

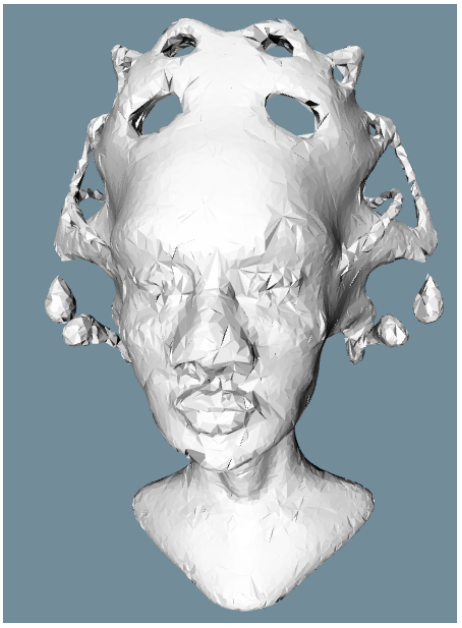
Maillage initial.



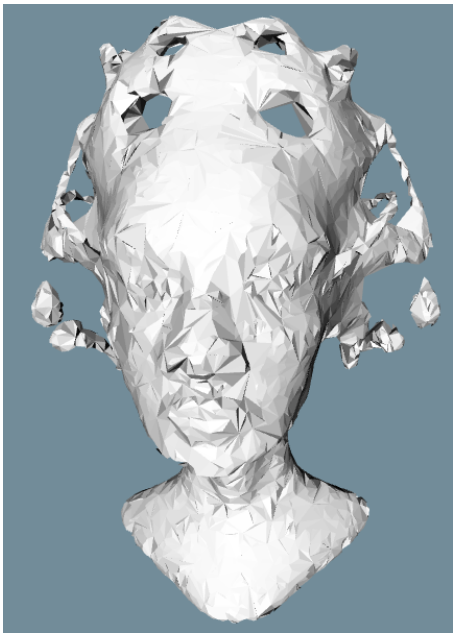
Retrait de 25% des triangles.



Retrait de 50% des triangles.



Retrait de 75% des triangles.



Retrait de 90% des triangles.

Retrait de 97% des triangles.

Architecture

Dans cette section, nous expliquons les considérations principales dans l'architecture du code, sans implication avec les algorithmes présentés, mais d'une importance capitale pour la qualité logiciel.

Orthogonalité géométrie / topologie

On remarque que les attributs associés à un élément (position, poids, couleur, normale, courbure, ...) sont indépendants de la topologie du maillage et qu'ils ne sont corrélés que par l'intermédiaire d'algorithmes, les calculant ou les utilisant.

Une conséquence importante de ce design est de pouvoir réutiliser la même géométrie pour différentes topologies: non seulement d'utiliser les mêmes modules dans le code mais aussi de pouvoir intervertir différentes topologies.

Cette distinction permet également d'aboutir à une grande simplification des algorithmes précédemment illustrés dès lors que l'on remarque que nombre de ceux effectuent les mêmes opérations sur la topologie. Par exemple, de nombreux algorithmes de simplification de maillage utilisent l'"edge collapse" et ne différent que par la géométrie (position) du sommet restant.

La topologie peut également être considérée par élément, pouvant possiblement permettre de combiner différentes formulations. Nous ne sommes cependant pas aventuré sur ce domaine là car nous n'avons pas eu à manipuler d'autres topologies de maillage que "face-vertex"

Orthogonalité données / sémantique

Du point de vue utilisateur, on souhaite avoir accès à tous les attributs d'un élément du maillage (position, normal, couleur, etc) et la manière naïve de parvenir à cela serait de stocker ses informations dans des structures associées à chaque élément.

L'impact d'un tel design est discuté en profondeur au sein du paradigme de la programmation orientée données ("Array of Struct" / "Struct of Array"). Il est préférable de stocker chaque collection d'attribut dans sa propre structure de données, augmentant grandement la localité des données, et donc les performances.

Patron de conception "Proxy"

Cependant, le choix de conception précédent rend la manipulation des éléments moins évidente puisque les attributs d'un même élément sont répartis au sein de plusieurs structures de données.

La patron de conception "proxy" permet de résoudre ce problème en fournissant une interface pour chaque élément en donnant un accès direct aux attributs d'un élément, sans se soucier de leur stockage.

En plus de cela, un proxy fournit également un accès au contexte de cet élément et permet donc de formuler des opérations locales, nécessitant de connaître le voisinage, à partir d'un unique proxy.
Un proxy est simplement implémenté comme l'index d'un élément accompagné d'un pointeur sur le maillage associé.

Vues géométriques

Entre nos deux applications 2D et 3D, nous avons eu besoin d'utiliser des attributs géométriques différents:

- en 2D: position 2D, hauteur paraboloid;
- en 3D: position 3D, normales, courbure moyenne.

Mais d'autres applications peuvent nécessiter des attributs géométriques supplémentaires (UV, couleur, etc).

Une mise en œuvre générique se doit donc de supporter un nombre quelconque d'attributs et de types arbitraires, mais aussi variables au cours de l'application. Cela peut être réalisé grâce au méthodes de "type erasure" qui impliquent une gestion externe des attributs par le biais de vue géométriques; les algorithmes ne peuvent pas inférer les attributs présents et leurs types.

Interviennent les vues géométriques, dont sont spécifiés les attributs requis et leurs types. La vue, par construction, récupère les structures de données sous-jacentes aux attributs et fournit un accès à ceux-ci grâce à l'indice d'un élément. Cette mise en œuvre est, en particulier, "type safe" - propriété fortement désirée du langage C++. Les algorithmes doivent donc spécifier la vue qu'ils nécessitent pour s'exécuter et pourront obtenir un accès, en temps constant, aux attributs des éléments.

Ces deux concepts d'orthogonalité permettent d'aboutir à une représentation de maillage à géométrie et à topologie arbitraires, indépendamment l'une de l'autre, mais aussi dynamique, pouvant changer au cours de l'exécution.

La mise en œuvre des vues n'a cependant pas pu aboutir à temps pour notre rendu.

Méthodologie

Dans cette section, nous expliquons comment nous avons progressé au long du développement et les méthodes qui en ont émergées.

Maintien des invariants

Au fil du développement, il nous est apparu rapidement que l'écriture d'algorithmes est particulièrement difficile: une topologie incorrecte ne peut avoir de répercussion que lorsqu'elle est utilisée comme entrée d'un algorithme, et donc ne pas causer de problème dans l'immédiat mais plusieurs opérations en amont.

Débugger une séquence d'opérations est particulièrement pénible car d'une unique faute peut émerger une série de nouvelles erreurs, et par récurrence, produire un résultat incorrect. Nous avons donc pris soin de formuler les invariants et de les vérifier systématiquement et automatiquement, nous assurant ainsi, au moins, de la validité topologique de nos algorithmes à chaque étape.

Haut niveau abstractif

Fournir une interface de haut niveau respectant ces invariants a grandement permis de faciliter le développement.

Certaines suites d'opérations sont sémantiquement identifiables et, par conséquent, nommables. De fait, le code se rapproche dès lors bien plus de sa formulation en pseudo-code; ce qui le rend, par conséquent, bien plus compréhensible, et donc debuggable et maintenable.

Ce haut niveau abstractif est obtenu grâce aux proxies permettant d'accéder à toutes les informations relatives à un élément ainsi que son voisinage ou, encore , n'importe quelle autre information du maillage. Les indices d'éléments ne sont manipulés que lorsque ultimement nécessaires.

Le code écrit est cependant moins optimisé du fait de moins de factorisation entre les calculs successifs mais le code résultat est plus court, moins redondant et plsu compréhensible.

Typage sémantique

Les indices des différents éléments sont représentés par des entiers (non signés). Seulement, bien qu'ils aient la même représentation, ils identifient des objets de natures différentes. Nous avons donc encapsulé ces indices dans des classes "VertexIndex, TriangleIndex, RelativeIndex, ...". Cela permet de détecter un nombre important d'erreurs d'inattention à la compilation sans aucune conséquence sur les performances de l'application.

Conclusion

Nous nous sommes concentrés sur la mise en place d'outils pour pouvoir expérimenter et prototyper rapidement. Se placer dans un cadre de plus long terme nous a permis de considérer des problématiques différentes comme la maintenabilité et l'extensibilité. En a résulté une mise en œuvre générique et originale qui pourra être maintenue et intégrée au sein de projets futurs.