Épreuve Commune de TIPE

FICHE SYNOPTIQUE 2013

AUVOLAT−−BERNSTEIN Alex, 11818

Sujet : **Découpage en polygones convexes pour la recherche de chemin**

Découpage en polygones convexes pour la recherche de chemins

1. Motivation

Passionné par les algorithmes relevant de l'intelligence artificielle, il m'a paru intéressant d'étudier un algorithme de recherche de chemin. Un tel algorithme peut trouver son utilité aussi bien dans le domaine de la robotique que dans celui des jeux vidéos...

2. Objectifs

Étant donné le plan d'un lieu, on cherche à élaborer un algorithme capable de tracer un chemin reliant deux points quelconques de la zone navigable, lorsque cela est possible.

3. Démarche

Étant donné que je connaissais déjà des algorithmes de recherche de chemin dans un graphe (Dijkstra, A*), je me suis demandé comment on pouvait les adapter dans le cas d'une zone polygonale du plan.

Un découpage convexe de notre zone navigable convient. J'ai donc cherché des algorithmes capables de réaliser un tel découpage.

Après avoir fait des recherches sur Internet, il m'a paru que la meilleure façon d'y parvenir était d'utiliser un algorithme de triangulation de polygones, c'est−à−dire un algorithme découpant un polygone en triangles. Pour cela, deux méthodes élémentaires et bien connues sont appliquables : méthode des oreilles et méthode des diagonales.

J'ai implémenté ces deux algorithmes, mais malheureusement, ils ne fonctionnent que sur des polygones simples, sans trous. Cela ne nous convient donc pas.

Mon contact à l'INRIA, Mme Yvinec, m'a alors orienté vers une méthode par balayage, afin de prendre en compte les trous. J'ai donc implémenté la méthode dite de décomposition en polygones monotones, qui donne la triangulation demandée.

J'ai ensuite implémenté plusieurs autres algorithmes : simplification convexe, recherche de chemin (algorithme A*), dessin de chemin.

Finalement, j'ai réalisé une interface graphique permettant l'utilisation interactive de mon programme, à l'aide de la bibliothèque graphique FOX. La production finale de mon TIPE est donc un programme graphique permettant le tracé d'une zone polygonale à trous et l'application des algorithmes de triangulation et de recherche de chemin.

J'ai écrit moi−même tous ces algorithmes dans le langage C++ que je maîtrise bien. Le projet totalise environ 3100 lignes de code.

4. Étude des résultats

J'ai appliqué mon programme sur un plan (simplifié) du quartier de mon lycée, ce qui m'a permis de déterminer un chemin pour y accéder depuis chez moi. J'ai pu constater que mon programme fonctionne sans difficulté sur des données de grosse taille.

Le chemin obtenu n'est généralement pas optimal, mais est largement satisfaisant. En terme de complexité, l'algorithme de décomposition en polygones monotones est un des meilleurs.

Je me suis également intéressé à un invariant de la triangulation d'un polygone : le nombre de triangles obtenus. J'ai déduit une formule empirique, que j'ai ensuite pu démontrer.

5. Références et contacts

Références

− Cours d'algorithmique, Sariel Har−Peled, http://valis.cs.uiuc.edu/~sariel/teach/2004/b/ (en particulier les chapitres 23 et 24)

− Documentation de la bibliothèque CGAL, http://www.cgal.org/

Découpage en polygones convexes pour la recherche de chemins

Alex AUVOLAT–BERNSTEIN

Table des matières

1 Étude générale du problème

1.1 Énoncé du problème

On se donne le plan d'un lieu connu, contenu dans un espace bi-dimensionnel borné. Ce plan est donné sous la forme d'un polygone donnant le contour du lieu, et de plusieurs polygones définissant des obstacles dans ce lieu. Le polygone contour privé des polygones obstacles donne un domaine que l'on appellera *zone navigable* (cf. figure 2).

On cherche à établir un algorithme capable de relier deux points quelconques de la zone navigable par un chemin (qui prendra la forme d'une ligne brisée) contenu dans cette zone.

1.2 Intérêt du découpage en parties convexes

Si on dispose d'une partie convexe $\mathcal{P}$ de l'espace, alors on sait que pour tous points $A, B \in \mathcal{P}$, le segment $[AB]$ est contenu dans $\mathcal{P}$.

L'idée est donc de découper notre zone navigable en parties convexes : si l'on dispose d'un découpage en polygones convexes et que l'on sait quels polygones partagent un côté, on peut déterminer une liste de polygones à parcourir pour relier deux points du plan. La détermination d'un chemin reliant les deux points est alors facile : il suffit de constituer la ligne brisée commençant au point de départ, passant par un point sur chaque côté partagé par deux polygones consécutifs à parcourir, puis terminant au point d'arrivée.

Dans notre vocabulaire, on appellera *tuiles* les polygones convexes de la décomposition obtenue.

1.3 Algorithmes possibles

Pour effectuer le découpage en polygones convexes de la zone navigable, on peut penser à un algorithme de triangulation (en effet, les triangles sont les "plus petits" polygones convexes).

On dispose de deux algorithmes classiques de triangulation de polygones : méthode des oreilles et méthode des diagonales. Malheureusement, ces deux méthodes ne s'appliquent que sur des polygones simples, c'est-à-dire "sans trous ", ce qui ne nous convient donc pas.

Une meilleure solution est donc d'utiliser un algorithme par balayage. Un tel algorithme permet dans un premier temps de décomposer la zone navigable en polygones monotones (définition donnée en 2.2.1, cf. figure 3), que l'on peut ensuite trianguler en temps linéaire (illustration en figure 4.)

La recherche de chemin se fait ensuite par un parcours de graphe non orienté type algorithme de Dijkstra, où les sommets du graphe sont les polygones de notre décomposition, et les arêtes sont les côtés partagés par deux polygones.

2 Triangulation de polygones

2.1 Triangulation de polygones simples

2.1.1 Généralités sur les polygones

On définit un polygone simple à partir d'une série $(p_0, p_1, \ldots, p_{n-1})$ de points du plan. Les côtés du polygone sont les segments $[p_i p_{i+1}]$ (on numérote les sommets modulo n). Le polygone peut être dit simple sous la condition que deux côtés du polygone ne s'intersectent jamais (sauf deux côtés consécutifs sur le sommet qu'ils partagent).

Tout polygone simple peut être triangulé. La démonstration de ce résultat vient des algorithmes constructifs donnés dans cette section.

Theorème 1. *Toute triangulation d'un polygone simple à n sommets comporte $n-2$ triangles.*

2.1.2 Méthode des oreilles

Définition 1. *On appelle* oreille *d'un polygone un sommet p_i tel que le triangle $p_{i-1}p_ip_{i+1}$ soit entièrement inclus dans le polygone.*

Remarque 1. *Une oreille est nécessairement un sommet convexe, c'est-à-dire que l'angle interne du polygone en ce sommet est inférieur ou égal à 180 degrés.*

Theorème 2. *Tout polygone simple ayant au moins 4 sommets comporte au moins deux oreilles non consécutives.*

Cette propriété se démontre assez facilement par récurrence sur le nombre de sommets du polygone.

Pour trianguler un polygone simple, il suffit donc de réduire notre polygone en lui enlevant une oreille à chaque itération, jusqu'à n'avoir plus que des triangles.

Avec un algorithme non optimisé, la recherche d'une oreille se fait en $O(n^2)$, et il y a $O(n)$ oreilles à enlever, ce qui fait un algorithme de complexité $O(n^3)$.

2.1.3 Méthode des diagonales

Dans cet algorithme, il s'agit de tracer des diagonales (ou cordes) dans notre polygone, c'est-à-dire des segments reliant deux sommets du polygone et appartenant entièrement au polygone. Le tracé d'une diagonale découpe le polygone en deux nouveaux polygones "plus petits" (avec moins de points) sur lesquels on peut appliquer l'algorithme récursivement.

La recherche d'une diagonale se fait selon l'algorithme suivant :

1. Choisir le sommet p_i avec l'ordonnée minimale.
2. Trouver, s'il existe, le sommet p_j contenu dans le triangle $p_{i-1}p_ip_{i+1}$ qui soit le plus proche de p_i dans la direction orthogonale à la droite $(p_{i-1}p_{i+1})$, avec $j \notin \{i-1, i, i+1\}$.
3. Si un tel sommet existe, alors le segment $[p_ip_j]$ est une diagonale.
4. Sinon, le segment $[p_{i-1}p_{i+1}]$ est une diagonale (on a une oreille en p_i).

La recherche des sommets p_i et p_j se fait en temps $O(n)$, et il y a $O(n)$ diagonales à tracer (le tracé d'une diagonale correspond à la division du polygone en deux, ce qui se fait en $O(n)$ opérations), ce qui donne un algorithme de complexité $O(n^2)$.

2.2 Triangulation d'un polygone à trous

Définition 2. *Un polygone à trous est défini comme étant le domaine constitué par un polygone contour privé d'un certain nombre de polygones (polygones obstacles) qui sont à l'intérieur de celui-ci.*

Dans notre problème, la zone navigable est constituée d'un ou plusieurs polygones à trous (dans le cas où il y en a plusieurs, ceux-ci forment chacun une composante connexe de la zone navigable).

Convention 1. *Les polygones contours doivent être orientés dans le sens direct. Les polygones obstacles doivent être orientés dans le sens indirect (cf. figure 2).*

Ainsi, lorsque l'on étudie une arête $[p_ip_{i+1}]$ de n'importe lequel des polygones servant à délimiter la zone navigable, l'intérieur de la zone navigable se trouve toujours à gauche de $[p_ip_{i+1}]$ (dans le sens $\overrightarrow{p_ip_{i+1}}$), et l'extérieur de la zone navigable se trouve toujours à droite.

2.2.1 Décomposition en polygones monotones

Définition 3. *On appelle* chaîne monotone *(selon l'axe vertical) une suite (finie) de points dont les ordonnées forment une suite monotone.*

Définition 4. *On appelle* polygone monotone *un polygone dont les côtés peuvent se décomposer en deux chaînes monotones (visuellement : une à gauche, une à droite ; cf. figure 3).*

L'algorithme de décomposition parcourt tous les sommets de bas en haut. À chaque sommet rencontré, on examine le type du sommet selon la position des deux côtés adjacents par rapport à la droite horizontale passant par le sommet étudié (appelée ligne de balayage). Les cinq types de sommets que l'on peut rencontrer sont illustrés en figure 5.

Remarquons ici qu'un polygone simple sans sommet "séparation" ni sommet "fusion" est déjà un polygone monotone.

Le but de l'algorithme est de construire des chaînes monotones qui délimitent des polygones monotones remplissant tout l'espace de la zone navigable. On construit donc ces chaînes de bas en haut, en prenant diverses actions selon le type du point rencontré. Les chaînes monotones construites fonctionnent par paires : en effet, un polygone monotone est constitué de deux chaînes monotones. Pour chaque chaîne monotone, l'intérieur de la zone navigable se trouve soit à gauche soit à droite. Lorsque l'on rajoute une

diagonale dans notre polygone pour préserver la monotonie, l'intérieur se trouve des deux côtés. Le segment rajouté appartiendra donc à deux chaînes monotones.

Ces chaînes monotones en cours de construction sont à chaque étape enregistrées dans une structure ordonnée, permettant de rechercher une chaîne ou d'en insérer une en temps logarithmique. L'ordre correspond à l'ordre géométrique des chaînes : elles sont rangées de gauche à droite.

Pour un sommet "départ", on commence deux nouvelles chaînes monotones, composée chacune d'une des deux arêtes adjacentes au sommet.

Pour un sommet "arrivée", on fusionne les deux chaînes monotones qui y arrivent, ce qui donne un des polygones de la décomposition.

Pour un sommet normal, il s'agit simplement de prolonger la chaîne monotone à laquelle il appartient en rajoutant le sommet qui est au-dessus.

Les sommets "séparation" et "fusion" sont plus problématiques : en effet, ce sont eux qui empêchent la monotonie du polygone.

Pour un sommet "séparation", il s'agirait de commencer deux nouvelles chaînes, mais l'intérieur du polygone est du mauvais côté. Il faut donc créer une nouvelle diagonale dans le polygone, reliant le sommet étudié à un sommet situé en-dessous de celui-ci pour permettre la monotonie, c'est-à-dire relier le point étudié à une des chaînes monotones à droite ou à gauche des chaînes que l'on souhaite insérer. Le tracé de cette diagonale correspond en fait dans l'algorithme à une modification des chaînes monotones concernées.

Pour un sommet "fusion", on doit également créer une diagonale, mais au-dessus du sommet problématique. Or, on ne sait pas comment vont évoluer les chaînes monotones au-dessus de ce sommet avant d'avoir parcouru les points suivants : on ne sait donc pas vers quel sommet tracer une diagonale. On marque alors les deux chaînes qui s'y rejoignent comme "problématiques" ("unresolved" dans le code de mon implémentation). Le problème se trouve résolu lors d'un test qui est par ailleurs effectué au moment du parcours de chaque sommet (quel que soit son type), qui vérifie la possibilité de tracer une diagonale depuis ce sommet vers un sommet marqué comme problématique, résolvant ainsi le problème. Au tracé d'une telle diagonale, un polygone monotone de la décomposition est créé.

Il est clair qu'un tel algorithme doit gérer un nombre important de cas différents, ce qui explique la longueur du programme (plus de 200 lignes pour mon implémentation, cf. annexe D).

Un exemple du processus d'une telle décomposition est illustré en figure 9 de l'annexe B.

L'algorithme de décomposition en polygones monotones a une complexité de $O(n\log(n))$. Il est donc meilleur que les deux algorithmes simples présentés en 2.1. Le facteur $\log(n)$ correspond à l'utilisation de la structure ordonnée qui mémorise les chaînes monotones en construction. Le facteur n correspond au parcours des n points des polygones définissant la zone navigable.

2.2.2 Triangulation d'un polygone monotone

La triangulation des polygones monotones pourrait se faire par les deux algorithmes simples présentés en 2.1. Mais ceux-ci ne sont pas optimaux : en effet, il existe des algorithmes exploitant les propriétés de monotonie du polygone pour effectuer cette triangulation en un temps linéaire.

La triangulation d'un polygone monotone se fait d'abord en découpant ce polygone en polygones dits "polygones montagnes" (cf. figure 6), puis en triangulant chaque montagne (cf. figure 7).

3 Recherche de chemin et optimisation du graphe

3.1 Une adaptation de l'algorithme de Dijkstra

L'algorithme de Dijkstra est un algorithme classique pour la recherche de chemins dans un graphe (orienté ou non), dans lequel les arêtes sont pondérées par une valeur représentant la distance entre les noeuds. Ici, on cherche simplement à savoir par quelle suite de tuiles passer pour aller de la tuile contenant le point de départ à la tuile contenant le point d'arrivée. Les tuiles deviennent donc les noeuds du graphe, et les bords partagés par deux tuiles en deviennent les arêtes.

La question a été de savoir quel poids attribuer aux arêtes du graphe. Une première possibilité était de donner pour poids la distance entre les barycentres des deux tuiles concernées.

L'approche que j'ai mise au point a été un peu plus subtile : en effet, pour chaque tuile T_i, en plus de mémoriser la tuile "parente", c'est-à-dire ayant permis d'atteindre T_i, on mémorise les coordonnées du point $I(T_i)$ par lequel T_i est atteinte (le point permettant d'atteindre la tuile contenant le point de départ est le point de départ). Ensuite, le poids de l'arête reliant cette tuile et une tuile suivante T_k du graphe est calculé comme étant la distance entre le point $I(T_i)$ et le point de $T_i \cap T_k$ le plus proche de $I(T_i)$, que

l'on notera I'. S'il est avantageux d'atteindre la tuile T_k par ce chemin, on met à jour les informations de la tuile T_k tel que $I(T_k) = I'$.

Cet algorithme ne garantit pas strictement l'optimalité de la longueur du chemin, mais en pratique on remarque qu'il donne de très bons résultats.

3.2 Simplification convexe

Le découpage en triangles de la zone navigable convient pour la recherche de chemins : en effet, il s'agit déjà d'une partition de la zone navigable en polygones convexes.

Mais puisque n'importe quel découpage en polygones convexes convient, on peut fusionner les triangles pour former des tuiles plus grosses afin d'améliorer la rapidité d'exécution de l'algorithme de recherche de chemins et pour améliorer la qualité des résultats (chemins plus simples et, en général, moins sinueux).

Pour cela, j'ai implémenté un algorithme très simple : on parcourt tous les triangles de la décomposition, et on regarde s'il existe parmi les paires de tuiles adjacentes une paire dont la fusion donnerait une nouvelle tuile convexe. Si c'est le cas, alors on peut fusionner les deux tuiles.

Le découpage obtenu à la sortie de cet algorithme est donc un découpage convexe minimal dans le sens où il ne reste plus aucune paire de tuiles pouvant être fusionnées en une tuile convexe plus grande. Mais l'algorithme ne garantit pas qu'il n'existe pas de regroupement qui soit composé d'un nombre inférieur de tuiles. On constate d'ailleurs sur le tableau de la figure 1 que le nombre de tuiles obtenues après simplification peut varier selon la triangulation sur laquelle on applique cet algorithme de regroupement.

Cet algorithme est illustré en figure 8.

4 Résultats

J'ai implémenté tous les algorithmes décrits ci-dessus en C++, langage que je maîtrise bien. J'ai également codé une interface graphique permettant l'utilisation interactive de ceux-ci, ce qui a permi de les tester de manière plus approfondie, sur un nombre important d'exemples.

Tous les algorithmes implémentés fonctionnent comme prévu dans les conditions pour lesquelles ils ont été conçus.

Ex	Polygones	Sommets Poly.	Total	Réflexe	Trous	Polys. Mono	Triangles	Tuiles
Ex1	1	12	12	4	0	3	10	5 à 6
Ex2	2	4,4	8	4	1	2	8	4
ExA	2	6,3	9	4	1	2	9	4
Ex9	2	4,5	9	4	1	3	9	5
Ex6	2	4,6	10	5	1	3	10	5
Ex7	2	8,4	12	6	1	2	12	6
Ex4	2	4,16	20	10	1	4	20	10 à 12
ExB	3	4,4,8	16	10	2	4	18	12
Ex5	9	4*	36	32	8	6	50	24 à 26
Ex8	16	11,5,5,4*	73	64	15	15	101	43 à 48

FIGURE 1: Tableau de résultats sur quelques exemples

On remarque que l'algorithme de simplification convexe permet généralement de diviser environ par deux le nombre de tuiles du découpage convexe. Je n'ai pas effectué de mesures de performance pour vérifier l'impact sur le calcul de chemins.

L'annexe C présente les résultats des algorithmes que j'ai implémentés appliqués à un exemple concret (ligne "ExB" dans le tableau de la figure 1).

5 Prolongements envisageables

5.1 Détection des intersections

Tel qu'il est écrit, mon algorithme de décomposition en polygones monotones ne fonctionne que lorsque les frontières des différents polygones sont disjointes deux à deux. Plus généralement, si pour chaque point du plan on calcule le nombre de polygones directs à l'intérieur desquels il se situe moins le nombre de polygones indirects à l'intérieur desquels il se situe, il faut, pour pouvoir calculer une décomposition en

polygones monotones, que tous les points arrivent à un total de 0 ou de 1 (0 signifiant alors que le point est à l'extérieur de la zone navigable, 1 signifiant qu'il est à l'intérieur).

Pour parfaire le programme, il faudrait donc un algorithme capable de détecter, parmi les polygones donnés en entrée, ceux qui s'intersectent. Par exemple si deux polygones directs s'intersectent, on peut les remplacer par leur union. Si un polygone direct intersecte un polygone indirect, on peut calculer le polygone délimité par le polygone direct privé du polygone indirect.

Au final, on pourrait implémenter un algorithme de réduction capable de prendre n'importe quel ensemble de polygones en entrée, et donnant en sortie un ensemble de polygones vérifiant la condition pour l'application de l'algorithme de décomposition en polygones monotones.

5.2 Autre algorithme de triangulation des polygones à trous

Mon contact à l'INRIA, Mme Mariette Yvinec, m'a suggéré de travailler sur un algorithme de triangulation basé sur une triangulation de l'ensemble des sommets des polygones, par exemple une triangulation de Delaunay, où l'on rajoute pour contrainte que les segments constituant les arêtes des polygones en entrée apparaissent dans la triangulation obtenue. On peut ensuite filtrer les triangles qui nous intéressent et ne garder que ceux qui définissent une partie de l'intérieur de la zone navigable.

Un tel algorithme résoudrait automatiquement le problème évoqué en 5.1.

Références

[1] Sariel Har-Peled, Cours d'algorithmique, `http://valis.cs.uiuc.edu/~sariel/teach/2004/b/` (en particulier les chapitres 23 et 24)

[2] Marc van Kreveld, Cours d'algorithmique géométrique, `http://www.cs.uu.nl/docs/vakken/ga/slides3.pdf` (en particulier les slides du chapitre 3)

[3] Rashid Bin Muhammad, Cours d'algorithmique géométrique, Kent University, `http://www.personal.kent.edu/~rmuhamma/Compgeometry/compgeom.html`

[4] Documentation de la bibliothèque CGAL, `http://www.cgal.org/`

[5] Heron Anzures, Animation Flash montrant le déroulement d'une triangulation d'après la méthode des polygones monotones, `http://computacion.cs.cinvestav.mx/~anzures/geom/triangulation.php`

[6] Description des algorithmes de parcours de graphe de Dijkstra et A^* sur Wikipédia, `http://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra` et `http://fr.wikipedia.org/wiki/Algorithme_A*`

[7] Documentation de référence du langage C++, `http://en.cppreference.com/w/`

[8] Documentation de référence de la librairie graphique FOX Toolkit, `http://www.fox-toolkit.org/`

A Illustrations

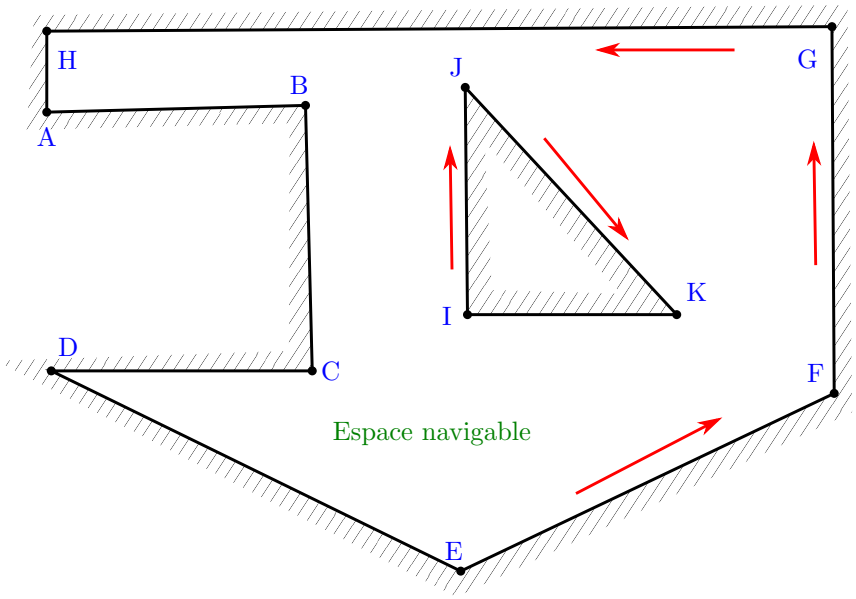

FIGURE 2: Convention : le polygone contour est orienté dans le sens direct, les polygones obstacles sont orientés dans le sens indirect.

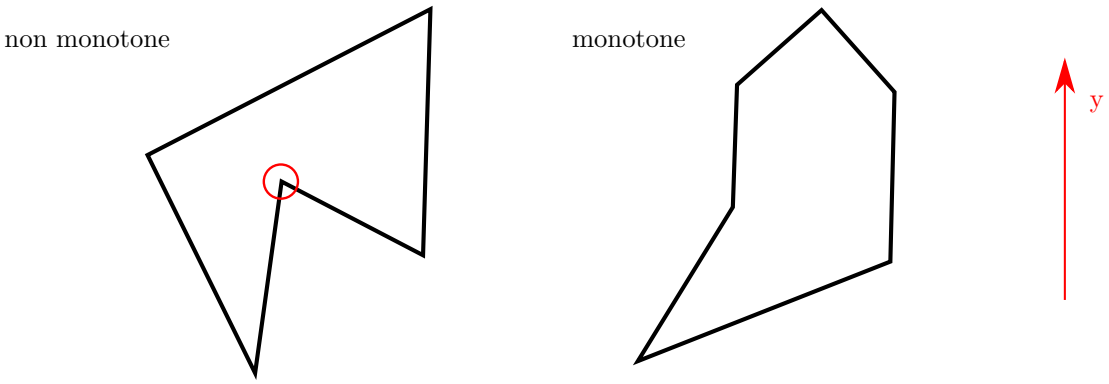

FIGURE 3: Polygone non monotone, polygone monotone

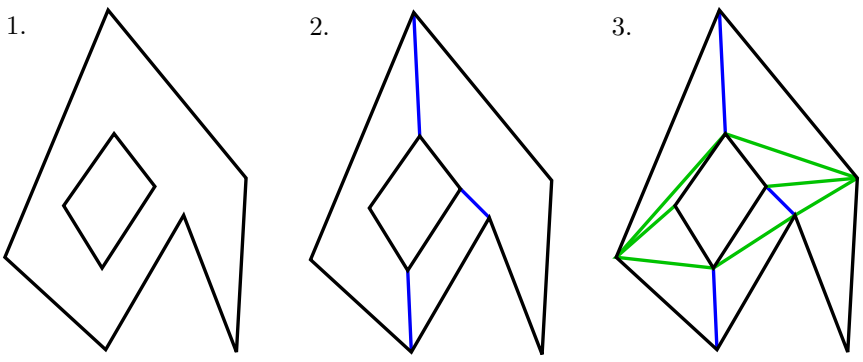

FIGURE 4: Les étapes de la triangulation passant par une décomposition en polygones monotones

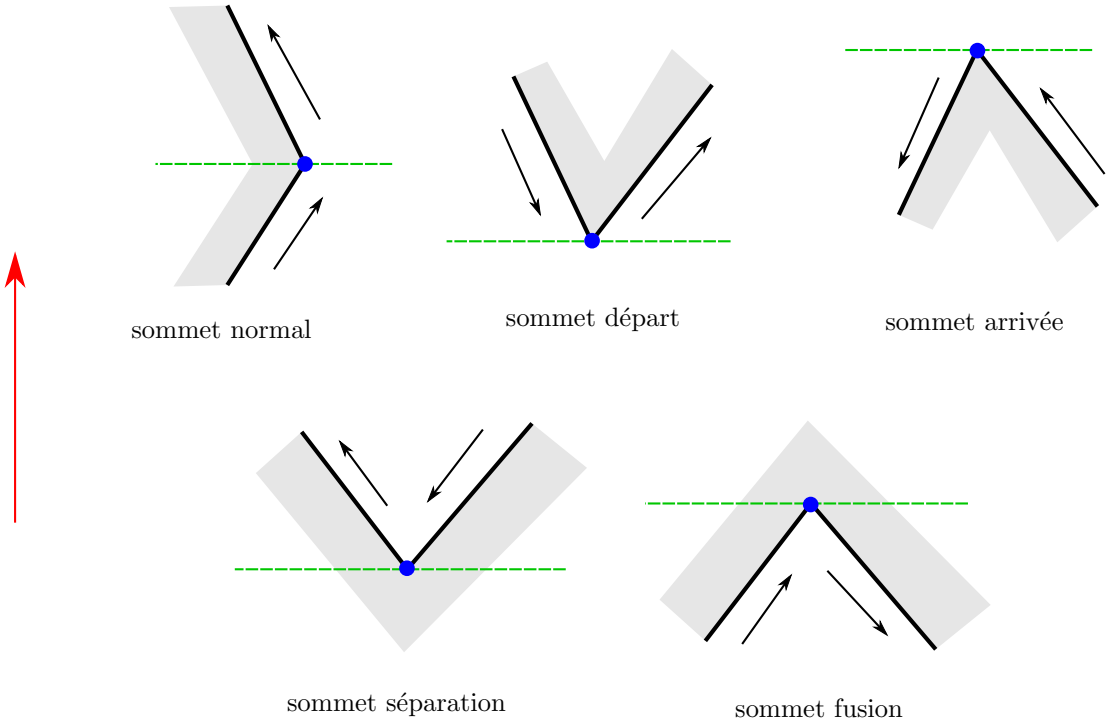

FIGURE 5: Types de sommets rencontrés lors de la décomposition en polygones monotones. L'intérieur du polygone (en gris) se situe toujours à gauche du contour pris dans le sens de parcours (sens dans lequel les points sont définis).

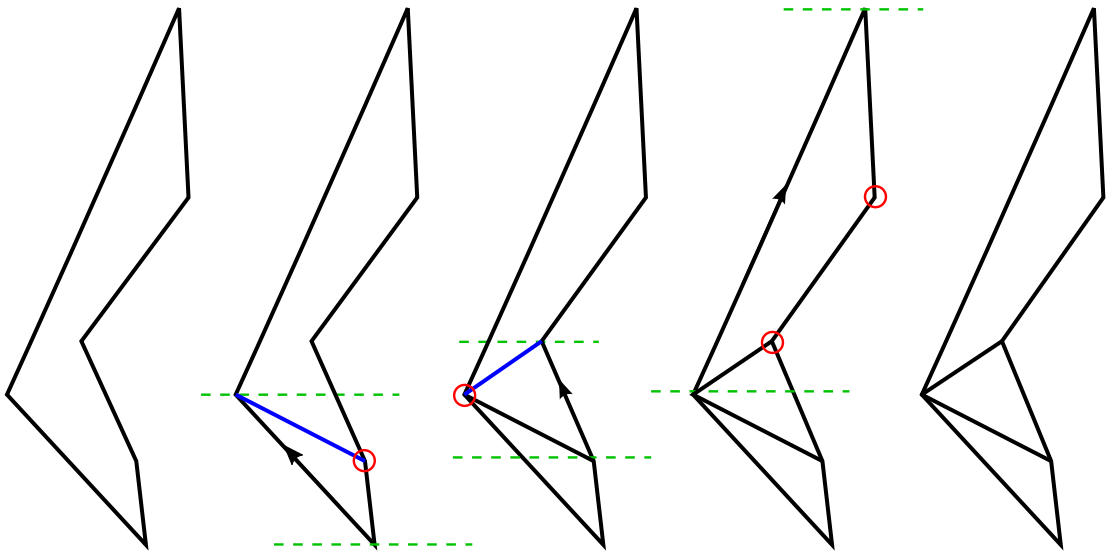

FIGURE 6: Décomposition d'un polygone monotone en polygones montagnes

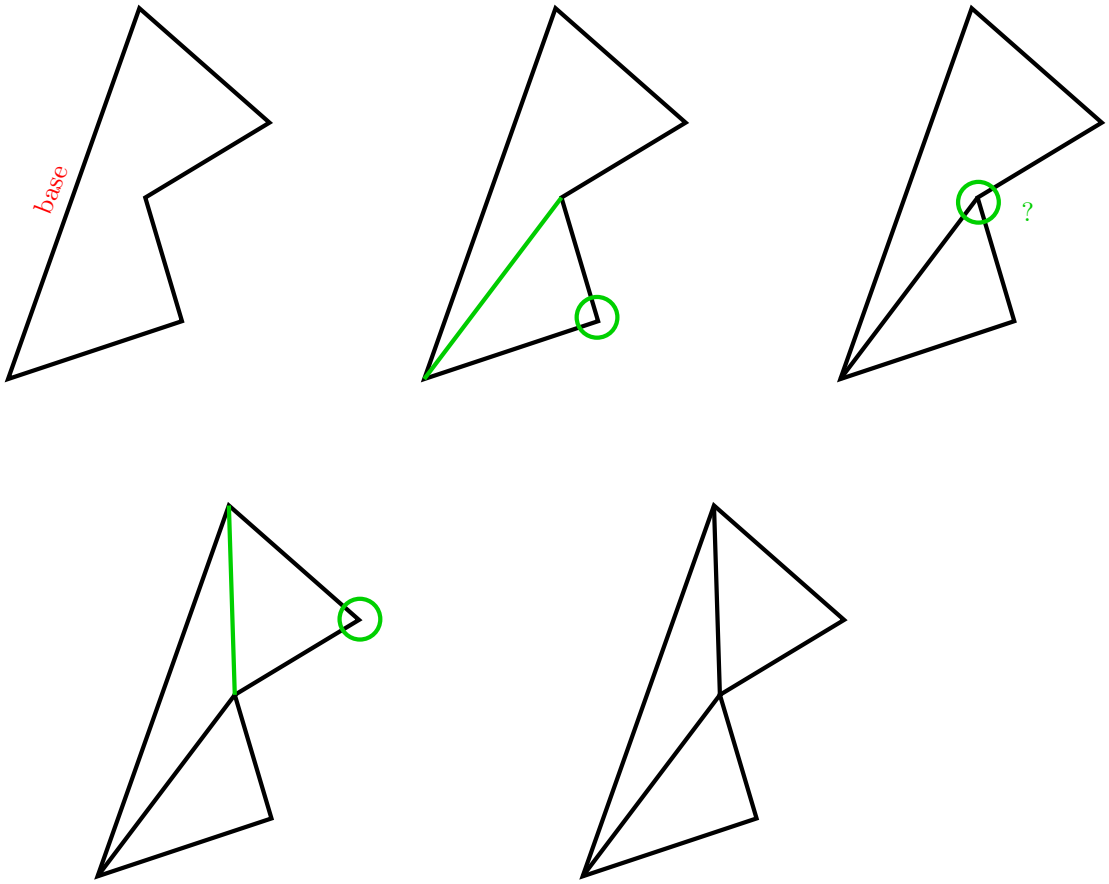

FIGURE 7: Triangulation d'un polygone montagne

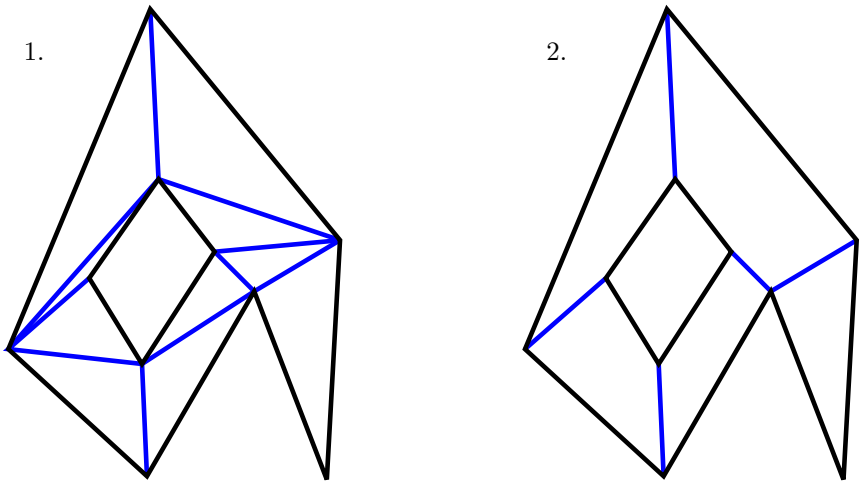

FIGURE 8: Simplification convexe d'une triangulation

B Exemple de déroulement d'une décomposition en polygones monotones

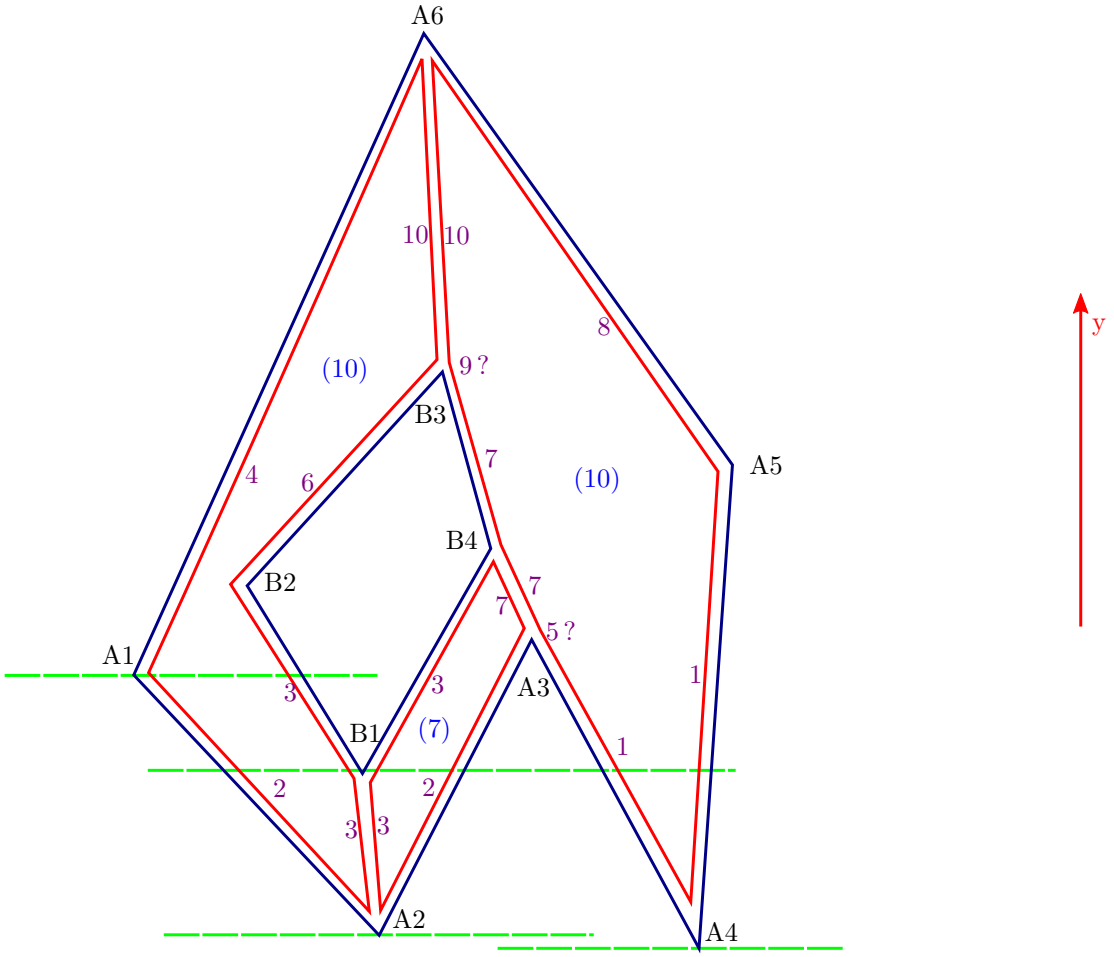

FIGURE 9: Illustration du déroulement d'une décomposition en polygones monotones.

Listing 1: Étapes de la décomposition en polygones monotones

```
1  1.  A4  (CUSP_A_IN)          A3A4   A5A4
2  2.  A2  (CUSP_A_IN)          A1A2   A3A2   A3A4   A5A4
3  3.  B1  (CUSP_A_OUT)         A1A2   <B2B1A2   B4B1A2   A3A2   A3A4   A5A4
4  4.  A1  (NORMAL_N)           >A6A1A2   B2B1A2   B4B1A2   A3A2   A3A4   A5A4
5  5.  A3  (CUSP_B_OUT)         A6A1A2   B2B1A2   B4B1A2   *<A3A2   *>A3A4   A5A4
6  6.  B2  (NORMAL_P)           A6A1A2   <B3B2B1A2   B4B1A2   *A3A2   *A3A4   A5A4
7  7.  B4  (NORMAL_N)           A6A1A2   B3B2B1A2   >B3B4B1A2   *A3A2   *A3A4   A5A4
8        >out(B4B1A2+B4A3A2)                    [B4B1A2A3]
9                               A6A1A2   B3B2B1A2   >B3B4A3A4   A5A4
10 8.  A5  (NORMAL_P)           A6A1A2   B3B2B1A2   B3B4A3A4   <A6A5A4
11 9.  B3  (CUSP_B_OUT)         A6A1A2   *<B3B2B1A2   *>B3B4A3A4   A6A5A4
12 10. A6  (CUSP_B_IN)          >A6A1A2   *B3B2B1A2   *B3B4A3A4   <A6A5A4
13       >out(A6A1A2+A6B3B2B1A2)                [A6A1A2B1B2B3]
14                              >A6B3B4A3A4   <A6A5A4
15       #out(A6B3B4A3A4+A6A5A4)                [A6B3B4A3A4A5*]
```

C Exemple complet d'application des algorithmes

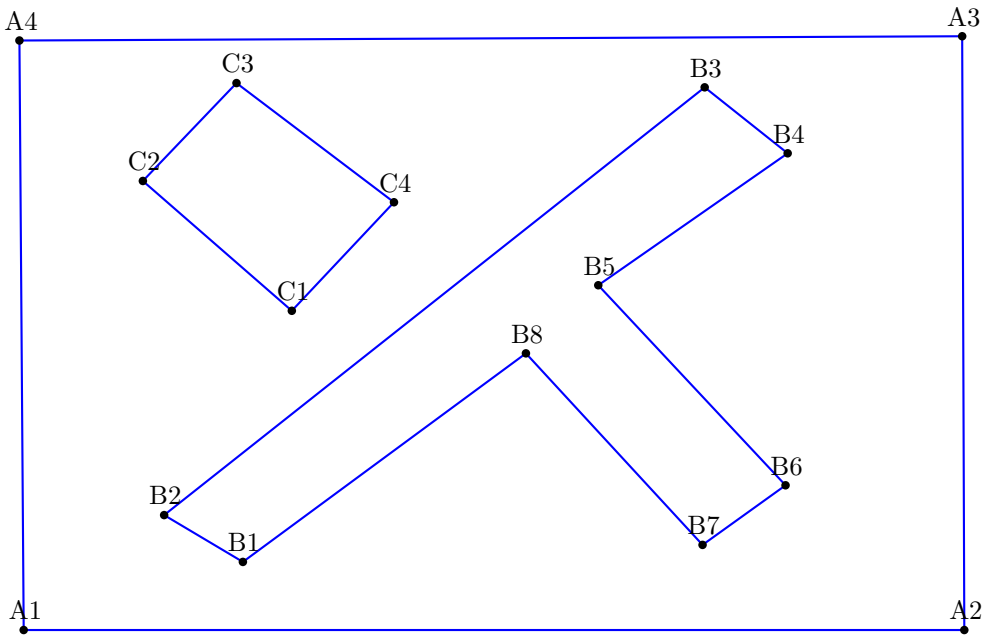

FIGURE 10: Les polygones donnés en entrée

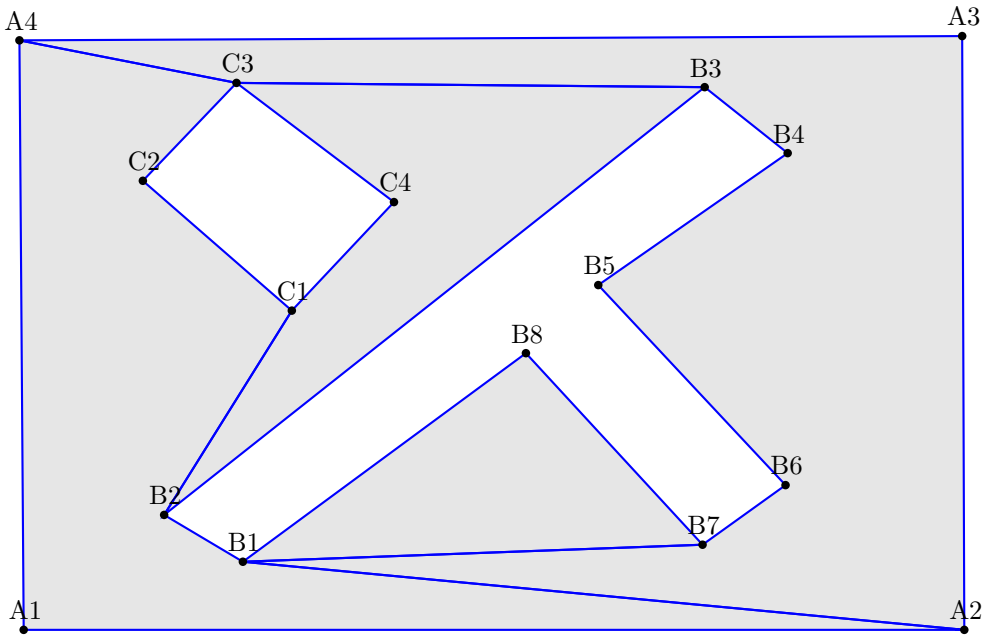

FIGURE 11: Décomposition en polygones monotones (4 polygones monotones)

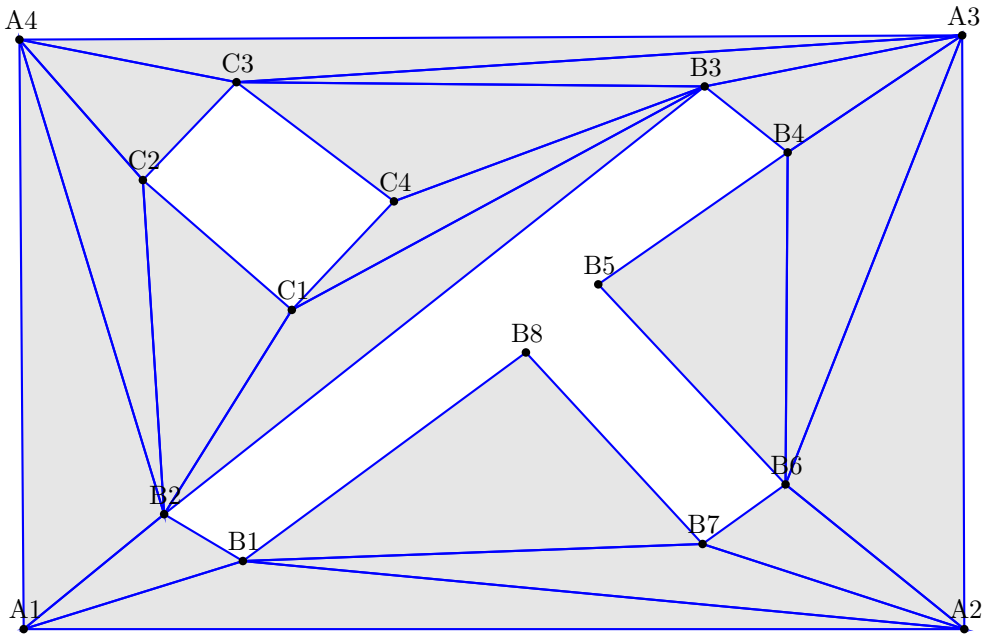

FIGURE 12: Triangulation à partir de la décomposition en polygones monotones (18 triangles)

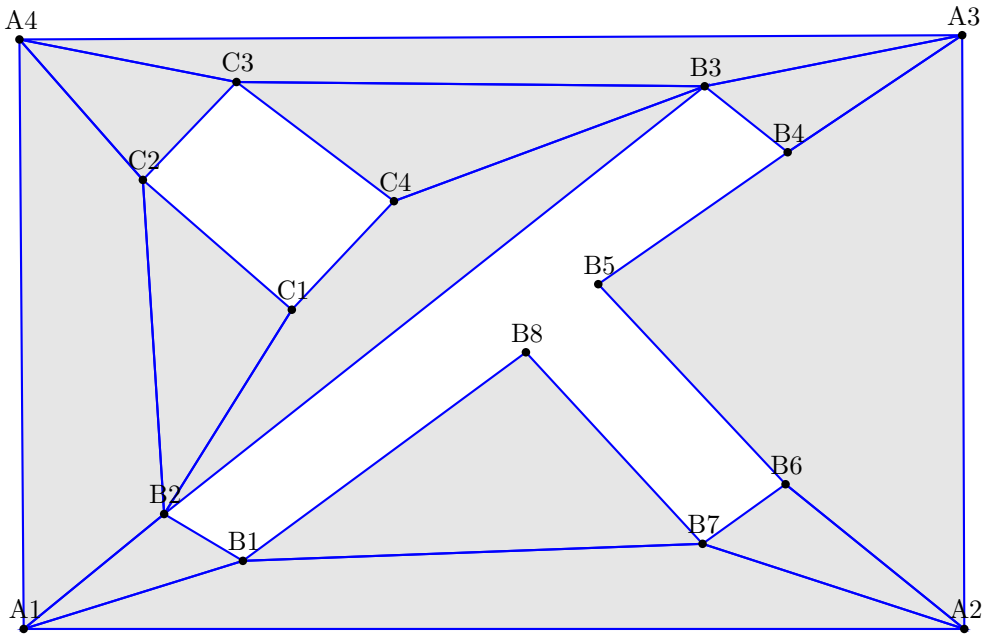

FIGURE 13: Simplification convexe de cette triangulation (12 tuiles)

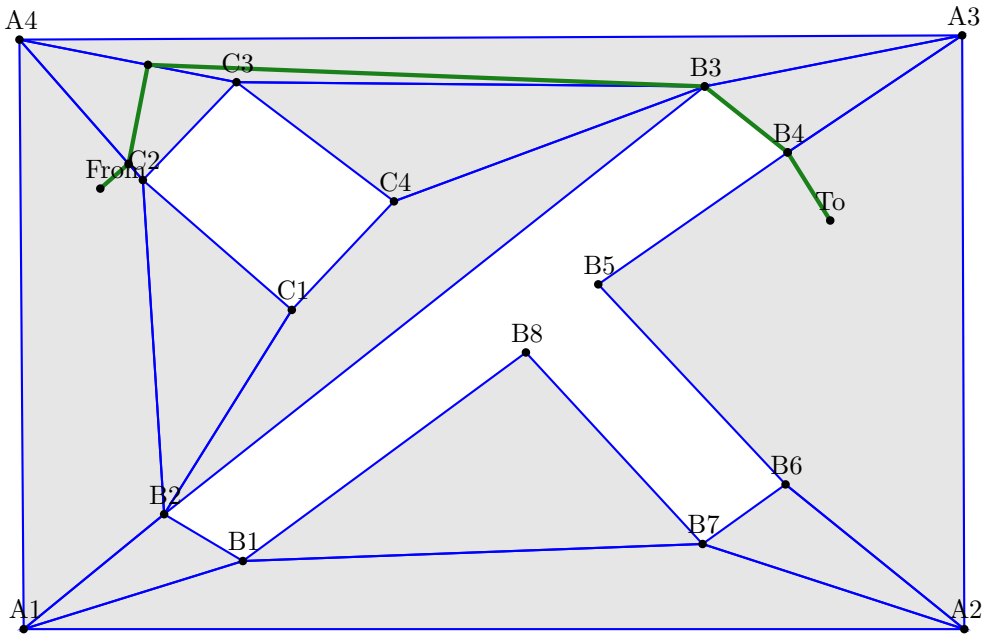

FIGURE 14: Exemple de recherche de chemin à partir de cette partition convexe

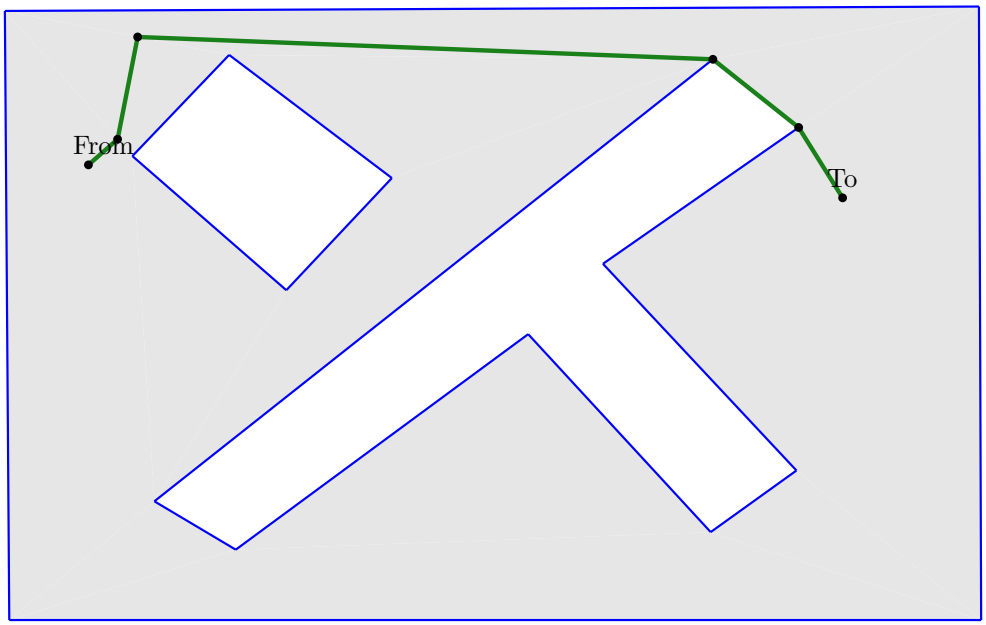

FIGURE 15: Le même chemin, sans affichage des tuiles du découpage

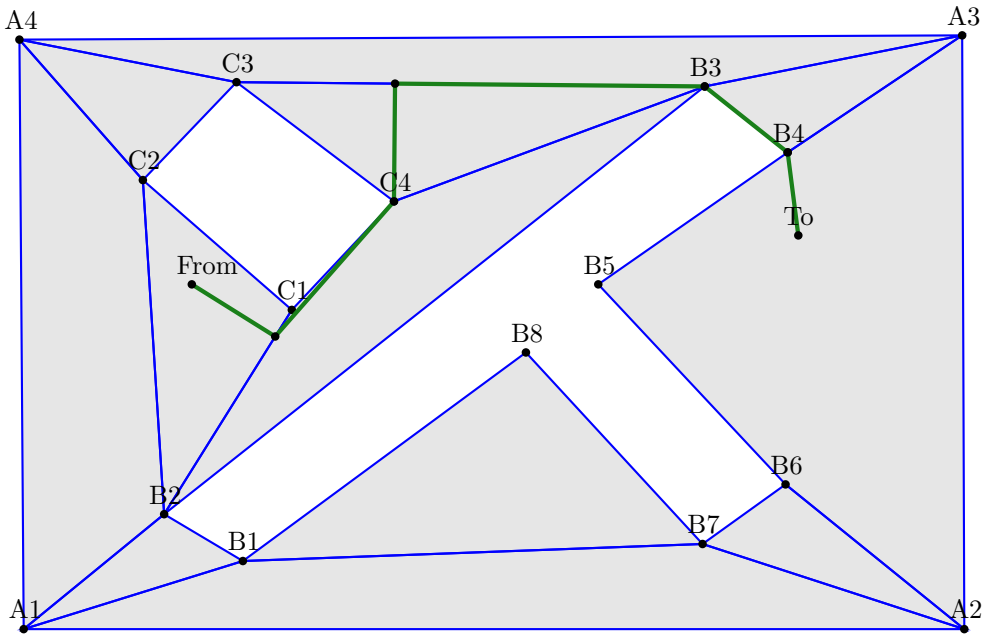

FIGURE 16: Second exemple de recherche de chemin

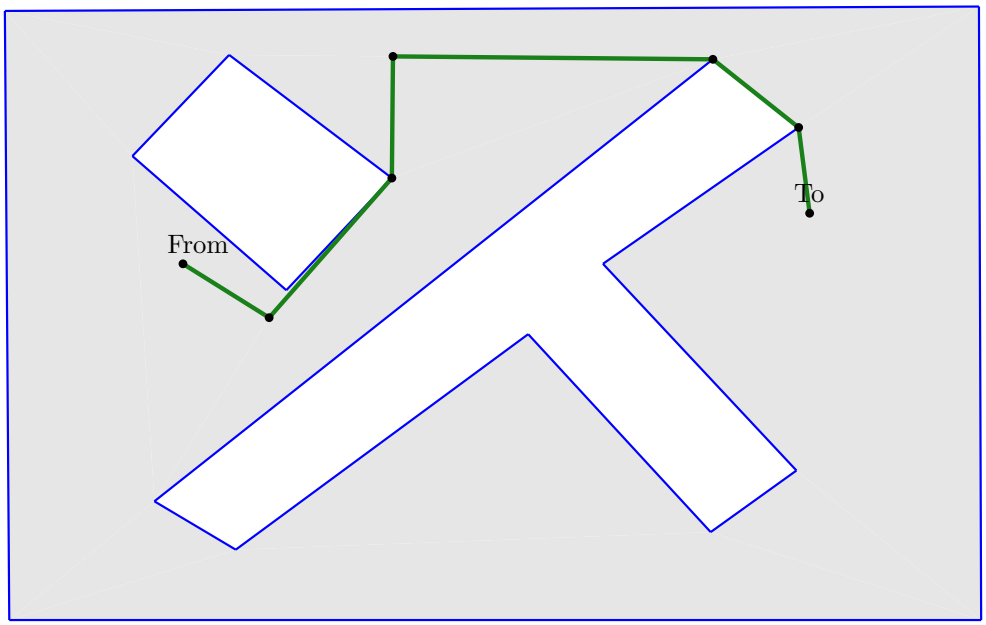

FIGURE 17: Le même chemin, sans affichage des tuiles du découpage

D Code source du programme

Listing 2: poly.h – main header file

```
1  #ifndef DEF_POLY_H
2  #define DEF_POLY_H
3
4  #include <string>
5  #include <list>
6  #include <vector>
7  #include <sstream>
8  #include <iostream>
9  #include <set>
10 #include <map>
11 #include <utility>
12
13 // ********** HELPERS
14
15 #define assert(cond, txt) { if (!(cond)) std::cerr << "Error in " << __FILE__ << "
       :" << __LINE__ << " : " << txt << std::endl; }
16
17 template<typename T>
18 std::string tostr(T i) {
19     std::ostringstream s;
20     s << i;
21     return s.str();
22 }
23
24 template<class ForwardIt>
25 ForwardIt next(ForwardIt it, typename std::iterator_traits<ForwardIt>::
       difference_type n = 1) {
26     std::advance(it, n);
27     return it;
28 }
29
30 template<class BidirIt>
31 BidirIt prev(BidirIt it, typename std::iterator_traits<BidirIt>::difference_type n
       = 1) {
32     std::advance(it, -n);
33     return it;
34 }
35
36 #define circ_next(l, it) (next(it) == l.end() ? l.begin() : next(it))
37 #define circ_prev(l, it) (it == l.begin() ? prev(l.end()) : prev(it))
38
39
40 // ********** POLYGONS
41
42 struct Vec;
43 struct Point {
44     float x, y;
45     std::string label;
46
47     Point() : x(0), y(0) {}
48     Point(float, float, std::string label = "");
49     Point(const Vec&, std::string label = "");
50
51     static Point* random(std::string label = "");
52     static Point* read(std::istream &in, float scale_x, float scale_y);
53
54     bool operator< (const Point& other);
55 };
56
57 struct PointXOrder {
```

```
58      bool operator() (Point* p1, Point* p2);
59  };
60
61  struct Line {           // line defined by ax + by + c = 0
62      float a, b, c;
63
64      Line(Point *a, Point *b);
65
66      bool same_side(Point* d, Point* e);
67      Vec normal_vec();       // returns a unit vector normal to line
68
69      float atY(float y);
70      Point orthog_projection(Point* p);
71      Point intersection(Line *l2);
72  };
73
74  struct Segment {
75      Point *a, *b;
76
77      Segment(Point *a, Point *b);
78
79      bool same_side(Point* d, Point* e);
80
81      Point middle(std::string label = "");
82      Point closest(Point* p);
83      Point closest(Point* p1, Point* p2);
84  };
85
86  struct Polygon;
87  struct Triangle {
88      Point *a, *b, *c;
89
90      Triangle (Point*, Point*, Point*);
91
92      bool is_inside(Point*);
93
94      Point barycenter(std::string label = "");
95      static std::list<Polygon*> simplify_triangle_partition(std::list<Triangle>
            triangles);
96  };
97
98  struct Polygon {
99      std::list<Point*> points;
100
101     Polygon() {}
102     Polygon(const std::list<Point*> &p);
103     Polygon(const std::vector<Point*> &p);
104     static Polygon* convex_hull(std::vector<Point*> points);
105     static Polygon* read(std::istream &in, float scale_x, float scale_y);
106     static Polygon* read(std::istream &in, std::map<std::string, Point*> &
            point_pool, float scale_x, float scale_y);
107
108     std::string name();
109     Point barycenter(std::string label = "");
110     float sum_of_angles();
111     bool is_direct();
112     bool is_convex();
113     Polygon* convex_hull();
114 };
115
116
117 // For simple math operations
118
119 struct Vertex;
```

```
120 struct Vec {
121     float x, y;
122     Vec(float, float);
123     Vec(Point *p);
124     Vec(Point *a, Point *b);
125     Vec(Vertex *a, Vertex *b);
126     Vec(Segment *s);
127     Vec(Line *l);       // unit vector in specified direction
128     bool is_nil() const { return x == 0 && y == 0; }
129     float norm() const;
130     Vec mul(float) const;
131     Vec unitvec() const;
132     static float cross(const Vec &a, const Vec &b);     // (fr) determinant
133     static float dot(const Vec &a, const Vec &b);       // (fr) produit scalaire
134     static float angle(const Vec &a, const Vec &b);
135 };
136
137
138 // Partition of a portion of the plane
139 // This structure is better than a list of polygons because it keeps track of
        shared edges between tiles
140
141 struct Edge;
142 struct Vertex {
143     Point* point;
144     std::map<Vertex*, Edge*> edgeTo;
145
146     Vertex(Point* p) : point(p) {}
147
148     Edge* edge_to(Vertex*);
149 };
150
151 struct Tile;
152 struct Edge {
153     Vertex *from, *to;
154     Edge *dual;         // the edge going the other way - always exists
155     Tile *left, *right;
156     Edge(Vertex* f, Vertex* t, Edge* d = NULL, Tile* l = NULL, Tile* r = NULL) :
157         from(f), to(t), dual(d), left(l), right(r) {}
158 };
159
160 struct Tile {       // all tiles are DIRECT!
161     std::list<Vertex*> points;
162
163     bool is_convex();
164     std::string name();
165
166     Polygon* make_poly();
167
168     // WARNING ! Only call Tile::split on a tile that is not registered in the
            partition
169     std::pair<Tile*, Tile*> split(std::list<Vertex*>::iterator v1, std::list<
            Vertex*>::iterator v2, bool check);
170     bool check();   // simply checks that we don't have twice the same vertice or
            same edge
171 };
172
173 struct Partition {
174     std::map<Point*, Vertex*> points;
175     std::set<Vertex*> vertices;
176     std::set<Tile*> tiles;
177
178     Partition();
179     Partition(Partition&);
```

```
180     void add(Polygon* poly);     // can only add direct (whole) polygons, holes not
            supported.
181
182     bool is_convex();        // true if all tiles are convex
183     Tile* find_tile_containing(Point* p);    // for path finding
184
185     // Mostly internal
186     bool add(Tile* t, bool check);  // checks if tile is correct and updates edge
            information acordingly
187     void remove(Tile* t);
188
189     Tile* combine_tiles(Edge* e, bool check);    // deletes edge and formes one
            tile from tiles left & right
190     std::pair<Tile*, Tile*> split_tile(Tile* t, std::list<Vertex*>::iterator v1,
            std::list<Vertex*>::iterator v2, bool check);
191
192     // Basic triangulations
193     Partition* earclip_triangulation(); // output = triangulation, which is a
            convex partition
194     void earclip_triangulation_inplace();    // same algorithm but doesn't create a
            new partition structure
195     void diagonal_triangulation();  // beter O(n^2) polygon triangulation
196
197     // First partition returned contains just the monotone polygons. Second
            partition is triangles.
198     static std::pair<Partition*, Partition*> monotone_triangulation(std::list<
            Polygon*> polygons);
199
200     // Convex partition simplification
201     bool delaunay_pass();    // true if some triangles were flipped. does nothing
            on shapes other than triangles.
202     void convex_simplify(); // simplifies a convex partition into a better convex
            partition (less tiles)
203 };
204
205
206 // PATH FINDING
207
208 struct Path {
209     Partition *partition;
210     Point *from, *to;
211     float margin;
212     bool found;
213
214     std::list<Edge*> edges;
215     std::list<Point*> points;
216
217     Path (Point* a, Point* b, Partition *p, float m) : partition(p), from(a), to(b
            ), margin(m), found(false) {}
218
219     static Path* find(Partition* partition, Point* from, Point* to, float margin =
            0);
220     void optimize();     // one iteration at a time
221
222     Path* clone();
223
224     float length();
225 };
226
227 #endif
```

Listing 3: util.cpp – utilities and helpers

```
1  /*
2      CONSTRUCTORS AND UTILITIES
3  */
4
5
6  #include "poly.h"
7
8  #include <stdlib.h>
9  #include <math.h>
10 #include <time.h>
11 #include <iostream>
12
13 using namespace std;
14
15 float frand(float min, float max) {
16     static bool i = false;
17     if (!i) {
18         srand(time(NULL)); i = true;
19     }
20     return ((float)rand() / (float)RAND_MAX) * (max - min) + min;
21 }
22
23 float frand_c(float min, float max, int f = 2) {
24     float acc = 0;
25     for (int i = 0; i < f; i++) {
26         acc += frand(min, max);
27     }
28     return acc / (float)f;
29 }
30
31 // ************ CONSTRUCTORS ***************
32
33 Point::Point(float xx, float yy, string lbl) : x(xx), y(yy), label(lbl) {
34 }
35
36 Point::Point(const Vec& v, string lbl) : x(v.x), y(v.y), label(lbl) {
37 }
38
39 Point* Point::random(string label) {
40     return new Point(frand_c(0, 1), frand_c(0, 1), label);
41 }
42
43 Point* Point::read(istream &in, float scale_x, float scale_y) {
44     float x, y;
45     string s;
46     in >> x >> y >> s;
47     return new Point(x * scale_x, y * scale_y, s);
48 }
49
50 bool PointXOrder::operator() (Point* p1, Point* p2) {
51     return p1->x < p2->x;
52 }
53
54 Line::Line(Point* p1, Point* p2) {
55     a = p1->y - p2->y;
56     b = p2->x - p1->x;
57     c = -(a * p1->x + b * p1->y);
58 }
59
60 Vec Line::normal_vec() {
61     return Vec(a, b).unitvec();
62 }
63
```

```
64 float Line::atY(float y) {
65     // ax + by + c = 0
66     // x = -c/a - by/a
67     return -(c + b * y) / a;
68 }
69
70 Segment::Segment(Point *p1, Point *p2) {
71     a = p1;
72     b = p2;
73 }
74
75 Triangle::Triangle (Point* aa, Point* bb, Point* cc) : a(aa), b(bb), c(cc) {}
76
77 Polygon::Polygon(const list<Point*> &p) : points(p) {}
78 Polygon::Polygon(const vector<Point*> &p) : points(p.begin(), p.end()) {}
79
80 Polygon* Polygon::read(istream &in, float sx, float sy) {
81     map<string, Point*> pts_map;
82     return read(in, pts_map, sx, sy);
83 }
84
85 Polygon* Polygon::read(istream &in, map<string, Point*> &pts_map, float sx, float
      sy) {
86     int n;
87     in >> n;
88     list<Point*> points;
89
90     for (int i = 0; i < n; i++) {
91         Point* p = Point::read(in, sx, sy);
92         if (pts_map.count(p->label) > 0) {
93             Point* p2 = pts_map[p->label];
94             assert(p->x == p2->x && p->y == p2->y,
95                 "Inconsistent input data (redefining point " + p->label + "). Old
                      definition is being kept.");
96             points.push_back(p2);
97             delete p;
98         } else {
99             points.push_back(p);
100            pts_map[p->label] = p;
101        }
102    }
103    return new Polygon(points);
104 }
105
106
107 Vec::Vec(float xx, float yy) : x(xx), y(yy) {}
108
109 Vec::Vec(Point *p) {
110     x = p->x;
111     y = p->y;
112 }
113
114 Vec::Vec(Point *a, Point *b) {
115     x = b->x - a->x;
116     y = b->y - a->y;
117 }
118
119 Vec::Vec(Vertex *a, Vertex *b) {
120     x = b->point->x - a->point->x;
121     y = b->point->y - a->point->y;
122 }
123
124 Vec::Vec(Segment *s) {
125     x = s->b->x - s->a->x;
```

```
126     y = s->b->y - s->a->y;
127 }
128
129 Vec::Vec(Line *l) {
130     x = 1;
131     if (l->a == 0) {
132         y = 0;
133     } else {
134         x = 1;
135         y = - l->b / l->a;
136         float l = sqrt(x*x + y*y);
137         x /= l; y /= l;
138     }
139 }
140
141 // *************** OPERATIONS WITH VECTORS
142
143 float Vec::norm() const {
144     return sqrt(x*x + y*y);
145 }
146
147 Vec Vec::mul(float r) const {
148     return Vec(x * r, y * r);
149 }
150
151 Vec Vec::unitvec() const {
152     return mul(1 / norm());
153 }
154
155 float Vec::cross(const Vec &a, const Vec &b) {
156     return a.x * b.y - a.y * b.x;
157 }
158
159 float Vec::dot(const Vec &a, const Vec &b) {
160     return a.x * b.x + a.y * b.y;
161 }
162
163 float Vec::angle(const Vec &a, const Vec &b) {
164     if (a.is_nil() || b.is_nil()) return 0;
165     float cos = dot(a.unitvec(), b.unitvec());
166     if (cos < -1) return 3.1415926;
167     float uangle = acos(cos);
168     if (cross(a, b) > 0) {
169         return uangle;
170     } else {
171         return -uangle;
172     }
173 }
```

Listing 4: poly.cpp – tools with vectors and polygons

```
1  /*
2      SIMPLE ALGORITHMS ON LINES, TRIANGLES, POLYGONS, ...
3  */
4
5  #include "poly.h"
6
7  #include <algorithm>
8
9  using namespace std;
10
11 bool Line::same_side(Point* d, Point* e) {
12     Point pa(0, -c/b, ""), pb(1, -(c+a)/b, "");
13     return Vec::cross(Vec(&pa, &pb), Vec(&pa, d)) * Vec::cross(Vec(&pa, &pb), Vec
           (&pa, e)) >= 0;
14 }
15
16 bool Segment::same_side(Point* c, Point* d) {
17     return Vec::cross(Vec(a, b), Vec(a, c)) * Vec::cross(Vec(a, b), Vec(a, d)) >=
           0;
18 }
19
20 bool Triangle::is_inside(Point* p) {
21     // Check P is on the same side of AB than C, on the same side of BC than A and
            on the same side of AC than B.
22     return Segment(a, b).same_side(c, p) && Segment(a, c).same_side(b, p) &&
           Segment(b, c).same_side(a, p);
23 }
24
25 Point Triangle::barycenter(std::string label) {
26     return Point((a->x + b->x + c->x) / 3, (a->y + b->y + c->y) / 3, label);
27 }
28
29 Point Polygon::barycenter(std::string label) {
30     if (points.size() == 0) return Point(0, 0, label);
31     float x = 0, y = 0;
32     for (list<Point*>::iterator it = points.begin(); it != points.end(); it++) {
33         x += (*it)->x;
34         y += (*it)->y;
35     }
36     return Point(x / points.size(), y / points.size(), label);
37 }
38
39 float Polygon::sum_of_angles() {
40     float sum = 0;
41     for (list<Point*>::iterator pt = points.begin(); pt != points.end(); pt++) {
42         if (*circ_prev(points, pt) == *circ_next(points, pt)) continue;
43         sum += Vec::angle(Vec(*circ_prev(points, pt), *pt), Vec(*pt, *circ_next(
               points, pt)));
44     }
45     return sum;
46 }
47
48 bool Polygon::is_direct() {
49     return sum_of_angles() >= 0;
50 }
51
52 bool Polygon::is_convex() {
53     if (points.size() < 4) return true;
54     float sign = 0;
55     for (list<Point*>::iterator pt = points.begin(); pt != points.end(); pt++) {
56         float s = Vec::cross(Vec(*circ_prev(points, pt), *pt), Vec(*pt, *circ_next
               (points, pt)));
57         if (sign == 0) {
```

```
58              sign = s;
59          } else {
60              if (sign * s < 0) return false;
61          }
62      }
63      return true;
64  }
65
66  /*
67      CONVEX HULL ALGORITHM
68  */
69
70  Polygon *Polygon::convex_hull(vector<Point*> points) {
71      list<Point*> result;
72
73      sort(points.begin(), points.end(), PointXOrder());
74
75      result.push_back(points[0]);
76
77      for (unsigned i = 1; i < points.size(); i++) {
78          // add to top list
79          while (result.size() > 2) {
80              Point *p1 = *(--result.end()), *p2 = *(-- --result.end());
81              if (Vec::cross(Vec(p1, p2), Vec(p1, points[i])) > 0) {
82                  result.pop_back();
83              } else {
84                  break;
85              }
86          }
87          result.push_back(points[i]);
88
89          // add do bottom list
90          while (result.size() > 2) {
91              Point *p1 = *(result.begin()), *p2 = *(++result.begin());
92              if (Vec::cross(Vec(p1, p2), Vec(p1, points[i])) < 0) {
93                  result.pop_front();
94              } else {
95                  break;
96              }
97          }
98          result.push_front(points[i]);
99      }
100
101     result.pop_back();  // the last point is in list twice
102
103     return new Polygon(result);
104 }
105
106 Polygon* Polygon::convex_hull() {
107     std::vector<Point*> pts(points.begin(), points.end());
108     return convex_hull(pts);
109 }
```

Listing 5: partition.cpp – tools with plane partitions

```
1  #include "poly.h"
2  #include <algorithm>
3
4  using namespace std;
5
6  Polygon* Tile::make_poly() {
7      list<Point*> pts;
8      for (list<Vertex*>::iterator it = points.begin(); it != points.end(); it++) {
9          pts.push_back((*it)->point);
10     }
11     return new Polygon(pts);
12 }
13
14 Edge* Vertex::edge_to(Vertex* v) {
15     if (edgeTo.count(v) != 0) {
16         return edgeTo[v];
17     } else {
18         Edge* e = edgeTo[v] = new Edge(this, v);
19         Edge* o = v->edgeTo[this] = new Edge(v, this, e);
20         e->dual = o;
21         return e;
22     }
23 }
24
25 Partition::Partition() {
26 }
27
28 Partition::Partition(Partition& other) {
29     // Duplicate a partition.
30     for (set<Tile*>::iterator it = other.tiles.begin(); it != other.tiles.end();
           it++) {
31         Tile* dup = new Tile();
32         for (list<Vertex*>::iterator t = (*it)->points.begin(); t != (*it)->points
               .end(); t++) {
33             Point* p = (*t)->point;
34             if (points.count(p) == 0) {
35                 points[p] = new Vertex(p);
36                 vertices.insert(points[p]);
37             }
38             dup->points.push_back(points[p]);
39         }
40         add(dup, false);
41     }
42 }
43
44 void Partition::add(Polygon* poly) {
45     float soa = poly->sum_of_angles();
46     if (soa >= 0) {
47         assert(soa > 6 && soa < 7,
48             "Incorrect input polygon " + poly->name() + " (sum of angles: " +
                   tostr(soa) + "), results not guaranteed.");
49         // ADD SHAPE TO TILING
50         // - add all points
51         // - create corresponding tile
52         Tile* t = new Tile();
53         for (list<Point*>::iterator pt = poly->points.begin(); pt != poly->points.
               end(); pt++) {
54             if (points.count(*pt) == 0) {
55                 points[*pt] = new Vertex(*pt);
56                 vertices.insert(points[*pt]);
57             }
58             t->points.push_back(points[*pt]);
59         }
```

```
60          // - add all edges
61          add(t, true);
62      } else {
63          assert(false, "Cannot add indirect polygon to partition.");
64      }
65  }
66
67  string Polygon::name() {
68      if (this == NULL) return "  ";
69      string ret ="";
70      for (list<Point*>::iterator it = points.begin(); it != points.end(); it++) {
71          ret += (*it)->label;
72      }
73      return ret;
74  }
75
76  string Tile::name() {
77      if (this == NULL) return "  ";
78      string ret = "";
79      for (list<Vertex*>::iterator it = points.begin(); it != points.end(); it++) {
80          ret += (*it)->point->label;
81      }
82      return ret;
83  }
84
85
86  // ********** CONVEX PARTITION ?
87
88  bool Tile::is_convex() {
89      // same as Polygon::is_convex();
90      if (points.size() < 4) return true;
91      float sign = 0;
92      for (list<Vertex*>::iterator pt = points.begin(); pt != points.end(); pt++) {
93          float s = Vec::cross(Vec(*circ_prev(points, pt), *pt), Vec(*pt, *circ_next
                (points, pt)));
94          if (sign == 0) {
95              sign = s;
96          } else {
97              if (sign * s < 0) return false;
98          }
99      }
100     return true;
101 }
102
103 bool Partition::is_convex() {
104     for (set<Tile*>::iterator tile = tiles.begin(); tile != tiles.end(); tile++) {
105         if (!(*tile)->is_convex()) return false;
106     }
107     return true;
108 }
109
110 // ************* SPLITTING & REUNITING TILES - INTERNAL
111
112 bool Tile::check() {
113     list<Vertex*>::iterator it = points.begin();
114     while (!points.empty() && it != points.end()) {
115         list<Vertex*>::iterator n = circ_next(points, it);
116         list<Vertex*>::iterator n2 = circ_next(points, n);
117         if (*it == *n) {
118             points.erase(n);
119             it = points.begin();
120         } else if (*it == *n2) {
121             points.erase(n);
122             points.erase(n2);
```

```
123             it = points.begin();
124         } else {
125             it++;
126         }
127     }
128     return points.size() >= 2;
129 }
130 
131 bool Partition::add(Tile* t, bool check) {
132     // make sure tile doesn't have strange things going on
133     if (check && !t->check()) return false;
134 
135     // update edge information
136     for (list<Vertex*>::iterator it = t->points.begin(); it != t->points.end(); it
        ++) {
137         Edge *e = (*it)->edge_to(*circ_next(t->points, it));
138         e->left = e->dual->right = t;
139     }
140     tiles.insert(t);
141     return true;
142 }
143 
144 void Partition::remove(Tile* t) {
145     tiles.erase(t);
146     for (list<Vertex*>::iterator it = t->points.begin(); it != t->points.end(); it
        ++) {
147         Vertex* nxt = *circ_next(t->points, it);
148         Edge* e = (*it)->edgeTo[nxt];
149         e->left = e->dual->right = NULL;
150         if (e->right == NULL) {
151             // delete edge
152             (*it)->edgeTo.erase(nxt);
153             (nxt)->edgeTo.erase(*it);
154             delete e->dual;
155             delete e;
156         }
157     }
158 }
159 
160 // SPLIT A TILE BY CREATING A SEGMENT BETWEEN V1 AND V2
161 pair<Tile*, Tile*> Tile::split(list<Vertex*>::iterator v1, list<Vertex*>::iterator
     v2, bool chk) {
162     assert(circ_next(points, v1) != v2 && circ_next(points, v2) != v1,
163         "Splitting a tile with two consecutive vertices.");
164 
165     // - create second tile && remove vertices from first tile
166     Tile *t2 = new Tile();
167     for (list<Vertex*>::iterator it = v1; it != v2; it = circ_next(points, it)) {
168         t2->points.push_back(*it);
169     }
170     t2->points.push_back(*v2);
171     while (circ_prev(points, v2) != v1) {
172         points.erase(circ_prev(points, v2));
173     }
174 
175     return pair<Tile*, Tile*>((!chk || check() ? this : NULL), (!chk || t2->check
        () ? t2 : NULL));
176 }
177 
178 pair<Tile*, Tile*> Partition::split_tile(Tile* t, list<Vertex*>::iterator v1, list
    <Vertex*>::iterator v2, bool check) {
179     assert(points[(*v1)->point] == *v1 && points[(*v2)->point] == *v2, "
        Inconsistent data.");
180 
```

```
181     // - remove tile edge information
182     remove(t);
183     // - split the tile
184     pair<Tile*, Tile*> tiles = t->split(v1, v2, check);
185     // - add it back
186     if (tiles.first != NULL) add(tiles.first, false);
187     if (tiles.second != NULL) add(tiles.second, false);
188
189     return tiles;
190 }
191
192 // COMBINE THE TWO TILES LEFT & RIGHT OF EDGE
193 Tile* Partition::combine_tiles(Edge* e, bool check) {
194     if (e->left == NULL || e->right == NULL) return NULL;
195
196     Tile *l = e->left, *r = e->right;
197     if (l == r) return l;
198     // tile r will be deleted and tile l will become the union of the two tiles.
199     Vertex *v1 = e->from, *v2 = e->to;
200
201     // - delete tiles
202     remove(l);
203     remove(r);
204     // - create new tile : fusion!
205     list<Vertex*>::iterator l2 = find(l->points.begin(), l->points.end(), v2);
206     assert(l2 != l->points.end(), "Cannot find vertex 2 in left tile's vertex list
            .");
207     list<Vertex*>::iterator r1 = find(r->points.begin(), r->points.end(), v1);
208     assert(r1 != r->points.end(), "Cannot find vertex 1 in right tile's vertex
            list.");
209     list<Vertex*>::iterator r2 = find(r->points.begin(), r->points.end(), v2);
210     assert(r2 != r->points.end(), "Cannot find vertex 2 in right tile's vertex
            list.");
211     for (list<Vertex*>::iterator it = circ_next(r->points, r1); it != r2; it =
            circ_next(r->points, it)) {
212         l->points.insert(l2, *it);
213     }
214     assert(add(l, check), "Tile fusion algorithm is bad.");
215     delete r;
216
217     return l;
218 }
```

Listing 6: basic_tri.cpp – basic triangulation algorithms

```
/*
    Basic triangulations :
    - Ear clipping
    - Diagonals
*/

#include "poly.h"

#include <algorithm>
#include <deque>
#include <cmath>

using namespace std;


/*
    EAR-CLIPPING METHOD FOR POLYGON TRIANGULATION
*/

Partition* Partition::earclip_triangulation() {
    Partition* ret = new Partition();

    for (set<Tile*>::iterator tile = tiles.begin(); tile != tiles.end(); tile++) {
        list<Point*> remaining;
        for (list<Vertex*>::iterator pt = (*tile)->points.begin(); pt != (*tile)->
            points.end(); pt++)
            remaining.push_back((*pt)->point);

        while (remaining.size() > 3) {
            Triangle tri(NULL, NULL, NULL);
            // Find a point for triangulation
            list<Point*>::iterator pt = remaining.begin(), prevpt, nextpt;
            bool ok = false;
            for (; pt != remaining.end(); pt++) {
                prevpt = circ_prev(remaining, pt);
                nextpt = circ_next(remaining, pt);

                // check that vertex is convex
                if (Vec::cross(Vec(*prevpt, *pt), Vec(*prevpt, *nextpt)) < 0)
                    continue;
                ok = true;

                // check that no other point is in triangle
                tri.a = *prevpt;
                tri.b = *pt;
                tri.c = *nextpt;
                for (list<Point*>::iterator it2 = circ_next(remaining, nextpt);
                    it2 != prevpt; it2 = circ_next(remaining, it2)) {
                    if (tri.is_inside(*it2) && *it2 != tri.a && *it2 != tri.b && *
                        it2 != tri.c) {
                        ok = false;
                        break;
                    }
                }

                if (ok) break;
            }
            assert(ok, "No triangle in polygon.");

            // Remove it!
            remaining.erase(pt);
            if (*circ_prev(remaining, prevpt) == *nextpt || *circ_next(remaining,
                nextpt) == *prevpt) {
```

```
59                  // handles the special case where the triangle goes back on itself
60                  // simply removing point pt would leave an segment [prevpt, nexpt
                        ], with no usefull inside
61                  remaining.erase(prevpt);
62                  remaining.erase(nextpt);
63              }
64              // and create corresponding triangle
65              list<Point*> poly;
66              poly.push_back(tri.a);
67              poly.push_back(tri.b);
68              poly.push_back(tri.c);
69              Polygon *p = new Polygon(poly);
70              ret->add(p);
71              delete p;
72          }
73
74          if (remaining.size() == 3) {
75              Polygon *p = new Polygon(remaining);
76              ret->add(p);
77              delete p;
78          }
79      }
80
81      return ret;
82  }
83
84  // SAME AS ABOVE BUT DOES NOT CREATE A NEW PARTITION STRUCTURE
85  // SEE COMMENTS ABOVE.
86  void Partition::earclip_triangulation_inplace() {
87      deque<Tile*> remaining;
88      while (!tiles.empty()) {
89          set<Tile*>::iterator it = tiles.begin();
90          remaining.push_back(*it);
91          remove(*it);
92      }
93
94      while (!remaining.empty()) {
95          Tile* t = remaining.front();
96          remaining.pop_front();
97
98          if (t->points.size() < 3) { // drop tile
99              delete t;
100             continue;
101         }
102         if (t->points.size() == 3) {    // it's a triangle !
103             add(t, false);
104             continue;
105         }
106
107         // Find an ear ( O(n^2) )
108         list<Vertex*>::iterator pt = t->points.begin(), prevpt, nextpt;
109         Triangle tri(NULL, NULL, NULL);
110         bool ok = false;
111         for (; pt != t->points.end(); pt++) {
112             prevpt = circ_prev(t->points, pt);
113             nextpt = circ_next(t->points, pt);
114
115             if (Vec::cross(Vec(*prevpt, *pt), Vec(*prevpt, *nextpt)) < 0) continue
                    ;
116             ok = true;
117
118             tri.a = (*prevpt)->point;
119             tri.b = (*pt)->point;
120             tri.c=  (*nextpt)->point;
```

```
121
122             for (list<Vertex*>::iterator it = circ_next(t->points, nextpt); it !=
                    prevpt; it = circ_next(t->points, it)) {
123                 Point* p = (*it)->point;
124                 if (tri.is_inside(p) && p != tri.a && p != tri.b && p != tri.c) {
125                     ok = false;
126                     break;
127                 }
128             }
129             if (ok) break;
130         }
131         assert(ok, "No triangle in polygon.");
132
133         // Cut it off
134         pair<Tile*, Tile*> p = t->split(prevpt, nextpt, true);
135         if (p.first != NULL) remaining.push_back(p.first);
136         if (p.second != NULL) remaining.push_back(p.second);
137     }
138 }
139
140 /*
141     OTHER METHOD FOR POLYGON TRIANGULATION, BY FINDING DIAGONALS AND SPLITTING THE
142     POLYGON ALONG THEM - TAKES O(n^2) TIME
143 */
144
145 void Partition::diagonal_triangulation() {
146     deque<Tile*> todo;
147     while (!tiles.empty()) {
148         set<Tile*>::iterator it = tiles.begin();
149         todo.push_back(*it);
150         remove(*it);
151     }
152
153     while (!todo.empty()) {
154         Tile* t = todo.front(); todo.pop_front();
155
156         if (t->points.size() < 3) {
157             delete t;
158             continue;
159         }
160         if (t->points.size() == 3) {    // it's a triangle - keep it.
161             add(t, false);
162             continue;
163         }
164
165         // Find a diagonal
166         // - find lower point of polygon
167         list<Vertex*>::iterator lower = t->points.begin();
168         for (list<Vertex*>::iterator it = next(t->points.begin()); it != t->points
                .end(); it++) {
169             if ((*it)->point->y < (*lower)->point->y) lower = it;
170         }
171         // - find nearest point for a diagonal
172         list<Vertex*>::iterator other = t->points.end();    // maybee we will find
                none
173
174         list<Vertex*>::iterator before = circ_prev(t->points, lower), after =
                circ_next(t->points, lower);
175         assert((*before)->point != (*after)->point, "Malformed polygon " + (t->
                name()));
176
177         Line l((*before)->point, (*after)->point);
178         Triangle tri((*before)->point, (*lower)->point, (*after)->point);
179         Vec v = l.normal_vec();
```

```
180         float d = fabs(Vec::dot(Vec(*lower, *after), v));
181         float d1 = fabs(Vec::dot(Vec(*lower, *before), v));
182
183         for (list<Vertex*>::iterator it = circ_next(t->points, after); it !=
                before; it = circ_next(t->points, it)) {
184             if (!tri.is_inside((*it)->point)) continue;
185             float dd = fabs(Vec::dot(Vec(*lower, *it), v));
186             if ((dd <= d || dd <= d1) && *it != *before && *it != *after && *it !=
                     *lower) {
187                 d = d1 = dd;
188                 other = it;
189             }
190         }
191
192         // Split
193         pair<Tile*, Tile*> p;
194         if (other == t->points.end()) {
195             p = t->split(before, after, true);
196         } else {
197             p = t->split(lower, other, true);
198         }
199         if (p.first != NULL) todo.push_back(p.first);
200         if (p.second != NULL) todo.push_back(p.second);
201     }
202 }
```

Listing 7: monotone_tri.cpp – monotone triangulation algorithm

```
1  #include "poly.h"
2  #include <algorithm>
3  #include <deque>
4
5  // If true, print log of monotone decomposition and triangulation to stderr
6  #define LOG_ENABLE false
7
8  using namespace std;
9
10 /*
11     ALGORITHM FOR TRIANGULATION USING MONOTONE POLYGONS & MONOTONE CHAINS
12 */
13
14 enum p_point_type {
15     P_UNDEFINED = 0,
16     P_NORMAL_P = 1,
17     P_NORMAL_N = 2,
18     P_CUSP_B_IN = 3,
19     P_CUSP_B_OUT = 4,
20     P_CUSP_A_IN = 5,
21     P_CUSP_A_OUT = 6
22 };
23
24 struct p_point {        // custom circular list
25     Point* p;
26     p_point *prev, *next;
27     p_point(Point* pp, p_point* pr, p_point* pn) : p(pp), prev(pr), next(pn) {}
28     p_point_type type;
29 };
30
31 struct p_point_compare {
32     bool operator() (p_point* a, p_point* b) {
33         return a->p->y < b->p->y || (a->p->y == b->p->y && a->p->x < b->p->x);
34     }
35 };
36
37 struct monotone_chain {
38     list<Point*> points;
39     bool unresolved;
40
41     monotone_chain(Point* a, Point* b) {
42         unresolved = false;
43         add(a); add(b);
44     }
45
46     void add(Point *p) {
47         points.push_front(p);
48     }
49     void rm() {
50         points.pop_front();
51     }
52     Point* first() {
53         return *points.begin();
54     }
55     Point* second() {
56         return *next(points.begin());
57     }
58     Point* last() {
59         return *points.rbegin();
60     }
61     string str() const {
62         string ret = "";
63         for (list<Point*>::const_iterator i = points.begin(); i != points.end(); i
```

```
            ++) {
64              ret += (*i)->label;
65          }
66          return ret;
67      }
68  };
69
70  struct m_c_compare {
71      bool do_cmp(monotone_chain *a, monotone_chain *b) const {
72          if (a->first()->y < b->last()->y) {
73              return a->first()->x < b->last()->x;
74          }
75          if (a->last()->y > b->first()->y) {
76              return a->last()->x < b->first()->x;
77          }
78
79
80          list<Point*>::iterator ita = a->points.begin(), itb = b->points.begin();
81          if (*ita == *itb) {
82              while (*ita == *itb) {
83                  ita++;
84                  itb++;
85                  if (ita == a->points.end() || itb == b->points.end()) return false
                        ; // are equal
86              }
87              return Vec::cross(Vec(*prev(ita), *ita), Vec(*prev(itb), *itb)) > 0;
88          }
89
90          float y = min((*ita)->y, (*itb)->y), lasty = max(a->last()->y, b->last()->
                y);
91          while (y >= lasty) {
92              if (y == (*ita)->y && y == (*itb)->y) {
93                  float xamin = (*ita)->x, xamax = (*ita)->x, xbmin = (*itb)->x,
                        xbmax = (*itb)->x;
94                  while (true) {
95                      ita++;
96                      if (ita == a->points.end() || (*ita)->y != y) break;
97                      if ((*ita)->x > xamax) xamax = (*ita)->x;
98                      if ((*ita)->x < xamin) xamin = (*ita)->x;
99                  }
100                 while(true) {
101                     itb++;
102                     if (itb == b->points.end() || (*itb)->y != y) break;
103                     if ((*itb)->x > xbmax) xbmax = (*itb)->x;
104                     if ((*itb)->x < xbmin) xbmin = (*itb)->x;
105                 }
106                 if (xamax < xbmin) return true;
107                 if (xbmax < xamin) return false;
108             } else if (y == (*ita)->y) {
109                 while ((*next(itb))->y > y) itb++;
110                 float xb = Line(*itb, *next(itb)).atY(y);
111                 while (ita != a->points.end() && (*ita)->y == y) {
112                     if ((*ita)->x < xb) return true;
113                     if (xb < (*ita)->x) return false;
114                     ita++;
115                 }
116                 itb++;
117             } else {
118                 while ((*next(ita))->y > y) ita++;
119                 float xa = Line(*ita, *next(ita)).atY(y);
120                 while (itb != b->points.end() && (*itb)->y == y) {
121                     if (xa < (*itb)->x) return true;
122                     if ((*itb)->x < xa) return false;
123                     itb++;
```

```
124                 }
125                 ita++;
126             }
127             if (ita == a->points.end() || itb == b->points.end()) break;
128             y = min((*ita)->y, (*itb)->y);
129         }
130
131         // chains probably equal
132         return false;
133     }
134
135     bool operator() (monotone_chain *a, monotone_chain *b) const {
136         bool c = do_cmp(a, b);
137         return c;
138     }
139 };
140
141 pair<Partition*, Partition*> Partition::monotone_triangulation(list<Polygon*>
        polygons) {
142     const char *ptt[] = { "UNDEFINED", "NORMAL_P", "NORMAL_N", "CUSP_B_IN", "
            CUSP_B_OUT", "CUSP_A_IN", "CUSP_A_OUT" };
143
144     set<p_point*, p_point_compare> events;
145
146     // Add all points
147     for (list<Polygon*>::iterator it = polygons.begin(); it != polygons.end(); it
            ++) {
148         p_point *p = NULL;
149         for (list<Point*>::iterator pit = (*it)->points.begin(); pit != (*it)->
                points.end(); pit++) {
150             if (p == NULL) {
151                 p = new p_point(*pit, NULL, NULL);
152                 p->prev = p->next = p;
153             } else {
154                 p = new p_point(*pit, p, p->next);
155                 p->next->prev = p;
156                 p->prev->next = p;
157             }
158             p->type = P_UNDEFINED;
159             events.insert(p);
160         }
161     }
162     for (set<p_point*, p_point_compare>::iterator it = events.begin(); it !=
            events.end(); it++) {
163         p_point *p = *it;
164
165         if (p->prev->p->y == p->p->y) {
166             if (p->next->p->y == p->p->y) {
167                 p->type = (p->prev->p->x < p->p->x ? P_NORMAL_P : P_NORMAL_N);
168             } else {
169                 if (p->prev->p->x < p->p->x) {
170                     p->type = (p->next->p->y < p->p->y ? P_CUSP_B_OUT : P_NORMAL_P
                            );
171                 } else {
172                     p->type = (p->next->p->y > p->p->y ? P_CUSP_A_OUT : P_NORMAL_N
                            );
173                 }
174             }
175         } else if (p->next->p->y == p->p->y) {
176             if (p->next->p->x > p->p->x) {
177                 p->type = (p->prev->p->y > p->p->y ? P_CUSP_A_IN : P_NORMAL_P );
178             } else {
179                 p->type = (p->prev->p->y < p->p->y ? P_CUSP_B_IN : P_NORMAL_N);
180             }
```

```
181         } else if (p->prev->p->y < p->p->y && p->next->p->y < p->p->y) {
182             if (Vec::cross(Vec(p->p, p->prev->p), Vec(p->p, p->next->p)) >= 0) {
183                 p->type = P_CUSP_B_OUT;
184             } else {
185                 p->type = P_CUSP_B_IN;
186             }
187         } else if (p->prev->p->y > p->p->y && p->next->p->y > p->p->y) {
188             if (Vec::cross(Vec(p->p, p->prev->p), Vec(p->p, p->next->p)) >= 0) {
189                 p->type = P_CUSP_A_OUT;
190             } else {
191                 p->type = P_CUSP_A_IN;
192             }
193         } else if (p->prev->p->y < p->p->y) {
194             p->type = P_NORMAL_P;
195         } else {
196             p->type = P_NORMAL_N;
197         }
198         if (LOG_ENABLE) {
199             cerr << "point " << p->p->label << " (" << p->prev->p->label << "->"
                    << p->next->p->label
200                 << ") : " << ptt[p->type] << endl;
201         }
202     }
203
204     // Create monotone chains && output polygons
205     list< pair<monotone_chain*, monotone_chain*> > output;
206     set<monotone_chain*, m_c_compare> chains;
207
208     for (set<p_point*, p_point_compare>::iterator eit = events.begin(); eit !=
            events.end(); eit++) {
209         p_point *p = *eit;
210
211         if (LOG_ENABLE) {
212             cerr << "point " << p->p->label << " (" << ptt[p->type] << ") -> ";
213         }
214
215         set<monotone_chain*, m_c_compare>::iterator ck_unr_before = chains.end(),
                ck_unr_after = chains.end();
216         bool must_output = false;
217
218         if (p->type == P_NORMAL_P) {
219             monotone_chain key(p->prev->p, p->p);
220             set<monotone_chain*, m_c_compare>::iterator c = chains.lower_bound(&
                    key);
221             assert(c != chains.end(), "P_NORMAL_P Points not ordered correctly ("
                    + key.str() + ").");
222             monotone_chain *ch = *c;
223             chains.erase(c);
224             ch->add(p->next->p);
225             ck_unr_before = chains.insert(ch).first;
226         } else if (p->type == P_NORMAL_N) {
227             monotone_chain key(p->next->p, p->p);
228             set<monotone_chain*, m_c_compare>::iterator c = chains.lower_bound(&
                    key);
229             assert(c != chains.end(), "P_NORMAL_N Points not ordered correctly ("
                    + key.str() + ").");
230             monotone_chain *ch = *c;
231             chains.erase(c);
232             ch->add(p->prev->p);
233             ck_unr_after = chains.insert(ch).first;
234         } else if (p->type == P_CUSP_A_IN) {
235             monotone_chain *ch = new monotone_chain(p->p, p->next->p);
236             chains.insert(ch);
237             ch = new monotone_chain(p->p, p->prev->p);
```

```
238             chains.insert(ch);
239         } else if (p->type == P_CUSP_B_IN) {
240             monotone_chain k1(p->prev->p, p->p), k2(p->next->p, p->p);
241             set<monotone_chain*, m_c_compare>::iterator it1 = chains.find(&k1),
                    it2 = chains.find(&k2);
242             assert(it1 != chains.end(), "P_CUSP_B_IN K1 Points not ordered
                    correctly (" + k1.str() + ").");
243             assert(it2 != chains.end(), "P_CUSP_B_IN K2 Points not ordered
                    correctly (" + k2.str() + ").");
244             ck_unr_before = it1;
245             ck_unr_after = it2;
246             must_output = true;
247         } else if (p->type == P_CUSP_A_OUT) {
248             monotone_chain k1(p->p, p->next->p), k2(p->p, p->prev->p);
249             set<monotone_chain*, m_c_compare>::iterator it = prev(chains.
                    lower_bound(&k1));
250             if ((*it)->unresolved) {
251                 monotone_chain *ch2 = *it, *ch1 = *prev(it);
252                 chains.erase(prev(it));
253                 chains.erase(it);
254                 ch2->add(p->p);
255                 ch2->add(p->prev->p);
256                 ch1->add(p->p);
257                 ch1->add(p->next->p);
258                 ch2->unresolved = false;
259                 ch1->unresolved = false;
260                 chains.insert(ch1);
261                 chains.insert(ch2);
262             } else if ((*next(it))->unresolved) {
263                 monotone_chain *ch2 = *next(next(it)), *ch1 = *next(it);
264                 chains.erase(next(next(it)));
265                 chains.erase(next(it));
266                 ch2->add(p->p);
267                 ch2->add(p->prev->p);
268                 ch1->add(p->p);
269                 ch1->add(p->next->p);
270                 ch2->unresolved = false;
271                 ch1->unresolved = false;
272                 chains.insert(ch1);
273                 chains.insert(ch2);
274             } else {
275                 monotone_chain *ch = *it, *nch = *next(it);
276                 if (ch->second()->y > nch->second()->y) {
277                     if (LOG_ENABLE) {
278                         cerr << " -- {{" << k1.str() << "-" << k2.str() << " -> "
                                << ch->str() << "}} => ";
279                     }
280                     chains.erase(it);
281 
282                     monotone_chain *ch2 = new monotone_chain(ch->second(), ch->
                            first());
283                     monotone_chain *ch3 = new monotone_chain(ch->second(), p->p);
284                     ch3->add(p->next->p);
285                     chains.insert(ch2);
286                     chains.insert(ch3);
287 
288                     ch->rm();
289                     ch->add(p->p);
290                     ch->add(p->prev->p);
291                     ck_unr_after = chains.insert(ch).first;
292                 } else {
293                     it++; ch = nch;
294                     if (LOG_ENABLE) {
295                         cerr << " -- {{" << k1.str() << "-" << k2.str() << " -> "
```

```
                        << ch->str() << "}} => ";
296                 }
297                 chains.erase(it);
298
299                 monotone_chain *chl = new monotone_chain(ch->second(), ch->
                        first());
300                 monotone_chain *chm = new monotone_chain(ch->second(), p->p);
301                 chm->add(p->prev->p);
302                 chains.insert(chl);
303                 chains.insert(chm);
304
305                 ch->rm();
306                 ch->add(p->p);
307                 ch->add(p->next->p);
308                 ck_unr_before = chains.insert(ch).first;
309             }
310         }
311     } else if (p->type == P_CUSP_B_OUT) {
312         monotone_chain k1(p->prev->p, p->p), k2(p->next->p, p->p);
313         set<monotone_chain*, m_c_compare>::iterator it1 = chains.find(&k1),
                it2 = chains.find(&k2);
314         assert(it1 != chains.end(), "P_CUSP_B_IN K1 Points not ordered
                correctly (" + k1.str() + ").");
315         assert(it2 != chains.end(), "P_CUSP_B_IN K2 Points not ordered
                correctly (" + k2.str() + ").");
316         (*it1)->unresolved = true;
317         (*it2)->unresolved = true;
318         ck_unr_before = it1;
319         ck_unr_after = it2;
320     }
321
322     if (LOG_ENABLE) {
323         for (set<monotone_chain*, m_c_compare>::iterator i = chains.begin(); i
                != chains.end(); i++) {
324             cerr << ((*i)->unresolved ? "*" : "") <<
325                     (ck_unr_before == i ? "<" : "") <<
326                     (ck_unr_after == i ? ">" : "") << (*i)->str() << "  ";
327         }
328         cerr << endl;
329     }
330
331     // resolve
332     while (next(ck_unr_after) != chains.end() && (*next(ck_unr_after))->
            unresolved &&
333         next(next(ck_unr_after)) != chains.end() && (*next(next(ck_unr_after))
                )->unresolved) {
334         set<monotone_chain*, m_c_compare>::iterator it = next(ck_unr_after),
                it2 = next(next(ck_unr_after));
335         monotone_chain *c1 = *ck_unr_after, *c2a = *it, *c2b = *it2;
336         chains.erase(ck_unr_after); chains.erase(it); chains.erase(it2);
337         if (c1->second()->y >= c2a->first()->y) {
338             c2a->add(c1->second());
339             c2b->add(c1->second());
340             c2b->add(c1->first());
341             c1->rm();
342         } else {
343             c2a->add(c1->first());
344             c2b->add(c1->first());
345         }
346
347         output.push_back(pair<monotone_chain*, monotone_chain*>(c1, c2a));
348         if (LOG_ENABLE) {
349             cerr << " >out(" << c1->str() << "+" << c2a->str() << ") ";
350         }
```

```
351
352             c2b->unresolved = c1->unresolved;
353             ck_unr_after = chains.insert(c2b).first;
354         }
355         while (ck_unr_before != chains.begin() && (*prev(ck_unr_before))->
                unresolved &&
356             prev(ck_unr_before) != chains.begin() && (*prev(prev(ck_unr_before)))
                    ->unresolved) {
357             set<monotone_chain*, m_c_compare>::iterator it = prev(ck_unr_before),
                    it2 = prev(prev(ck_unr_before));
358             monotone_chain *c1a = *it2, *c1b = *it, *c2 = *ck_unr_before;
359             chains.erase(ck_unr_before); chains.erase(it); chains.erase(it2);
360             if (c2->second()->y >= c1a->first()->y) {
361                 c1b->add(c2->second());
362                 c1a->add(c2->second());
363                 c1a->add(c2->first());
364                 c2->rm();
365             } else {
366                 c1a->add(c2->first());
367                 c1b->add(c2->first());
368             }
369
370             output.push_back(pair<monotone_chain*, monotone_chain*>(c1b, c2));
371             if (LOG_ENABLE) {
372                 cerr << " <out(" << c1b->str() << "+" << c2->str() << ") ";
373             }
374
375             c1a->unresolved = c2->unresolved;
376             ck_unr_before = chains.insert(c1a).first;
377         }
378
379         if (must_output) {
380             output.push_back(pair<monotone_chain*, monotone_chain*>(*ck_unr_after,
                    *ck_unr_before));
381             if (LOG_ENABLE) {
382                 cerr << " #out(" << (*ck_unr_after)->str() << "+" << (*
                        ck_unr_before)->str() << ") ";
383             }
384             chains.erase(ck_unr_after);
385             chains.erase(ck_unr_before);
386         }
387
388         if (LOG_ENABLE) {
389             cerr << "\t\t=>  ";
390             for (set<monotone_chain*, m_c_compare>::iterator i = chains.begin(); i
                    != chains.end(); i++) {
391                 cerr << ((*i)->unresolved ? "*" : "") <<
392                         (ck_unr_before == i ? "<" : "") <<
393                         (ck_unr_after == i ? ">" : "") << (*i)->str() << "  ";
394             }
395             cerr << endl;
396         }
397     }
398
399     // Triangulate each monotone chain
400     Partition* monotone = new Partition();
401     Partition* tri = new Partition();
402     for (list< pair<monotone_chain*, monotone_chain*> >::iterator it = output.
            begin(); it != output.end(); it++) {
403         list<Point*> a(it->first->points), b(it->second->points);
404
405         if ((*next(a.begin()))->y >= (*next(b.begin()))->y) {
406             a.pop_front();
407         } else {
```

```
408             b.pop_front();
409         }
410
411         if ((*next(a.rbegin()))->y <= (*next(b.rbegin()))->y && a.size() > 1) {
412             a.pop_back();
413         } else {
414             b.pop_back();
415         }
416
417         Polygon *m = new Polygon(a);
418         m->points.insert(m->points.end(), b.rbegin(), b.rend());
419         assert(m->is_direct(), "Monotone polygon is not direct - monotone chains
                inverted?");
420         monotone->add(m);
421
422         while (a.size() > 1 || b.size() > 1) {
423             if (LOG_ENABLE) {
424                 cerr << " * got chains: ";
425                 for (list<Point*>::iterator it = a.begin(); it != a.end(); it++)
426                     cerr << (*it)->label;
427                 cerr << " and ";
428                 for (list<Point*>::iterator it = b.begin(); it != b.end(); it++)
429                     cerr << (*it)->label;
430             }
431
432             list<Point*> mountain;
433             if (a.front()->y >= b.front()->y && a.size() >= 2) {
434                 mountain.push_front(a.front());
435                 a.pop_front();
436                 while (b.front()->y >= a.front()->y) {
437                     mountain.push_front(b.front());
438                     b.pop_front();
439                 }
440                 b.push_front(mountain.front());
441                 mountain.push_front(a.front());
442             } else {
443                 mountain.push_back(b.front());
444                 b.pop_front();
445                 while (a.front()->y >= b.front()->y) {
446                     mountain.push_back(a.front());
447                     a.pop_front();
448                 }
449                 a.push_front(mountain.back());
450                 mountain.push_back(b.front());
451             }
452             if (LOG_ENABLE) {
453                 cerr << ", forming mountain: ";
454                 for (list<Point*>::iterator it = mountain.begin(); it != mountain.
                        end(); it++)
455                     cerr << (*it)->label;
456                 cerr << endl;
457             }
458
459             if (mountain.size() == 3) {
460                 // good, it's already a triangle
461                 Polygon *p = new Polygon(mountain);
462                 assert(p->is_direct(), "Mountain triangle not direct.");
463                 tri->add(p);
464             } else {
465                 // triangulate.
466                 deque<list<Point*>::iterator> todo;
467                 for (list<Point*>::iterator it = next(mountain.begin()); next(it)
                        != mountain.end(); it++) {
468                     if (Vec::cross(Vec(*prev(it), *it), Vec(*it, *next(it))) > 0)
```

```
                        {
469                         todo.push_back(it);
470                     }
471                 }
472                 while (!todo.empty() && mountain.size() > 3) {
473                     list<Point*>::iterator it = todo.front();
474                     todo.pop_front();
475                     list<Point*>::iterator pr = prev(it), nx = next(it);
476 
477                     bool ca_pr = (pr != mountain.begin() && Vec::cross(Vec(*prev(
                            pr), *pr), Vec(*pr, *it)) <= 0);
478                     bool ca_nx = (nx != prev(mountain.end()) && Vec::cross(Vec(*it
                            , *nx), Vec(*nx, *next(nx))) <= 0);
479 
480                     Polygon *p = new Polygon();
481                     p->points.push_back(*pr);
482                     p->points.push_back(*it);
483                     p->points.push_back(*nx);
484                     assert(p->is_direct(), "Cut off piece of mountain but is not
                            direct.");
485                     tri->add(p);
486 
487                     mountain.erase(it);
488 
489                     if (ca_pr && Vec::cross(Vec(*prev(pr), *pr), Vec(*pr, *nx)) >
                            0) todo.push_back(pr);
490                     if (ca_nx && Vec::cross(Vec(*pr, *nx), Vec(*nx, *next(nx))) >
                            0) todo.push_back(nx);
491                 }
492                 assert(mountain.size() == 3, "Remaining piece of mountain is not a
                        triangle.");
493                 Polygon *p = new Polygon(mountain);
494                 assert(p->is_direct(), "Remaining piece of mountain is not direct.
                        ");
495                 tri->add(p);
496             }
497         }
498     }
499 
500     return pair<Partition*, Partition*>(monotone, tri);
501 }
```

Listing 8: convex_part_simplify.cpp – convex simplification algorithm

```
1  /*
2      Algorithms for (naive) simplificaion of convex partitions
3  */
4
5  #include "poly.h"
6
7  #include <algorithm>
8  #include <deque>
9
10 using namespace std;
11
12
13 /*
14     "Delaunay triangulation"-like triangle flipping pass.
15     This function analyzes every pair of triangles and flips them if they
16     can be better balanced.
17     This has nothing to do with a real delaunay triangulation.
18     Returns true if a pair of triangles was flipped.
19 */
20
21 bool Partition::delaunay_pass() {
22     bool ret = false;
23
24     deque<Tile*> remaining(tiles.begin(), tiles.end());
25     while (!remaining.empty()) {
26         Tile* l = remaining.front();
27         remaining.pop_front();
28         if (l->points.size() != 3) continue;
29
30         for (list<Vertex*>::iterator pt = l->points.begin(); pt != l->points.end()
               ; pt++) {
31             Edge *e = (*pt)->edgeTo[*circ_next(l->points, pt)];
32             assert(e->left == l, "Invalid edge information.");
33             Tile* r = e->right;
34             if (r == NULL || r->points.size() != 3) continue;
35
36             Vertex *a = *circ_prev(l->points, pt);
37             Vertex *b = e->from;
38             Vertex *c = e->to;
39             Vertex *d = *circ_next(r->points, find(r->points.begin(), r->points.
                   end(), b));
40             if (Vec::cross(Vec(a, b), Vec(b, d)) <= 0) continue;
41             if (Vec::cross(Vec(d, c), Vec(c, a)) <= 0) continue;
42             float angle = Vec::angle(Vec(a, b), Vec(a, c)) + Vec::angle(Vec(d, c),
                    Vec(d, b));
43             if (angle <= 3.141592) continue;
44
45             // FLIP!
46             ret = true;
47             // - delete
48             remove(l);
49             remove(r);
50             // - reconstruct triangles
51             l->points.clear();
52             l->points.push_back(a);
53             l->points.push_back(b);
54             l->points.push_back(d);
55             r->points.clear();
56             r->points.push_back(d);
57             r->points.push_back(c);
58             r->points.push_back(a);
59             // - rebuild edge
60             add(l, false);
```

```
61              add(r, false);
62              // - did one change on this triangle, drop it.
63              break;
64          }
65      }
66
67      return ret;
68  }
69
70
71  /*
72      ALGORITHM FOR REUNITING CONVEX POLYGONS IN A PARTITION INTO BIGGER CONVEX
            POLYGONS
73      THIS ALGORITHM DOES NOT FIND THE OPTIMAL SOLUTION (IE WITH THE LESS POLYGONS)
74      BUT JUST ONE POSSIBLE SOLUTION
75  */
76
77  void Partition::convex_simplify() {
78      if (!is_convex()) return;
79
80      set<Tile*> remaining(tiles);
81      while (!remaining.empty()) {
82          Tile* t = *remaining.begin();
83          remaining.erase(remaining.begin());
84
85          for (list<Vertex*>::iterator it = t->points.begin(); it != t->points.end()
                ; it++) {
86              list<Vertex*>::iterator v2_it = circ_next(t->points, it);
87              Vertex *v1 = *it, *v2 = *v2_it;
88
89              assert(v1->edgeTo.count(v2) > 0, "No edge to next vertice in vertice's
                    edge list");
90              Edge* e = v1->edgeTo[v2];
91              assert(e->left == t, "Edge " << e->from->point->label << e->to->point
                    ->label <<
92                  (e->right == t ? " is going the wrong way." : " does not have
                        correct neighbour information"));
93              Tile* t2 = e->right;
94              if (t2 == NULL) continue;
95
96              set<Tile*>::iterator it2 = remaining.find(t2);
97              if (it2 == remaining.end()) continue;
98
99              list<Vertex*>::iterator v1n = find(t2->points.begin(), t2->points.end
                    (), v1),
100                                     v2n = find(t2->points.begin(), t2->points.end
                                            (), v2);
101             assert(v1n != t2->points.end(), "Cannot find vertex v1 in neighbour's
                    vertex list.");
102             assert(v2n != t2->points.end(), "Cannot find vertex v2 in neighbour's
                    vertex list.");
103
104             if (Vec::cross(Vec(*circ_prev(t->points, it), v1), Vec(v1, *circ_next(
                    t2->points, v1n))) >= 0
105                 && Vec::cross(Vec(*circ_prev(t2->points, v2n), v2), Vec(v2, *
                        circ_next(t->points, v2_it))) >= 0) {
106                 // Combine tiles !
107                 remaining.erase(it2);
108                 remaining.insert(combine_tiles(e, false));  // start over
109                 break;
110             }
111         }
112     }
113 }
```

Listing 9: old.cpp – old algorithm for simplifying triangle partitions

```
1  /*
2      ALGORITHM FOR REUNITING TRIANGLES IN A PARTITION INTO BIGGER CONVEX POLYGONS
3      THIS ALGORITHM DOES NOT FIND THE OPTIMAL SOLUTION (IE WITH THE LESS POLYGONS)
4      BUT JUST ONE POSSIBLE SOLUTION
5  */
6
7  #include "poly.h"
8  #include <algorithm>
9
10 using namespace std;
11
12 /* OLD ALGORITHM - WORKED WITH TRIANGLES. */
13 list<Polygon*> Triangle::simplify_triangle_partition(list<Triangle> triangles) {
14     list<Polygon*> ret;
15     for (list<Triangle>::iterator it = triangles.begin(); it != triangles.end();
           it++) {
16         list<Point*> pts;
17         pts.push_back(it->a);
18         pts.push_back(it->b);
19         pts.push_back(it->c);
20         Polygon *p = new Polygon(pts);
21         if (!p->is_direct()) reverse(p->points.begin(), p->points.end());
22         ret.push_back(p);
23     }
24
25     for (list<Polygon*>::iterator convex_it = ret.begin(); convex_it != ret.end();
           convex_it++) {
26         Polygon *convex = *convex_it;
27
28         for(list<Polygon*>::iterator tri_it = next(convex_it); tri_it != ret.end()
               ; tri_it++) {
29             Polygon *tri = *tri_it;
30
31             for (int i = 0; i < 3; i++) {
32                 list<Point*>::iterator common = find(
33                     convex->points.begin(), convex->points.end(), *tri->points.
                           begin());
34                 if (common != convex->points.end()) {
35                     list<Point*>::iterator newpt, common2;
36
37                     if (*circ_next(convex->points, common) == *prev(tri->points.
                           end())) {
38                         common2 = circ_next(convex->points, common);
39                         newpt = next(tri->points.begin());
40                     } else if (*circ_prev(convex->points, common) == *next(tri->
                           points.begin())) {
41                         common2 = common;
42                         common = circ_prev(convex->points, common2);
43                         newpt = prev(tri->points.end());
44                     } else {
45                         // only one point in common
46                         break;
47                     }
48
49                     if (Vec::cross(Vec(*common2, *circ_next(convex->points,
                           common2)), Vec(*common2, *newpt))
50                         * Vec::cross(Vec(*common, *circ_prev(convex->points,
                               common)), Vec(*common, *newpt)) <= 0) {
51                         // FUSION !
52                         convex->points.insert(common2, *newpt);
53                         ret.erase(tri_it--);
54                     }
55
```

```
56                      break;
57                  }
58                  // rotate triangle
59                  tri->points.push_back(tri->points.front());
60                  tri->points.pop_front();
61              }
62          }
63      }
64
65      return ret;
66  }
```

Listing 10: path.cpp – path finding algorithm

```
1  #include "poly.h"
2  #include <algorithm>
3  #include <map>
4
5  // If true, print log of path finding to stderr
6  #define LOG_ENABLE false
7
8  using namespace std;
9
10 // Utility functions used here
11
12 Point Segment::middle(std::string label) {
13     return Point((a->x + b->x) / 2, (a->y + b->y) / 2, label);
14 }
15
16 Point Line::orthog_projection(Point* p) {
17     //  a x + b y + c = d
18     //  (x - px) = k a
19     //  (y - py) = k b
20     //  a (k a + px) + b (k b + py) + c = 0
21     //  k = ( -c - a px - b py) / (a^2 + b^2)
22     float k = - (c  + a * p->x + b * p->y) / (a * a + b * b);
23
24     return Point(p->x + k * a, p->y + k * b, "");
25 }
26
27 Point Line::intersection(Line *l2) {
28     //  a x  + b y   = -c
29     //  a' x + b' y  = -c'
30     // d = a b' - b a'
31     // x = (-c b' + b c') / d
32     // y = (-a c' + c a') / d
33     float d = a * l2->b - b * l2->a;
34     if (d == 0) return Point(0, 0, "no intersection");
35     return Point((- c * l2->b + l2->c * b) / d, (- a * l2->c + c * l2->a) / d);
36 }
37
38 Point Segment::closest(Point* p) {
39     Point orth = Line(a, b).orthog_projection(p);
40     if (Vec::dot(Vec(a, &orth), Vec(a, b)) >= 0 && Vec::dot(Vec(b, &orth), Vec(b,
           a)) >= 0 &&
41         Vec(p, &orth).norm() < Vec(p, a).norm() && Vec(p, &orth).norm() < Vec(p, b
              ).norm()) {
42         return orth;
43     } else {
44         if (Vec(p, a).norm() < Vec(p, b).norm()) {
45             return *a;
46         } else {
47             return *b;
48         }
49     }
50 }
51
52 Point Segment::closest(Point *p1, Point *p2) {
53     Line l(p1, p2), l2(a, b);
54     Point i = l2.intersection(&l);
55     if (i.label != "no intersection" && Vec::dot(Vec(a, &i), Vec(a, b)) >= 0 &&
           Vec::dot(Vec(b, &i), Vec(b, a)) >= 0) {
56         return i;
57     } else {
58         Point ortha = l.orthog_projection(a);
59         Point orthb = l.orthog_projection(b);
60         if (Vec(a, &ortha).norm() < Vec(b, &orthb).norm()) {
```

```
61              return *a;
62          } else {
63              return *b;
64          }
65      }
66  }
67
68  Tile* Partition::find_tile_containing(Point* p) {
69      for (set<Tile*>::iterator it = tiles.begin(); it != tiles.end(); it++) {
70          bool found = true;
71          for (list<Vertex*>::iterator vtx = (*it)->points.begin(); vtx != (*it)->
                points.end(); vtx++) {
72              if (Vec::cross(Vec(*circ_prev((*it)->points, vtx), *vtx), Vec((*vtx)->
                    point, p)) < 0) {
73                  found = false;
74                  break;
75              }
76          }
77
78          if (found) return *it;
79      }
80      return NULL;
81  }
82
83
84  // Real pathfinding code
85
86  struct tile_info {
87      Tile* tile;
88
89      float weight_h, weight_d;
90      Edge* from;      // left = prev, right = this tile
91      Point bary;
92      Point from_p;
93
94      tile_info(Tile* t) : tile(t), weight_h(-1), from(NULL) {
95          Polygon *p = tile->make_poly();
96          bary = p->barycenter();
97          delete p;
98      }
99  };
100
101 struct t_i_compare {
102     bool operator() (tile_info *a, tile_info *b) {
103         return a->weight_h < b->weight_h || (a->weight_h == b->weight_h && a < b);
104     }
105 };
106
107 Path* Path::find(Partition* partition, Point* from, Point* to, float margin) {
108     Path *p = new Path(from, to, partition, margin);
109
110     assert(partition->is_convex(), "Trying to find a path in a non-convex
            partition.");
111
112     Tile *start = partition->find_tile_containing(from), *end = partition->
            find_tile_containing(to);
113     assert(start != NULL, "No tile contains start point.");
114     assert(end != NULL, "No tile contains arrival point.");
115     if (start == NULL || end == NULL) return p;      // p = path not found from
            from to to
116
117     if (LOG_ENABLE) {
118         cerr << "Start tile: " << start->name() << ", end tile: " << end->name()
                << endl;
```

```
119     }
120     if (start == end) return p;
121
122     map<Tile*, tile_info*> info;
123     for (set<Tile*>::iterator it = partition->tiles.begin(); it != partition->
            tiles.end(); it++) {
124         info[*it] = new tile_info(*it);
125     }
126
127     set<tile_info*, t_i_compare> open;
128     set<tile_info*> closed;
129
130     info[start]->weight_h = info[start]->weight_d = 0;
131     info[start]->from_p = *from;
132     open.insert(info[start]);
133
134     while (closed.count(info[end]) == 0 && !open.empty()) {
135         tile_info* ti = *open.begin();
136         open.erase(open.begin());
137         closed.insert(ti);
138
139         if (LOG_ENABLE) {
140             cerr << "Got " << ti->tile->name() << ", h: " << ti->weight_h << " d:
                    " << ti->weight_d << endl;
141         }
142
143         for (list<Vertex*>::iterator it = ti->tile->points.begin(); it != ti->tile
                ->points.end(); it++) {
144             Edge* e = (*it)->edgeTo[*circ_next(ti->tile->points, it)];
145
146             Tile* t2 = e->right;
147             if (t2 == NULL) continue;
148
149             Point pa(*e->from->point), pb(*e->to->point);
150             if (margin > 0) {
151                 if (Vec(&pa, &pb).norm() < 2 * margin) continue;
152                 Vec uv = Vec(&pa, &pb).unitvec();
153                 pa.x += uv.x * margin;
154                 pa.y += uv.y * margin;
155                 pb.x -= uv.x * margin;
156                 pb.y -= uv.y * margin;
157             }
158
159             tile_info *ti2 = info[t2];
160
161             Point ip = Segment(&pa, &pb).closest(&ti->from_p);
162
163             float d = Vec(&ti->from_p, &ip).norm();
164
165             float h = 0;     // heuristic (A*)
166             if (t2 == end) {
167                 d += Vec(&ip, to).norm();
168             } else {
169                 h = Vec(&ip, to).norm();
170             }
171
172             float p_weight_h = ti->weight_d + d + h;
173             if (p_weight_h < ti2->weight_h || ti2->weight_h < 0) {
174                 open.erase(ti2);
175                 ti2->weight_h = p_weight_h;
176                 ti2->weight_d = ti->weight_d + d;
177                 ti2->from = e;
178                 ti2->from_p = ip;
179                 open.insert(ti2);
```

```
180
181                 if (LOG_ENABLE) {
182                     cerr << "  +" << ti2->tile->name() << " h: " << ti2->weight_h
                           << " d: " << ti2->weight_d << endl;;
183                 }
184             }
185         }
186     }
187
188     if (closed.count(info[end]) > 0) {
189         if (LOG_ENABLE) {
190             cerr << "Path determined." << endl;
191         }
192         p->points.push_front(to);
193
194         tile_info *ti = info[end];
195         while (ti->tile != start) {
196             p->edges.push_front(ti->from);
197             p->points.push_front(new Point(ti->from_p));
198
199             ti = info[ti->from->left];
200         }
201
202         p->points.push_front(from);
203
204         p->found = true;
205     } else {
206         if (LOG_ENABLE) {
207             cerr << "No path." << endl;
208         }
209     }
210     for (map<Tile*, tile_info*>::iterator it = info.begin(); it != info.end(); it
          ++) {
211         delete it->second;
212     }
213     return p;
214 }
215
216
217 void Path::optimize() {
218     list<Edge*>::iterator edge_it = edges.begin();
219     list<Point*>::iterator pt_it = next(points.begin());
220     while (edge_it != edges.end()) {
221         Point pa = *(*edge_it)->from->point, pb = *(*edge_it)->to->point;
222
223         if (margin > 0) {
224             Vec uv = Vec(&pa, &pb).unitvec();
225             pa.x += uv.x * margin;
226             pa.y += uv.y * margin;
227             pb.x -= uv.x * margin;
228             pb.y -= uv.y * margin;
229         }
230
231         Point m = Segment(&pa, &pb).closest(*prev(pt_it), *next(pt_it));
232         if (Vec(&m, *prev(pt_it)).norm() + Vec(&m, *next(pt_it)).norm()
233                 < Vec(*pt_it, *prev(pt_it)).norm() + Vec(*pt_it, *next(pt_it)).
                      norm()) {
234             **pt_it = m;
235         }
236
237         edge_it++; pt_it++;
238     }
239 }
240
```

```
241 Path* Path::clone() {
242     Path* clone = new Path(from, to, partition, 0);
243     clone->found = true;
244     clone->edges = edges;
245     for (list<Point*>::iterator it = points.begin(); it != points.end(); it++) {
246         clone->points.push_back(new Point(**it));
247     }
248     return clone;
249 }
250
251 float Path::length() {
252     if (points.size() < 2) return 0;
253
254     float l = 0;
255     for (list<Point*>::iterator pt = next(points.begin()); pt != points.end(); pt
        ++) {
256         l += Vec(*prev(pt), *pt).norm();
257     }
258     return l;
259 }
```

Listing 11: postscript.h – header file for PostScript writing functions

```
#ifndef DEF_POSTSCRIPT_H
#define DEF_POSTSCRIPT_H

#include <string>
#include <stdio.h>

#include "poly.h"

class PostScript {
    private:
    FILE *file;
    int w, h;
    int write_y;
    int page;

    public:
    PostScript(std::string fstr, int page_width, int page_height);
    ~PostScript();

    void new_page();
    void end_page();
    void write(std::string s);

    void draw(Point *p, bool label = true);
    void draw(Line *l);
    void draw(Segment *s);
    void draw(Triangle *t, bool draw_points = true);
    void draw(Polygon *p, bool draw_points = true, bool fill = false);

    void draw(Partition *p, bool info = true);
    void draw_part_outline(Partition *p);
    void draw(Path *p);
};

#endif
```

Listing 12: postscript.cpp – code for writing PostScript files

```
1  #include "postscript.h"
2
3  using namespace std;
4
5  PostScript::PostScript(string fstr, int page_width, int page_height) : w(
       page_width), h(page_height) {
6      page = 1;
7
8      file = fopen(fstr.c_str(), "w");
9      if (!file) {
10         cerr << "Unable to open file for output: " << file << endl;
11     } else {
12         fprintf(file, "%%!PS-Adobe-2.0 EPSF-2.0\n");
13         fprintf(file, "%%%%BoundingBox: 0 0 %d %d\n\n", w, h);
14         fprintf(file, "/Georgia findfont 14 scalefont setfont\n\n");
15     }
16 }
17
18 PostScript::~PostScript() {
19     fclose(file);
20 }
21
22 void PostScript::new_page() {
23     fprintf(file, "\n%%%%Page: %d\n", page);
24     page++;
25     write_y = 5;
26 }
27
28 void PostScript::end_page() {
29     fprintf(file, "showpage\n");
30     fflush(file);
31 }
32
33 void PostScript::write(string s) {
34     fprintf(file, "0 0 0 setrgbcolor 5 %d moveto (%s) show\n", write_y, s.c_str())
           ;
35     write_y += 15;
36 }
37
38 // Drawing primitives
39
40 void PostScript::draw(Point *p, bool label) {
41     fprintf(file, "0 0 0 setrgbcolor ");
42     fprintf(file, "newpath %g %g 2 0 360 arc fill stroke\n", p->x, p->y);
43     if (p->label != "" && label) {
44         fprintf(file, "1 0 0 setrgbcolor ");
45         fprintf(file, "%g %g moveto (%s) show\n", p->x - 7, p->y + 5, p->label.
               c_str());
46     }
47 }
48
49 void PostScript::draw(Line *l) {
50     fprintf(file, "0.5 0 0.5 setrgbcolor 2 setlinewidth ");
51     if (l->b == 0) {
52         if (l->a == 0) return;  // should not happen
53         float x = -l->c / l->a;
54         fprintf(file, "newpath %g %d moveto %g %d lineto stroke\n", x, 0, x, h);
55     } else {
56         float y0 = -l->c / l->b;
57         float yf = -(l->c + l->a*w) / l->b;
58         fprintf(file, "newpath %d %g moveto %d %g lineto stroke\n", 0, y0, w, yf);
59     }
60
```

```
61 }
62
63 void PostScript::draw(Segment *s) {
64     fprintf(file, "0.1 0.5 0.1 setrgbcolor 2 setlinewidth ");
65     fprintf(file, "newpath %g %g moveto %g %g lineto stroke\n", s->a->x, s->a->y,
           s->b->x, s->b->y);
66 }
67
68 void PostScript::draw(Triangle *t, bool draw_points) {
69     fprintf(file, "0 0.5 0.5 setrgbcolor 1 setlinewidth ");
70     fprintf(file, "newpath %g %g moveto %g %g lineto %g %g lineto closepath stroke
           \n",
71         t->a->x, t->a->y, t->b->x, t->b->y, t->c->x, t->c->y);
72     if (draw_points) {
73         draw(t->a);
74         draw(t->b);
75         draw(t->c);
76     }
77 }
78
79 void PostScript::draw(Polygon *p, bool draw_points, bool fill) {
80     if (fill) fprintf(file, "0.9 0.9 0.9 setrgbcolor 1 setlinewidth ");
81     else fprintf(file, "0 0 1 setrgbcolor ");
82     fprintf(file, "newpath");
83     for (list<Point*>::iterator it = p->points.begin(); it != p->points.end(); it
           ++) {
84         fprintf(file, " %g %g %s", (*it)->x, (*it)->y, (it == p->points.begin() ?
               "moveto" : "lineto"));
85     }
86     if (fill) fprintf(file, " fill\n");
87     else fprintf(file, " closepath stroke\n");
88     if (draw_points) {
89         for (list<Point*>::iterator it = p->points.begin(); it != p->points.end();
               it++) {
90             draw(*it);
91         }
92     }
93 }
94
95 // Drawing other stuff
96
97
98 // Nicely draws the partition on screen. Also writes out error messages if data is
       inconsistent.
99 void PostScript::draw(Partition *part, bool info) {
100     string tiles_str;
101
102     for (set<Tile*>::iterator it = part->tiles.begin(); it != part->tiles.end();
           it++) {
103         if (tiles_str != "") tiles_str += ", ";
104         tiles_str += (*it)->name();
105
106         Polygon* p = (*it)->make_poly();
107         draw(p, false, true);
108         delete p;
109     }
110
111     for (set<Tile*>::iterator it = part->tiles.begin(); it != part->tiles.end();
           it++) {
112         Polygon* p = (*it)->make_poly();
113         draw(p, false);
114         delete p;
115     }
116
```

```
117     for (set<Vertex*>::iterator it = part->vertices.begin(); it != part->vertices.
            end(); it++) {
118         draw((*it)->point);
119     }
120 
121     if (info) {
122         write("tiles: " + tiles_str);
123         write(string(part->is_convex() ? "convex" : "not convex") + ", " +
124             tostr(part->vertices.size()) + " vertices, "+
125             tostr(part->tiles.size()) + " tiles");
126 
127         // check if all edge info is correct
128         for (set<Tile*>::iterator it = part->tiles.begin(); it != part->tiles.end
                (); it++) {
129             for (list<Vertex*>::iterator it2 = (*it)->points.begin(); it2 != (*it)
                    ->points.end(); it2++) {
130                 Edge *e = (*it2)->edgeTo[*circ_next((*it)->points, it2)];
131                 if (e->left != *it)
132                         write("Warn: edge " + e->from->point->label + e->to->point
                                ->label
133                             + " has incorrect left tile info.");
134                 if (e->dual->right != *it)
135                         write("Warn: edge " + e->to->point->label + e->from->point
                                ->label
136                             + " has incorrect right tile info.");
137             }
138         }
139     }
140 }
141 
142 void PostScript::draw_part_outline(Partition *part) {
143     for (set<Tile*>::iterator it = part->tiles.begin(); it != part->tiles.end();
            it++) {
144         Polygon* p = (*it)->make_poly();
145         draw(p, false, true);
146         delete p;
147     }
148 
149     fprintf(file, "0 0 1 setrgbcolor 1 setlinewidth\n");
150     for (set<Tile*>::iterator it = part->tiles.begin(); it != part->tiles.end();
            it++) {
151         for (list<Vertex*>::iterator it2 = (*it)->points.begin(); it2 != (*it)->
                points.end(); it2++) {
152             Edge *e = (*it2)->edge_to(*circ_next((*it)->points, it2));
153             if (e->right == NULL) {
154                 fprintf(file, "newpath %g %g moveto %g %g lineto stroke\n", e->
                        from->point->x, e->from->point->y,
155                         e->to->point->x, e->to->point->y);
156             }
157         }
158     }
159 }
160 
161 
162 void PostScript::draw(Path *p) {
163     if (!p->found) {
164         draw(p->from);
165         draw(p->to);
166         write("No path found.");
167         return;
168     }
169     float l = 0;
170     for (list<Point*>::iterator pt = next(p->points.begin()); pt != p->points.end
            (); pt++) {
```

```
171             Segment s(*prev(pt), *pt);
172             draw(&s);
173             l += Vec(*prev(pt), *pt).norm();
174         }
175         for (list<Point*>::iterator pt = p->points.begin(); pt != p->points.end(); pt
                ++) {
176             draw(*pt, (pt == p->points.begin() || next(pt) == p->points.end()));
177         }
178         write("path length: " + tostr(l));
179         write("path nodes: " + tostr(p->points.size()));
180 }
```

Listing 13: main.cpp – main command line application

```
1  #include "poly.h"
2  #include "postscript.h"
3  #include <fstream>
4
5  using namespace std;
6
7  void log(PostScript &ps, string s) {
8      cout << s << endl;
9      ps.write(s);
10 }
11
12 pair<Partition*, Partition*> do_algo_b(PostScript &ps, string sample_name,
       Partition* tri) {
13     Partition *del = new Partition(*tri);
14
15     tri->convex_simplify();
16
17     ps.new_page();
18     ps.draw(tri);
19     log(ps, "(" + sample_name + "b) simplification of (" + sample_name + ")");
20     ps.end_page();
21
22     ps.new_page();
23     bool u = del->delaunay_pass();
24     ps.draw(del);
25     log(ps, "(" + sample_name + "c) delaunay pass on not-yet-simplified (" +
           sample_name + ") : "
26             + string(u ? "usefull" : "useless"));
27     ps.end_page();
28
29     pair<Partition*, Partition*> ret(new Partition(*del), del);
30
31     ps.new_page();
32     del->convex_simplify();
33     ps.draw(del);
34     log(ps, "(" + sample_name + "d) simplification of (" + sample_name + "c)");
35     ps.end_page();
36
37     return ret;
38 }
39
40 void do_algo_a(PostScript &ps, Partition* tri) {
41     ps.new_page();
42     ps.draw(tri);
43     log(ps, "(0) original data");
44     ps.end_page();
45     Partition *tri2 = new Partition(*tri);
46
47     ps.new_page();
48     tri->earclip_triangulation_inplace();
49     ps.draw(tri);
50     log(ps, "(1) ear-clipping triangulation method on (0)");
51     ps.end_page();
52
53     do_algo_b(ps, "1", tri);
54
55     ps.new_page();
56     tri2->diagonal_triangulation();
57     ps.draw(tri2);
58     log(ps, "(2) diagonal triangulation method on (0)");
59     ps.end_page();
60
61     do_algo_b(ps, "2", tri2);
```

```
62 }
63
64 pair<Partition*, Partition*> do_algo(PostScript &ps, list<Polygon*> &pieces) {
65     // first method : make one big tile, ear clipping, simplification
66     Partition *part = new Partition();
67     for (list<Polygon*>::iterator it = pieces.begin(); it != pieces.end(); it++)
           part->add(*it);
68     do_algo_a(ps, part);
69
70     // second method : using monotone polygons
71     pair<Partition*, Partition*> monotri = Partition::monotone_triangulation(
           pieces);
72
73     ps.new_page();
74     ps.draw(monotri.first);
75     log(ps, "(3) monotone polygon decomposition");
76     ps.end_page();
77
78     ps.new_page();
79     ps.draw(monotri.second);
80     log(ps, "(4) triangulation of monotone polygons in (3)");
81     ps.end_page();
82
83     return do_algo_b(ps, "4", monotri.second);
84 }
85
86 void path_on_partition(PostScript &ps, Partition* part, Point* from, Point* to) {
87         Path *path = Path::find(part, from, to, 0);
88
89         ps.new_page();
90         ps.draw_part_outline(part);
91         if (path->found) ps.write("first approximation of a path");
92         ps.draw(path);
93         ps.end_page();
94
95         if (path->found) {
96             ps.new_page();
97             path->optimize();
98             ps.draw_part_outline(part);
99             ps.write("once optimized path");
100            ps.draw(path);
101            ps.end_page();
102        }
103 }
104
105
106 int main(int argc, char *argv[]) {
107     if (argc != 3) {
108         cout << "Usage: " << argv[0] << " <input_file> <output_file>" << endl;
109         return 1;
110     }
111
112     ifstream in;
113     in.open(argv[1], ifstream::in);
114
115     int w, h;
116     in >> w;
117     in >> h;
118     PostScript ps(argv[2], w, h);
119
120     // Earclipping triangulation method - read points from stdin
121     int n; in >> n;
122     for (int i = 0; i < n; i++) {
123         Polygon *poly = Polygon::read(in, w, h);
```

```
124
125         ps.new_page();
126         ps.draw(poly);
127         ps.write(tostr(poly->points.size()) + " points");
128         ps.write("sum of angles: " + tostr(poly->sum_of_angles()));
129         ps.write(poly->is_direct() ? "direct" : "indirect");
130         ps.write(poly->is_convex() ? "convex" : "not convex");
131         ps.end_page();
132
133         int algo;
134         in >> algo;
135
136         if (algo == 0) {
137             Partition* part = new Partition();
138             part->add(poly);
139
140             do_algo_a(ps, part);
141             delete part;
142         } else {
143             list<Polygon*> p; p.push_back(poly);
144             do_algo(ps, p);
145         }
146     }
147
148     // Bigger set of data : polygons and holes
149     in >> n;     // n polygons
150     map<string, Point*> points_map;
151     list<Polygon*> pieces;
152     ps.new_page();
153     for (int i = 0; i < n; i++) {
154         Polygon *p = Polygon::read(in, points_map, w, h);
155         if (!p->is_direct()) {
156             ps.draw(p, false, true);
157         }
158         ps.draw(p, true, false);
159         pieces.push_back(p);
160     }
161     ps.write("map & obstacles");
162     ps.end_page();
163
164     pair<Partition*, Partition*> maps = do_algo(ps, pieces);
165
166     in >> n;     // n paths to calculate
167     for (int i = 0; i < n; i++) {
168         int p; in >> p;
169         float x = 0, y = 0;
170         for (int j = 0; j < p; j++) {
171             string id; in >> id;
172             x += points_map[id]->x; y += points_map[id]->y;
173         }
174         Point *from = new Point(x / p, y / p, "From");
175
176         in >> p; x = y = 0;
177         for (int j = 0; j < p; j++) {
178             string id; in >> id;
179             x += points_map[id]->x; y += points_map[id]->y;
180         }
181         Point *to = new Point(x / p, y / p, "To");
182
183         path_on_partition(ps, maps.first, from, to);
184         path_on_partition(ps, maps.second, from, to);
185     }
186
187
```

```
188     return 0;
189 }
```

Listing 14: FXUI_main.cpp – FOX GUI application

```
1  #include <fx.h>
2
3  #include "poly.h"
4  #include "postscript.h"
5
6  using namespace std;;
7
8  enum {
9      DS_NONE,
10     DS_POLYSET,
11     DS_PARTITION,
12     DS_TRIANGULATION,
13     DS_CONVEX_PARTITION,
14     DS_PATH,
15 };
16
17 struct DataSet {
18     string name;
19     int type;
20     list<Polygon*> polygons;
21     Partition* partition;
22     Path* path;
23
24     DataSet(string n, int t) : name(n), type(t), partition(NULL), path(NULL) {}
25 };
26
27
28 class Triangulator : public FXMainWindow {
29     FXDECLARE(Triangulator);
30
31     private:
32     FXApp                   *a;
33     FXIconSource            *iconsrc;
34
35     FXHorizontalFrame       *contents;
36     FXVerticalFrame         *viewFrame;
37     FXVerticalFrame         *cmdFrame;
38     FXLabel                 *viewTitle;
39     FXCanvas                *view;
40     FXList                  *pageList;
41     FXButton                *polyToPartB;
42     FXButton                *earTriB, *diagTriB;
43     FXButton                *monoTriB, *delTriB;
44     FXButton                *delPassB, *convexSimplifyB;
45     FXButton                *shortenB;
46
47     FXImage                 *bgImage;
48
49     int page;
50     vector<DataSet> data;
51     Polygon *inputPoly;
52     Point *pathPtA;
53
54     int selectedType();
55     DataSet *selected();
56
57     void paintPoint(FXDCWindow &dc, Point *point, FXColor color);
58     void paintFilledPoly(FXDCWindow &dc, Polygon *poly, FXColor color);
59     void paintPoly(FXDCWindow &dc, Polygon *poly, FXColor color, FXColor
           pointscolor, bool pts = true);
60     void paintPath(FXDCWindow &dc, Path *path, FXColor color, FXColor pointscolor,
           bool pts = true);
61
```

```
62      protected:
63      Triangulator() {}
64
65      public:
66      // message handlers
67      long onPaint(FXObject*, FXSelector, void*);
68      long onMouseDown(FXObject*, FXSelector, void*);
69      long onMouseUp(FXObject*, FXSelector, void*);
70      long onMouseMove(FXObject*, FXSelector, void*);
71
72      long onSelectItem(FXObject*, FXSelector, void*);
73      long onUpdateList(FXObject*, FXSelector, void*);
74      long onUpdateClear(FXObject*, FXSelector, void*);
75      long onUpdatePoly2part(FXObject*, FXSelector, void*);
76      long onUpdateEarClipT(FXObject*, FXSelector, void*);
77      long onUpdateDiagonalT(FXObject*, FXSelector, void*);
78      long onUpdateMonotoneT(FXObject*, FXSelector, void*);
79      long onUpdateDelaunayT(FXObject*, FXSelector, void*);
80      long onUpdateDelaunayPass(FXObject*, FXSelector, void*);
81      long onUpdateConvexS(FXObject*, FXSelector, void*);
82      long onUpdateShorten(FXObject*, FXSelector, void*);
83
84      long onClear(FXObject*, FXSelector, void*);
85      long onAddPage(FXObject*, FXSelector, void*);
86      long onDelPage(FXObject*, FXSelector, void*);
87      long onPoly2part(FXObject*, FXSelector, void*);
88      long onEarTri(FXObject*, FXSelector, void*);
89      long onDiagTri(FXObject*, FXSelector, void*);
90      long onMonoTri(FXObject*, FXSelector, void*);
91      long onDelTri(FXObject*, FXSelector, void*);
92      long onDelPass(FXObject*, FXSelector, void*);
93      long onConvexSimplify(FXObject*, FXSelector, void*);
94      long onShorten(FXObject*, FXSelector, void*);
95
96      long onLoadBG(FXObject*, FXSelector, void*);
97      long onRstBG(FXObject*, FXSelector, void*);
98      long onExport(FXObject*, FXSelector, void*);
99
100
101     // messages
102     enum {
103         ID_CANVAS = FXMainWindow::ID_LAST,
104         ID_LIST,
105
106         ID_CLEAR,
107         ID_LOADBG,
108         ID_RSTBG,
109         ID_EXPORT,
110
111         ID_DELPAGE,
112         ID_ADDPAGE,
113         ID_POLY2PART,
114         ID_EARCLIPT,
115         ID_DIAGONALT,
116         ID_MONOTONET,
117         ID_DELAUNAYT,
118         ID_DELAUNAYPASS,
119         ID_CONVEXS,
120         ID_SHORTEN,
121
122         ID_LAST
123     };
124
125     // constructor
```

```
126     Triangulator(FXApp *a);
127
128     virtual void create();
129     virtual ~Triangulator() {}
130 };
131
132 FXDEFMAP(Triangulator) TriangulatorMap[] = {
133     // message type, id, message handler
134
135     // actions on canvas
136     FXMAPFUNC(SEL_PAINT,              Triangulator::ID_CANVAS,     Triangulator::
            onPaint),
137     FXMAPFUNC(SEL_LEFTBUTTONPRESS,    Triangulator::ID_CANVAS,     Triangulator::
            onMouseDown),
138     FXMAPFUNC(SEL_RIGHTBUTTONPRESS,   Triangulator::ID_CANVAS,     Triangulator::
            onMouseDown),
139     FXMAPFUNC(SEL_LEFTBUTTONRELEASE, Triangulator::ID_CANVAS,     Triangulator::
            onMouseUp),
140     FXMAPFUNC(SEL_RIGHTBUTTONRELEASE, Triangulator::ID_CANVAS,    Triangulator::
            onMouseUp),
141     FXMAPFUNC(SEL_MOTION,             Triangulator::ID_CANVAS,     Triangulator::
            onMouseMove),
142     // select another dataset
143     FXMAPFUNC(SEL_CHANGED,            Triangulator::ID_LIST,       Triangulator::
            onSelectItem),
144     // update data
145     FXMAPFUNC(SEL_UPDATE,             Triangulator::ID_LIST,       Triangulator::
            onUpdateList),
146     FXMAPFUNC(SEL_UPDATE,             Triangulator::ID_CLEAR,      Triangulator::
            onUpdateClear),
147     FXMAPFUNC(SEL_UPDATE,             Triangulator::ID_POLY2PART,  Triangulator::
            onUpdatePoly2part),
148     FXMAPFUNC(SEL_UPDATE,             Triangulator::ID_EARCLIPT,   Triangulator::
            onUpdateEarClipT),
149     FXMAPFUNC(SEL_UPDATE,             Triangulator::ID_DIAGONALT,  Triangulator::
            onUpdateDiagonalT),
150     FXMAPFUNC(SEL_UPDATE,             Triangulator::ID_MONOTONET,  Triangulator::
            onUpdateMonotoneT),
151     FXMAPFUNC(SEL_UPDATE,             Triangulator::ID_DELAUNAYT,  Triangulator::
            onUpdateDelaunayT),
152     FXMAPFUNC(SEL_UPDATE,             Triangulator::ID_DELAUNAYPASS, Triangulator::
            onUpdateDelaunayPass),
153     FXMAPFUNC(SEL_UPDATE,             Triangulator::ID_CONVEXS,    Triangulator::
            onUpdateConvexS),
154     FXMAPFUNC(SEL_UPDATE,             Triangulator::ID_SHORTEN,    Triangulator::
            onUpdateShorten),
155     // actions
156     FXMAPFUNC(SEL_COMMAND,            Triangulator::ID_CLEAR,      Triangulator::
            onClear),
157     FXMAPFUNC(SEL_COMMAND,            Triangulator::ID_ADDPAGE,    Triangulator::
            onAddPage),
158     FXMAPFUNC(SEL_COMMAND,            Triangulator::ID_DELPAGE,    Triangulator::
            onDelPage),
159     FXMAPFUNC(SEL_COMMAND,            Triangulator::ID_POLY2PART,  Triangulator::
            onPoly2part),
160     FXMAPFUNC(SEL_COMMAND,            Triangulator::ID_EARCLIPT,   Triangulator::
            onEarTri),
161     FXMAPFUNC(SEL_COMMAND,            Triangulator::ID_DIAGONALT,  Triangulator::
            onDiagTri),
162     FXMAPFUNC(SEL_COMMAND,            Triangulator::ID_MONOTONET,  Triangulator::
            onMonoTri),
163     FXMAPFUNC(SEL_COMMAND,            Triangulator::ID_DELAUNAYT,  Triangulator::
            onDelTri),
164     FXMAPFUNC(SEL_COMMAND,            Triangulator::ID_DELAUNAYPASS, Triangulator::
```

```
        onDelPass),
165     FXMAPFUNC(SEL_COMMAND,          Triangulator::ID_CONVEXS,    Triangulator::
            onConvexSimplify),
166     FXMAPFUNC(SEL_COMMAND,          Triangulator::ID_SHORTEN,    Triangulator::
            onShorten),
167
168     FXMAPFUNC(SEL_COMMAND,          Triangulator::ID_LOADBG,     Triangulator::
            onLoadBG),
169     FXMAPFUNC(SEL_COMMAND,          Triangulator::ID_RSTBG,      Triangulator::
            onRstBG),
170     FXMAPFUNC(SEL_COMMAND,          Triangulator::ID_EXPORT,     Triangulator::
            onExport),
171 };
172
173 FXIMPLEMENT(Triangulator,FXMainWindow,TriangulatorMap,ARRAYNUMBER(TriangulatorMap)
        );
174
175 Triangulator::Triangulator(FXApp *app) : FXMainWindow(app, "Triangulator", NULL,
        NULL, DECOR_ALL, 0, 0, 800, 600) {
176     a = app;
177     iconsrc = new FXIconSource(a);
178
179     FXHorizontalFrame *hf;
180
181     contents = new FXHorizontalFrame(this, LAYOUT_SIDE_TOP|LAYOUT_FILL_X|
            LAYOUT_FILL_Y);
182
183     viewFrame = new FXVerticalFrame(contents, FRAME_SUNKEN|LAYOUT_FILL_X|
            LAYOUT_FILL_Y|LAYOUT_TOP|LAYOUT_LEFT,0,0,0,0,10,10,10,10);
184
185     viewTitle = new FXLabel(viewFrame, "(no page selected)", NULL,
            JUSTIFY_CENTER_X|LAYOUT_FILL_X);
186     new FXHorizontalSeparator(viewFrame, SEPARATOR_GROOVE|LAYOUT_FILL_X);
187     view = new FXCanvas(viewFrame, this, ID_CANVAS, FRAME_SUNKEN|FRAME_THICK|
            LAYOUT_FILL_X|LAYOUT_FILL_Y|LAYOUT_FILL_ROW|LAYOUT_FILL_COLUMN);
188
189     cmdFrame = new FXVerticalFrame(contents, FRAME_SUNKEN|LAYOUT_FILL_Y|LAYOUT_TOP
            |LAYOUT_LEFT, 0, 0, 0, 0, 10, 10, 10, 10);
190
191     pageList = new FXList(cmdFrame, this, ID_LIST, LAYOUT_FILL_X|LAYOUT_FILL_Y|
            FRAME_THICK|FRAME_SUNKEN);
192
193     hf = new FXHorizontalFrame(cmdFrame, LAYOUT_TOP|LAYOUT_LEFT|LAYOUT_FILL_X);
194     new FXButton(hf, "&Add page", NULL, this, ID_ADDPAGE, FRAME_THICK|FRAME_RAISED
            |LAYOUT_FILL_X|LAYOUT_TOP|LAYOUT_LEFT,0,0,0,0,10,10,5,5);
195     polyToPartB = new FXButton(hf, "To Partition", NULL, this, ID_POLY2PART,
            FRAME_THICK|FRAME_RAISED|LAYOUT_FILL_X|LAYOUT_TOP|LAYOUT_LEFT
            ,0,0,0,0,10,10,5,5);
196
197     hf = new FXHorizontalFrame(cmdFrame, LAYOUT_TOP|LAYOUT_LEFT|LAYOUT_FILL_X);
198     earTriB = new FXButton(hf, "Ear clip tri", NULL, this, ID_EARCLIPT,
            FRAME_THICK|FRAME_RAISED|LAYOUT_FILL_X|LAYOUT_TOP|LAYOUT_LEFT
            ,0,0,0,0,10,10,5,5);
199     diagTriB = new FXButton(hf, "Diagonal tri", NULL, this, ID_DIAGONALT,
            FRAME_THICK|FRAME_RAISED|LAYOUT_FILL_X|LAYOUT_TOP|LAYOUT_LEFT
            ,0,0,0,0,10,10,5,5);
200
201     hf = new FXHorizontalFrame(cmdFrame, LAYOUT_TOP|LAYOUT_LEFT|LAYOUT_FILL_X);
202     monoTriB = new FXButton(hf, "Monotone tri", NULL, this, ID_MONOTONET,
            FRAME_THICK|FRAME_RAISED|LAYOUT_FILL_X|LAYOUT_TOP|LAYOUT_LEFT
            ,0,0,0,0,10,10,5,5);
203     delTriB = new FXButton(hf, "Delaunay tri", NULL, this, ID_DELAUNAYT,
            FRAME_THICK|FRAME_RAISED|LAYOUT_FILL_X|LAYOUT_TOP|LAYOUT_LEFT
            ,0,0,0,0,10,10,5,5);
```

```
204
205     hf = new FXHorizontalFrame(cmdFrame, LAYOUT_TOP|LAYOUT_LEFT|LAYOUT_FILL_X);
206     delPassB = new FXButton(hf, "Delaunay pass", NULL, this, ID_DELAUNAYPASS,
            FRAME_THICK|FRAME_RAISED|LAYOUT_FILL_X|LAYOUT_TOP|LAYOUT_LEFT
            ,0,0,0,0,10,10,5,5);
207     convexSimplifyB = new FXButton(hf, "Convex simplify", NULL, this, ID_CONVEXS,
            FRAME_THICK|FRAME_RAISED|LAYOUT_FILL_X|LAYOUT_TOP|LAYOUT_LEFT
            ,0,0,0,0,10,10,5,5);
208
209     shortenB = new FXButton(cmdFrame, "Shorten path", NULL, this, ID_SHORTEN,
            FRAME_THICK|FRAME_RAISED|LAYOUT_FILL_X|LAYOUT_TOP|LAYOUT_LEFT
            ,0,0,0,0,10,10,5,5);
210
211     new FXHorizontalSeparator(cmdFrame, SEPARATOR_GROOVE|LAYOUT_FILL_X);
212     hf = new FXHorizontalFrame(cmdFrame, LAYOUT_TOP|LAYOUT_LEFT|LAYOUT_FILL_X);
213     new FXButton(hf, "&Clear", NULL, this, ID_CLEAR, FRAME_THICK|FRAME_RAISED|
            LAYOUT_FILL_X|LAYOUT_TOP|LAYOUT_LEFT,0,0,0,0,10,10,5,5);
214     new FXButton(hf, "&Delete page", NULL, this, ID_DELPAGE, FRAME_THICK|
            FRAME_RAISED|LAYOUT_FILL_X|LAYOUT_TOP|LAYOUT_LEFT,0,0,0,0,10,10,5,5);
215
216     hf = new FXHorizontalFrame(cmdFrame, LAYOUT_TOP|LAYOUT_LEFT|LAYOUT_FILL_X);
217     new FXButton(hf, "&Load BG", NULL, this, ID_LOADBG, FRAME_THICK|FRAME_RAISED|
            LAYOUT_FILL_X|LAYOUT_TOP|LAYOUT_LEFT,0,0,0,0,10,10,5,5);
218     new FXButton(hf, "&Clear BG", NULL, this, ID_RSTBG, FRAME_THICK|FRAME_RAISED|
            LAYOUT_FILL_X|LAYOUT_TOP|LAYOUT_LEFT,0,0,0,0,10,10,5,5);
219     new FXButton(hf, "&Export", NULL, this, ID_EXPORT, FRAME_THICK|FRAME_RAISED|
            LAYOUT_FILL_X|LAYOUT_TOP|LAYOUT_LEFT,0,0,0,0,10,10,5,5);
220
221     new FXButton(cmdFrame,"&Exit",NULL,getApp(),FXApp::ID_QUIT,FRAME_THICK|
            FRAME_RAISED|LAYOUT_FILL_X|LAYOUT_TOP|LAYOUT_LEFT,0,0,0,0,10,10,5,5);
222
223     // private data
224     page = 0;
225     inputPoly = NULL;
226     pathPtA = NULL;
227
228     bgImage = NULL;
229 }
230
231
232 void Triangulator::create(){
233   FXMainWindow::create();
234   show(PLACEMENT_SCREEN);
235 }
236
237 int Triangulator::selectedType() {
238     FXint item = pageList->getCurrentItem();
239     if (item >= 0 && item < data.size()) {
240         return data[item].type;
241     }
242     return DS_NONE;
243 }
244
245 DataSet *Triangulator::selected() {
246     FXint item = pageList->getCurrentItem();
247     if (item >= 0 && item < data.size()) {
248         return &data[item];
249     }
250     return NULL;
251 }
252
253 // Painting
254
255 void Triangulator::paintPoint(FXDCWindow &dc, Point* p, FXColor c) {
```

```
256     dc.setForeground(c);
257     dc.fillEllipse(p->x - 3, p->y - 3, 6, 6);
258     dc.setFont(getApp()->getNormalFont());
259     dc.drawText(p->x - 7, p->y - 8, p->label.c_str(), p->label.length());
260 }
261
262 void Triangulator::paintFilledPoly(FXDCWindow &dc, Polygon* p, FXColor c) {
263     int n = p->points.size();
264     FXPoint *pts = new FXPoint[n];
265     int i = 0;
266     for (list<Point*>::iterator it = p->points.begin(); it != p->points.end(); it
            ++) {
267         pts[i].set((*it)->x, (*it)->y);
268         i++;
269     }
270     dc.setForeground(c);
271     dc.fillConcavePolygon(pts, p->points.size());
272     delete pts;
273 }
274
275 void Triangulator::paintPoly(FXDCWindow &dc, Polygon* p, FXColor c, FXColor pc,
        bool pts) {
276     Point* prev = *(p->points.rbegin());
277     dc.setForeground(c);
278     for (list<Point*>::iterator it = p->points.begin(); it != p->points.end(); it
            ++) {
279         dc.drawLine(prev->x, prev->y, (*it)->x, (*it)->y);
280         prev = *it;
281     }
282     if (pts) {
283         for (list<Point*>::iterator it = p->points.begin(); it != p->points.end();
                it++) {
284             paintPoint(dc, *it, pc);
285         }
286     }
287 }
288
289 void Triangulator::paintPath(FXDCWindow &dc, Path* p, FXColor c, FXColor pc, bool
        pts) {
290     Point* prev = NULL;
291     dc.setForeground(c);
292     for (list<Point*>::iterator it = p->points.begin(); it != p->points.end(); it
            ++) {
293         if (prev) dc.drawLine(prev->x, prev->y, (*it)->x, (*it)->y);
294         prev = *it;
295     }
296     if (pts) {
297         for (list<Point*>::iterator it = p->points.begin(); it != p->points.end();
                it++) {
298             paintPoint(dc, *it, pc);
299         }
300     }
301 }
302
303 long Triangulator::onPaint(FXObject *, FXSelector, void* ptr) {
304     if (ptr) {
305         FXEvent *ev = (FXEvent*) ptr;
306         FXDCWindow dc(view, ev);
307         dc.setForeground(view->getBackColor());
308         dc.fillRectangle(ev->rect.x,ev->rect.y,ev->rect.w,ev->rect.h);
309     }
310
311     FXDCWindow dc(view);
312     if (!ptr) {
```

```
313         dc.setForeground(view->getBackColor());
314         if (bgImage == NULL) dc.fillRectangle(0,0,view->getWidth(),view->getHeight
                ());
315     }
316
317     if (bgImage) {
318         dc.drawImage(bgImage, 0, 0);
319     }
320
321     DataSet *d = selected();
322
323     if (selectedType() == DS_POLYSET) {
324         for (list<Polygon*>::iterator it = d->polygons.begin(); it != d->polygons.
                end(); it++) {
325             paintPoly(dc, *it, ((*it)->is_direct() ? FXRGB(0, 100, 0) : FXRGB
                    (255,0,0)), FXRGB(0,0,0));
326         }
327         if (inputPoly != NULL) {
328             paintPoly(dc, inputPoly, FXRGB(0,0,255),FXRGB(0,0,0));
329         }
330     } else if (d != NULL && d->partition != NULL) {
331         Polygon *p[d->partition->tiles.size()];
332         int i = 0;
333         for (set<Tile*>::iterator it = d->partition->tiles.begin(); it != d->
                partition->tiles.end(); it++) {
334             p[i] = (*it)->make_poly();
335             i++;
336         }
337
338         for (i = 0; i < d->partition->tiles.size(); i++) {
339             paintFilledPoly(dc, p[i],FXRGB(230, 230, 230));
340         }
341         for (i = 0; i < d->partition->tiles.size(); i++) {
342             paintPoly(dc, p[i], FXRGB(200,0,0),FXRGB(0,0,0), true);
343         }
344
345         for (i = 0; i < d->partition->tiles.size(); i++) delete p[i];
346
347         if (d->path != NULL) {
348             paintPath(dc, d->path, FXRGB(0,50,200), FXRGB(0,0,0), true);
349         }
350     }
351
352     return 1;
353 }
354
355 // Events
356
357 long Triangulator::onMouseDown(FXObject *, FXSelector, void* ptr) {
358     // nothing to do
359     return 1;
360 }
361
362 long Triangulator::onMouseUp(FXObject *, FXSelector, void* ptr) {
363     FXDCWindow dc(view);
364     FXEvent *ev = (FXEvent*)ptr;
365
366     if (selectedType() == DS_POLYSET) {
367         if (ev->click_button == LEFTBUTTON) {
368             char p = 'A' + selected()->polygons.size();
369             string n;
370             n += p;
371             if (!inputPoly) {
372                 inputPoly = new Polygon();
```

```
373                 inputPoly->points.push_back(new Point(ev->win_x, ev->win_y, n + "1
                        "));
374             }
375             inputPoly->points.push_back(new Point(ev->win_x, ev->win_y, n + tostr(
                    inputPoly->points.size() + 1)));
376         } else if (inputPoly && ev->click_button == RIGHTBUTTON) {
377             if (inputPoly->points.size() < 3) {
378                 delete inputPoly;
379             } else {
380                 selected()->polygons.push_back(inputPoly);
381             }
382             inputPoly = NULL;
383             handle(this, FXSEL(SEL_PAINT, ID_CANVAS), NULL);     // repaint
384         }
385     } else if (selectedType() == DS_TRIANGULATION || selectedType() ==
        DS_CONVEX_PARTITION) {
386         if (ev->click_button == LEFTBUTTON) {
387             if (pathPtA == NULL) {
388                 pathPtA = new Point(ev->win_x, ev->win_y, "From");
389             } else {
390                 Point *b = new Point(ev->win_x, ev->win_y, "To");
391
392                 data.push_back(DataSet(selected()->name + ".path", DS_PATH));
393                 data.back().partition = selected()->partition;
394                 data.back().path = Path::find(data.back().partition, pathPtA, b,
                        0);
395                 pathPtA = NULL;
396
397                 if (!data.back().path->found) {
398                     delete data.back().path;
399                     data.pop_back();
400                     handle(this, FXSEL(SEL_PAINT, ID_CANVAS), NULL);     // repaint
401                 }
402             }
403         } else if (ev->click_button == RIGHTBUTTON) {
404             if (pathPtA != NULL) {
405                 delete pathPtA;
406                 pathPtA = NULL;
407             }
408         }
409     }
410
411     return 1;
412 }
413
414 long Triangulator::onMouseMove(FXObject *, FXSelector, void* ptr) {
415     FXDCWindow dc(view);
416     FXEvent *ev = (FXEvent*)ptr;
417
418     if (selectedType() == DS_POLYSET && inputPoly != NULL) {
419         handle(this, FXSEL(SEL_PAINT, ID_CANVAS), NULL);     // repaint
420
421         paintPoly(dc, inputPoly, view->getBackColor(), 0, false);
422
423         inputPoly->points.back()->x = ev->win_x;
424         inputPoly->points.back()->y = ev->win_y;
425         paintPoly(dc, inputPoly, FXRGB(0,0,255), 0, false);
426     } else if ((selectedType() == DS_TRIANGULATION || selectedType() ==
        DS_CONVEX_PARTITION) && pathPtA != NULL) {
427         handle(this, FXSEL(SEL_PAINT, ID_CANVAS), NULL);     // repaint
428
429         dc.setForeground(view->getBackColor());
430         dc.drawLine(pathPtA->x, pathPtA->y, ev->last_x, ev->last_y);
431         dc.setForeground(FXRGB(0,50,200));
```

```
432         dc.drawLine(pathPtA->x, pathPtA->y, ev->win_x, ev->win_y);
433         paintPoint(dc, pathPtA, FXRGB(0,0,0));
434     }
435
436     return 1;
437 }
438
439 // Actions
440
441 long Triangulator::onClear(FXObject*, FXSelector, void*) {
442     set<Partition*> pp;
443     for (unsigned i = 0; i < data.size(); i++) {
444         for (list<Polygon*>::iterator it = data[i].polygons.begin(); it != data[i
                ].polygons.end(); it++) {
445             delete (*it);
446         }
447         if (data[i].partition) {
448             if (pp.count(data[i].partition) == 0) {
449                 pp.insert(data[i].partition);
450                 delete data[i].partition;
451             }
452         }
453         if (data[i].path) delete data[i].path;
454     }
455     data.clear();
456     page = 0;
457
458     return 1;
459 }
460
461 long Triangulator::onAddPage(FXObject*, FXSelector, void*) {
462     data.push_back(DataSet("p" + tostr(++page), DS_POLYSET));
463     return 1;
464 }
465
466 long Triangulator::onDelPage(FXObject*, FXSelector, void*) {
467     FXint item = pageList->getCurrentItem();
468     if (item >= 0 && item < data.size()) {
469         data.erase(data.begin() + item);
470     }
471     return 1;
472 }
473
474 long Triangulator::onPoly2part(FXObject*, FXSelector, void*) {
475     DataSet *d = selected();
476     if (!d || d->polygons.size() == 0) return 0;
477
478     Partition *p = new Partition();
479     for (list<Polygon*>::iterator it = d->polygons.begin(); it != d->polygons.end
            (); it++) {
480         p->add(*it);
481     }
482     data.push_back(DataSet(d->name + "$", DS_PARTITION));
483     data.back().partition = p;
484     return 1;
485 }
486
487 long Triangulator::onEarTri(FXObject*, FXSelector, void*) {
488     DataSet *d = selected();
489     if (!d || d->partition == 0) return 0;
490
491     Partition *p = new Partition(*d->partition);
492     p->earclip_triangulation_inplace();
493     data.push_back(DataSet(d->name + ".earclip_tri", DS_TRIANGULATION));
```

```
494     data.back().partition = p;
495     return 1;
496 }
497
498 long Triangulator::onDiagTri(FXObject*, FXSelector, void*) {
499     DataSet *d = selected();
500     if (!d || d->partition == 0) return 0;
501
502     Partition *p = new Partition(*d->partition);
503     p->diagonal_triangulation();
504     data.push_back(DataSet(d->name + ".diagonal_tri", DS_TRIANGULATION));
505     data.back().partition = p;
506     return 1;
507 }
508
509 long Triangulator::onMonoTri(FXObject*, FXSelector, void*) {
510     DataSet *d = selected();
511     if (!d || d->polygons.size() == 0) return 0;
512     string s = d->name;
513     string s1 = s + ".monotone";
514     string s2 = s + ".monotone_tri";
515
516     pair<Partition*, Partition*> p = Partition::monotone_triangulation(d->polygons
            );
517     data.push_back(DataSet(s1, DS_PARTITION));
518     data.back().partition = p.first;
519     data.push_back(DataSet(s2, DS_TRIANGULATION));
520     data.back().partition = p.second;
521     return 1;
522 }
523
524 long Triangulator::onDelTri(FXObject*, FXSelector, void*) {
525     FXMessageBox::error(this, MBOX_OK, "Error", "Not implemented.");
526     return 1;
527 }
528
529 long Triangulator::onDelPass(FXObject*, FXSelector, void*) {
530     DataSet *d = selected();
531     if (!d || d->partition == 0) return 0;
532
533     Partition *p = new Partition(*d->partition);
534     p->delaunay_pass();
535     data.push_back(DataSet(d->name + ".del_pass", DS_TRIANGULATION));
536     data.back().partition = p;
537     return 1;
538 }
539
540 long Triangulator::onConvexSimplify(FXObject*, FXSelector, void*) {
541     DataSet *d = selected();
542     if (!d || d->partition == 0) return 0;
543
544     Partition *p = new Partition(*d->partition);
545     p->convex_simplify();
546     data.push_back(DataSet(d->name + ".simplify", DS_CONVEX_PARTITION));
547     data.back().partition = p;
548     return 1;
549 }
550
551 long Triangulator::onShorten(FXObject*, FXSelector, void*) {
552     DataSet *d = selected();
553     if (!d || d->path == 0) return 0;
554
555     data.push_back(DataSet(d->name + "-", DS_PATH));
556     data.back().partition = d->partition;
```

```
557     data.back().path = d->path->clone();
558     data.back().path->optimize();
559     return 1;
560 }
561
562 long Triangulator::onLoadBG(FXObject*, FXSelector, void*) {
563     // Load background image
564     FXFileDialog dialog(this, "Export to file...");
565     dialog.setDirectory(getenv("PWD"));
566     dialog.setPatternList("*.jpg,*.png,*.gif,*.bmp\n*");
567     if (dialog.execute() & FXDialogBox::ID_ACCEPT) {
568         if (bgImage != NULL) {
569             delete bgImage;
570             bgImage = NULL;
571         }
572         bgImage = iconsrc->loadImageFile(dialog.getFilename());
573         if (bgImage == NULL) {
574             FXMessageBox::error(this, MBOX_OK, "Error", "Could not load file. (
                    Unsupported format?)");
575         } else {
576             bgImage->create();
577             handle(this, FXSEL(SEL_PAINT, ID_CANVAS), NULL);     // repaint
578         }
579     }
580     return 1;
581 }
582
583 long Triangulator::onRstBG(FXObject*, FXSelector, void*) {
584     // Remove background image
585     if (bgImage != NULL) {
586         delete bgImage;
587         bgImage = NULL;
588         handle(this, FXSEL(SEL_PAINT, ID_CANVAS), NULL);     // repaint
589     }
590
591     return 1;
592 }
593
594 long Triangulator::onExport(FXObject*, FXSelector, void*) {
595     FXFileDialog dialog(this, "Export to file...");
596     dialog.setDirectory(getenv("PWD"));
597     dialog.setPatternList("*.ps\n*");
598     if (dialog.execute() & FXDialogBox::ID_ACCEPT) {
599         string file (dialog.getFilename().text());
600         cout << "Save to file: " << file << endl;
601
602         PostScript out(file, view->getWidth(), view->getHeight());
603         for (unsigned i = 0; i < data.size(); i++) {
604             out.new_page();
605             out.write(data[i].name);
606
607             if (data[i].type == DS_POLYSET) {
608                 for (list<Polygon*>::iterator it = data[i].polygons.begin(); it !=
                        data[i].polygons.end(); it++) {
609                     out.draw(*it);
610                 }
611             } else if (data[i].type == DS_PARTITION || data[i].type ==
                    DS_TRIANGULATION || data[i].type == DS_CONVEX_PARTITION) {
612                 out.draw(data[i].partition, true);
613             } else if (data[i].type == DS_PATH) {
614                 out.draw(data[i].partition, true);
615                 out.draw(data[i].path);
616
617                 out.end_page();
```

```
618                 out.new_page();
619                 out.write(data[i].name);
620                 out.draw_part_outline(data[i].partition);
621                 out.draw(data[i].path);
622             }
623
624             out.end_page();
625         }
626     }
627     return 1;
628 }
629
630
631 // Updates
632
633 long Triangulator::onSelectItem(FXObject*, FXSelector, void*) {
634     if (selectedType() != DS_POLYSET && inputPoly) {
635         delete inputPoly;
636         inputPoly = NULL;
637     }
638     if (pathPtA && selectedType() != DS_TRIANGULATION && selectedType() !=
        DS_CONVEX_PARTITION) {
639         delete pathPtA;
640         pathPtA = NULL;
641     }
642
643     handle(this, FXSEL(SEL_PAINT, ID_CANVAS), NULL);     // repaint
644
645     DataSet *d = selected();
646     if (d) {
647         string s = d->name + "  -  ";
648         if (d->type == DS_POLYSET) s += "polygon set, " + tostr(d->polygons.size()
            ) + " polygons";
649         if (d->type == DS_PARTITION) s += "partition, " + tostr(d->partition->
            tiles.size()) + " tiles";
650         if (d->type == DS_TRIANGULATION) s += "triangulation, " + tostr(d->
            partition->tiles.size()) + " triangles";
651         if (d->type == DS_CONVEX_PARTITION) s += "convex partition, " + tostr(d->
            partition->tiles.size()) + " tiles";
652         if (d->type == DS_PATH) s += "path, " + tostr(d->partition->tiles.size())
            + "tiles, path length " + tostr(d->path->length());
653         viewTitle->setText(s.c_str());
654     } else {
655         viewTitle->setText("(no page selected)");
656     }
657
658     return 1;
659 }
660
661 long Triangulator::onUpdateList(FXObject*, FXSelector, void*) {
662     bool c = false;
663
664     if (pageList->getNumItems() > data.size()) {
665         c = true;
666         pageList->clearItems();
667     }
668
669     while (pageList->getNumItems() < data.size()) {
670         pageList->appendItem(data[pageList->getNumItems()].name.c_str());
671         c = true;
672     }
673     if (c) {
674         if (!data.empty()) {
675             pageList->killSelection();
```

```
676             pageList->setCurrentItem(pageList->getNumItems() - 1);
677             pageList->selectItem(pageList->getNumItems() - 1);
678         }
679         handle(this, FXSEL(SEL_CHANGED, ID_LIST), NULL);
680     }
681     return 1;
682 }
683
684 long Triangulator::onUpdateClear(FXObject* sender, FXSelector sel, void*) {
685     sender->handle(this,FXSEL(SEL_COMMAND,
686         (pageList->getNumItems() > 0 ? FXWindow::ID_ENABLE : FXWindow::ID_DISABLE)
687         ), NULL);
688     return 1;
689 }
690
691 long Triangulator::onUpdatePoly2part(FXObject* sender, FXSelector sel, void*) {
692     sender->handle(this,FXSEL(SEL_COMMAND,
693         (selectedType() == DS_POLYSET ? FXWindow::ID_ENABLE : FXWindow::ID_DISABLE
                )
694         ), NULL);
695     return 1;
696 }
697
698 long Triangulator::onUpdateEarClipT(FXObject* sender, FXSelector sel, void*) {
699     sender->handle(this,FXSEL(SEL_COMMAND,
700         (selectedType() == DS_PARTITION || selectedType() == DS_CONVEX_PARTITION
701               ? FXWindow::ID_ENABLE : FXWindow::ID_DISABLE)
702         ), NULL);
703     return 1;
704 }
705
706 long Triangulator::onUpdateDiagonalT(FXObject* sender, FXSelector sel, void*) {
707     sender->handle(this,FXSEL(SEL_COMMAND,
708         (selectedType() == DS_PARTITION || selectedType() == DS_CONVEX_PARTITION
709               ? FXWindow::ID_ENABLE : FXWindow::ID_DISABLE)
710         ), NULL);
711     return 1;
712 }
713
714 long Triangulator::onUpdateMonotoneT(FXObject* sender, FXSelector sel, void*) {
715     sender->handle(this,FXSEL(SEL_COMMAND,
716         (selectedType() == DS_POLYSET ? FXWindow::ID_ENABLE : FXWindow::ID_DISABLE
                )
717         ), NULL);
718     return 1;
719 }
720
721 long Triangulator::onUpdateDelaunayT(FXObject* sender, FXSelector sel, void*) {
722     sender->handle(this,FXSEL(SEL_COMMAND,
723         (selectedType() == DS_POLYSET ? FXWindow::ID_ENABLE : FXWindow::ID_DISABLE
                )
724         ), NULL);
725     return 1;
726 }
727
728 long Triangulator::onUpdateDelaunayPass(FXObject* sender, FXSelector sel, void*) {
729     sender->handle(this,FXSEL(SEL_COMMAND,
730         (selectedType() == DS_TRIANGULATION ? FXWindow::ID_ENABLE : FXWindow::
                ID_DISABLE)
731         ),NULL);
732     return 1;
733 }
734
735 long Triangulator::onUpdateConvexS(FXObject* sender, FXSelector sel, void*) {
```

```
736     sender->handle(this,FXSEL(SEL_COMMAND,
737         (selectedType() == DS_TRIANGULATION || selectedType() ==
                DS_CONVEX_PARTITION ? FXWindow::ID_ENABLE : FXWindow::ID_DISABLE)
738         ),NULL);
739     return 1;
740 }
741
742 long Triangulator::onUpdateShorten(FXObject* sender, FXSelector sel, void*) {
743     sender->handle(this,FXSEL(SEL_COMMAND,
744         (selectedType() == DS_PATH ? FXWindow::ID_ENABLE : FXWindow::ID_DISABLE)
745         ),NULL);
746     return 1;
747 }
748
749 int main(int argc, char *argv[]) {
750     FXApp app("Triangulator", "Alex AUVOLAT");
751
752     app.init(argc, argv);
753
754     new Triangulator(&app);
755
756     app.create();
757
758     return app.run();
759 }
```