UNIVERSITÉ PIERRE ET MARIE CURIE

Rapport Projet CPA

Géométrie Algorithmique

Auteur:
AFFES Mohamed Amin

Table des matières.

1 Introduction

Parmi les problèmes classiques de la géométrie algorithmique, celui de la collision d'objets géométriques. Ceci est un problème très complexe car il faut arriver à détecter une collision entre deux objets de la façon la plus précise possible, tout en ayant le meilleur temps d'exécution possible.

Le choix de l'objet géométrique est très important dans ce problème, et à une conséquence énorme sur la qualité de notre solution finale. En effet, si nous prenons l'exemple des jeux vidéos qui reflète très bien notre problématique, nous voulons avoir comme citée précédemment un meilleur temps d'exécution possible. Cependant, la qualité visuelle dans ce domaine est aussi exigé, donc prendre comme objet de très grands rectangles couvrants plusieurs pixels donnent un meilleurs temps d'exécution mais un manque de précision énorme et rendu visuel qui peut être médiocre.

Donc, la problématique est de choisir un objet permettant d'avoir le meilleur rapport entre le temps d'exécution et la qualité du résultat et degré de précision, qui doit être supérieur à une borne minimale exigé.

Dans le cadre de ce projet, nous étudions deux algorithmes :

- **Algorithme Toussaint** - Problème du rectangle minimum dans un nuage de point.
- **Algorithme Ritter** - Problème du cercle minimum dans un nuage de point.

Nous allons donc tout d'abord présenter chacun des deux algorithmes ainsi que leurs complexités et qualités, ensuite nous allons comparer la qualités des deux algorithmes et comparer leurs temps d'exécution. Puis pour finir effectuer une comparaison générale entre ces deux algorithmes pour notre problématique.

Les détails formules utilisés dans les différentes méthodes se trouvent en Annexe dans une section approprié.

Remarque:

Pour tester le code vous avez à votre disposition la classe Test.java dans le package test. Si vous appelez la méthode « Test() » de l'instance f dans le main Ritter, Graham et Toussaint sont lancées sur tous les fichiers de la base de test la qualité pour chaque fichier est affiché, le rectangle minimum sur tous les rectangles minimum de tous les fichiers est

affiché à la fin dans une fenêtre graphique, les qualités moyennes sont aussi affichés sur la console.

la méthode « Testfile(i) » de l'instance f effectue un affichage graphique correspondant au rectangle minimum et enveloppe convexe et cercle minimum du fichier i, vous pouvez appuyez sur le touche 'e' de votre clavier pour passer au fichier suivant.

2 Algorithme Toussaint

Nous allons aborder dans cette section, l'algorithme Toussaint permettant de trouver le rectangle minimum dans un nuage de points.

2.1 Calcul de l'enveloppe convexe - Graham

Cet algorithme commence par une phase de pré calcul. On prend pour un ensemble de points ayant la même abscisse **x** dans notre nuage de points, le point d'ordonné **y minimale** et le point d'ordonné **y maximale** ayant la même abscisse **x**, ou le point en lui même s'il est unique. On parcours notre ensemble de points, et on affecte deux tableaux. Le premier contenant les points d'ordonnés **y max** et le deuxième les points d'ordonnés **y min** comme cité au dessus. Ensuite on ajoute le tout dans une liste de sorte à ce qu'on ai les points triés **dans un sens anti-trigonométrique**.

Ensuite on parcourt la liste résultante du pré calcul précédent :
Soit p1, p2 et p3 **pour chaque itération i** tel que :
p1 = Point à la position i.
p2 = Point à la position ((i+1) % taille enveloppe).
p3 = Point à la position ((i+2) % taille enveloppe).

Soit un segment constitué des points **p1** et **p2** de l'enveloppe, si le résultat de son calcul vectoriel avec le point **p3** est négatif (c'est a dire que le point **p2** est inutile car on peux relier directement **p1** et **p3** qui couvrent ce point) on supprime le point **p2.** On effectue ceci jusqu'à la fin de la boucle parcourant la liste de points, afin d'avoir une enveloppe couvrant tous les points du nuage et ayant le minimum de cotés possibles.

La complexité de ce calcul est en O(n + n) = O(n). (car **O(n)** pour pré calcul et **O(n)** pour le traitement après le pré calcul).

L'implémentation de cet algorithme se trouve dans la classe **Graham.java** se trouvant dans le package « algorithms ».

2.2 Algorithme

Cet algorithme peut être brièvement décrit par les étapes suivantes :

1. Calculer l'enveloppe convexe.
2. Dessiner un rectangle contenant tous les points du nuage, et ayant un points de l'enveloppe convexe superposé avec chacun de ses côtés (*nord*, *sud*, *est* et *ouest*). Ces **quatre points** correspondent aux points les plus au *nord*, *sud*, *est* et *ouest* de notre enveloppe.
3. Ce rectangle est notre rectangle minimum actuel.
4. Effectuer **enveloppe.size()** fois :
    1. Calculer les 4 angles tel que présenté dans la **figure 1**.
    2. Effectuer une rotation des droites superposés sur les cotés de notre rectangle actuel et ayant donc le même vecteur directeur, suivant un ordre **anti-trigonométrique** de degré égal à celui de l'angle minimal parmi les angles précédents.
    3. Calculer les intersections entre ses droites pour dessiner les segments représentants le nouveau rectangle.
    4. Si l'aire du nouveau rectangle est inférieure à l'aire du rectangle minimum actuel, prendre le nouveau rectangle comme nouveau rectangle minimum.
    5. Enfin pour le points **p (i)** correspondant au points parmi les 4 points cités précédemment ayant l'angle selon lequel la rotation a été effectué, le remplacer par le points suivant dans l'enveloppe donc le point **p (i+1)**.
5. A la fin de cette boucle nous avons le rectangle minimum pour notre nuage de points qu'on retourne.

Un exemple d'une suite de rotations, et des dessins des rectangles retrouvés à chaque fois se trouve dans la **figure 2**.

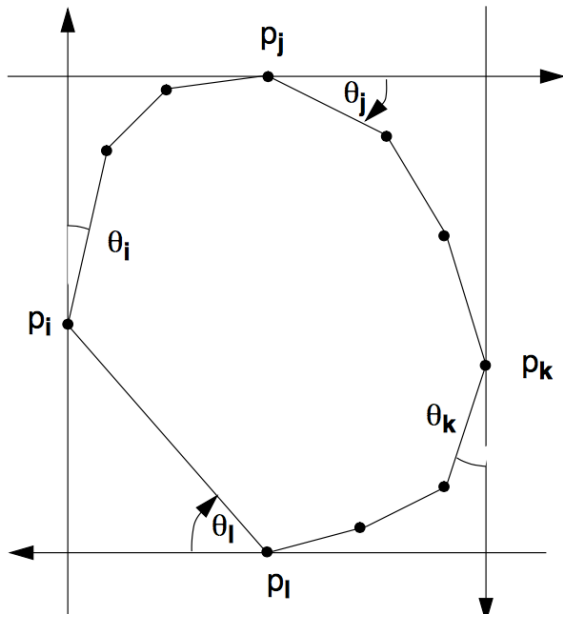


Figure 1

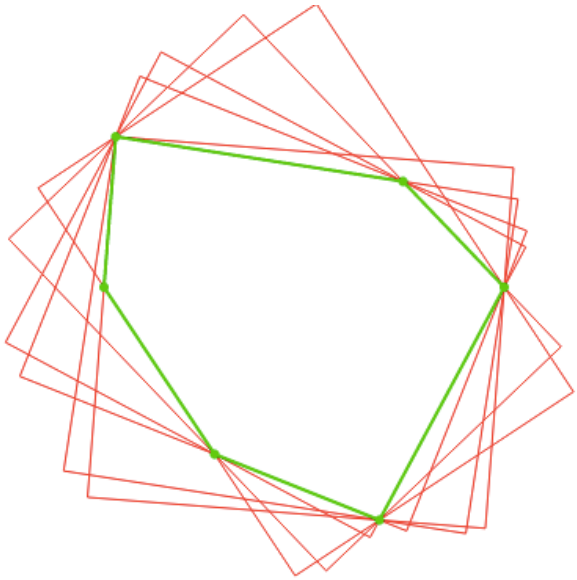


Figure 2

Voici le pseudo code de l'algorithme Toussaint décrit précédemment :

function Toussaint(List<Point> points)

1. **enveloppe <— Graham(points)**
2. **Pleft <— le point d'abscisse minimum de l'enveloppe**
3. **Pright <— le point d'abscisse maximum de l'enveloppe**
4. **Ptop <— le point d'ordonnée minimum de l'enveloppe**
5. **Pbottom <— le point d'ordonnée maximum de l'enveloppe**
6. **leftV <— droite passant par Pleft, vecteur directeur (0, -1)**
7. **rightV <— droite passant par Pright, vecteur directeur (0, 1)**
8. **topH <— droite passant par Ptop, vecteur directeur (1, 0)**
9. **bottomH <— droite passant par Pbottom, vecteur directeur (-1, 0)**
10. **rectangleMin <— Rectangle(Intersection(topH, leftV), Intersection(bottomH, leftV), Intersection(topH, rightV), Intersection(bottomH, rightV))**
11. **for j=0 to j=(enveloppe.size - 1)**
    1. **angleMin <— Min(θi, θj, θk, θl)**
    2. **rotationDroite(topH, -angleMin)** //angle négatif car sens anti-trigonométrique
    3. **rotationDroite(bottomH, -angleMin)**
    4. **rotationDroite(leftV, -angleMin)**
    5. **rotationDroite(rightV, -angleMin)**
    6. **rectangle <— Rectangle(Intersection(topH, leftV), Intersection(bottomH, leftV), Intersection(topH, rightV), Intersection(bottomH, rightV))**
    7. **if (Aire(rectangle) < Aire(rectangleMin)) then**
       1. **rectangleMin <— rectangle**
    8. **end if**

9. p correspondant angleMin <— suivant(enveloppe, p) //point suivant de la liste enveloppe
12. end for
13. return rectangleMin
end function

2.3 Complexité

Si l'on prend donc m la taille de l'enveloppe convexe la complexité de cet algorithme est en **O(m)** + la complexité de **Graham**. La complexité totale est donc celle de Graham **O(n)** + **O(m)** pour n le nombre de points du nuage de points et m le nombre de points de l'enveloppe convexe. Ce qui donne donc **O(n + m),** et puisque m <= n **donc la complexité finale est O(n).**

Je tient à préciser que dans la fonction Toussaint, toutes les fonctions appelés mis à part Graham ont une complexité **O(1)**.

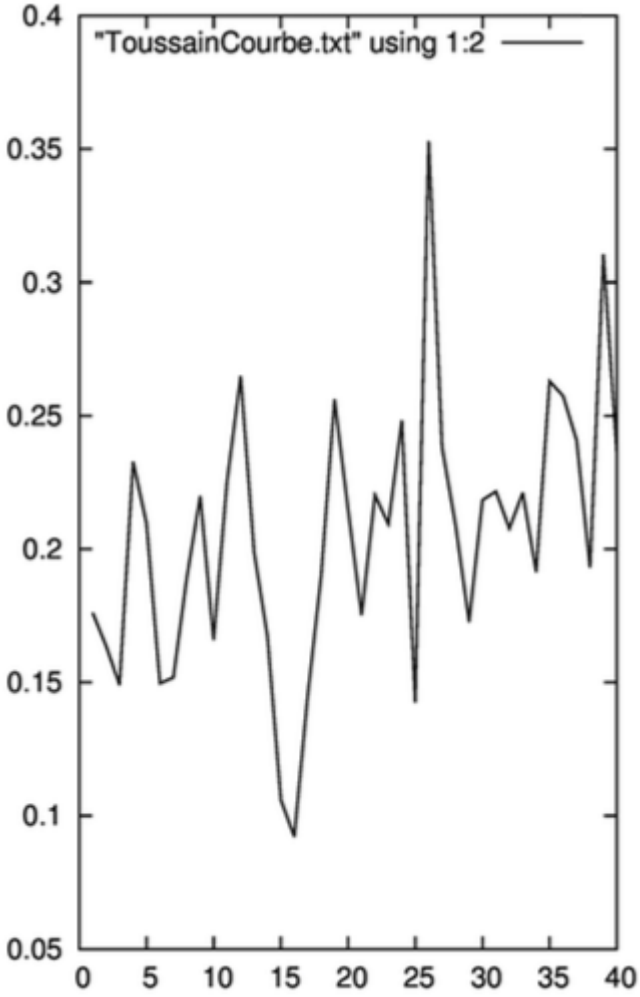
2.4 Qualité

la qualité moyenne (somme qualité pour chaque fichier / nombre de fichiers) est égale à 0.20264 soit de l'ordre de 20%. Donc **l'aire du rectangle minimum est de 20% plus grand que celui de l'enveloppe convexe**.

Ce résultat n'est pas étonnant, car pour qu'un rectangle puisse couvrir un rectangle convexe, il faut forcément que son air soit plus grand et que l'écart soit de cette importance. J'ai essayé d'améliorer cette qualité au maximum dans mon algorithme, en prenant des points de coordonnées décimales et non entières pour le vecteur directeur, ainsi que pour les points du rectangle et des segments, et enfin dans les calculs effectués afin d'avoir la meilleur exactitude possible. Cependant, je pense que le résultat reste quand même assez approximatif compte tenu de la difficulté d'avoir une précision optimale.

Voici une courbe représentant les différentes qualités calculés sur différents fichiers choisis aléatoirement, l'axe y représente la qualité calculé la courbe a été faite après calcul sur 40 fichiers distincts :

On peux voir sur cet exemple, que la qualité moyenne tourne autour de la valeur 0.2 ce qui correspond au calcul précédent effectué. Nous pouvons aussi remarquer que sur certains fichiers, cette valeur baisse et atteint même pour un fichier moins de 0.1 ! ce qui est une bonne qualité. Donc on peux constater que la qualité varie en fonctions de l'ensemble des points donnée, et donc peut être très bonne (proche de 0.1), comme elle peux être beaucoup moins bonne et atteindre 0.37 environs pour d'autres fichiers. cependant, la qualité moyenne reste acceptable et pas très lointaine de la meilleur valeur trouvé.



3 Algorithme Ritter

Nous allons aborder dans cette section, l'algorithme Ritter permettant de trouver le cercle minimum dans un nuage de points.

3.1 Algorithme

Le principe de cet algorithme est le suivant :

1. Prendre un point dummy quelconque appartenant à l'ensemble de points de départ.
2. Parcourir l'ensemble de points pour trouver un point P de distance maximum au point dummy.
3. Re-parcourir l'ensemble de points pour trouver un point Q de distance maximum au point P.
4. Considérer le point C, le centre du segment PQ.
5. Considérer le cercle centré en C, de rayon |CP| : il passe par P et Q.
6. Re-parcourir l'ensemble de points et enlever tout point contenu dans le cercle considéré.
7. S'il reste des points dans l'ensemble de points, considérer un tel point, appelé S.
8. Tracer la droite passant par S et C. Celle-ci coupe le périmètre du cercle courant en deux points : soit T le point le plus éloigné de S.
9. Considérer le point C', le centre du segment ST.
10. Considérer le cercle centré en C', de rayon |C'T| : il passe par S et T.
11. Répéter l'étape 6. Jusqu'à ce qu'il ne reste plus de points dans la liste.

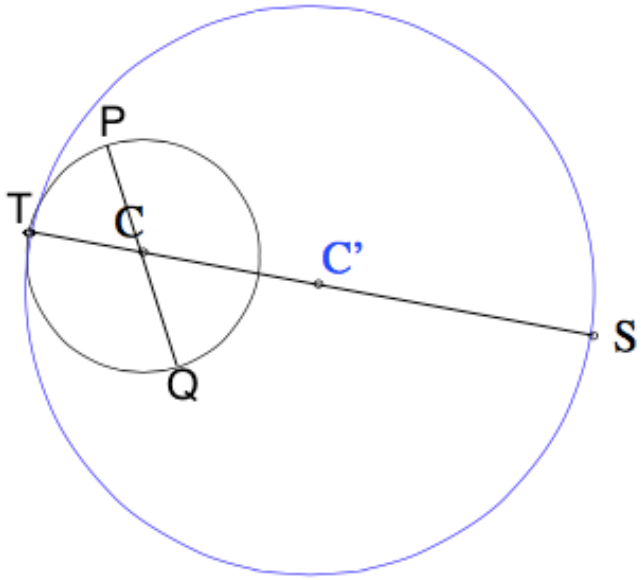


Figure 3

Voici le pseudo code de l'algorithme Ritter décrit précédemment :

function Ritter(List<Point> points)

1. **listeP <— points**
2. **dummy <— listP.head**
3. **P <— Lointain(listP, dummy)** //retourne le point de distance maximum du point dummy se trouvant dans la liste listP
4. **Q <— Lointain(listP, P)**
5. **C <— centre(P, Q)** //retourne le centre du segment PQ
6. **CR <— Circle(C, |PC|))** //retourne un cercle de centre c, et de rayon |PC|
7. **while (lp.IsNotEmpty)**
   1. **On supprime les points de la liste lp se trouvant dans le cercle CR**
   2. **if (lp.IsNotEmpty)**
      1. **S <— listP.head**
      2. **CPS <— ((rayon(CR) + |CS|) / 2)**
      3. **a <— CPS/ |CS|**
      4. **b <— (|CS| - CPS) / |CS|**
      5. **C <— a*C+ b*C**
      6. **lp.remove(S)**
      7. **CR <— Circle(C, |PC|))**
   3. **end if**
8. **end while**
9. **return Circle(c, rad)**

end function

3.2 Complexité

- Le calcul du point P est en **O(n)**.
- Le calcul du point Q est en **O(n)**.
- L'étape 7 de notre pseudo code est en **O(m)**, avec m < n car pour le premier tour de boucle au moins deux points sont supprimés (les deux points P et Q appartenants au premier cercle CR). Ensuite pour les tours de boucle suivants, au moins un point est supprimé à chaque tour de boucle.
- L'étape 7.1 de notre pseudo code est en **O(n)**.

Nous avons donc **O(3n * m) = O(nm)** avec m < n. **La complexité de cet algorithme est donc en O(n).**

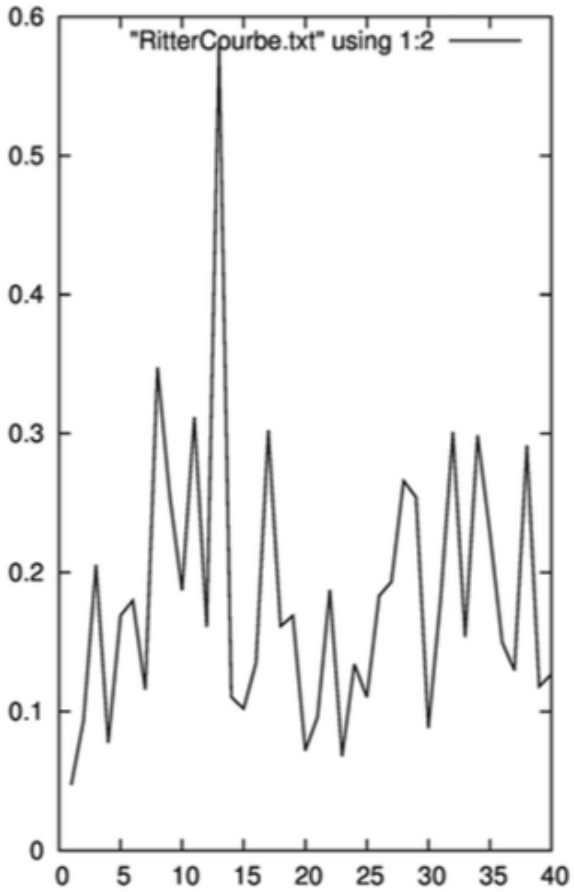
3.3 Qualité

La qualité moyenne est de 0.17804757959324974 soit de l'ordre de 17,8%. Donc **l'aire du cercle minimum est de 17,8% plus grand que celui de l'enveloppe convexe**.

Ceci est un bon résultat quant au fait que notre algorithme est une approximation, mais elle permet quand même d'avoir une qualité assez bonne même si elle n'est pas optimale.

Voici une courbe représentant les différentes qualités calculés sur différents fichiers choisis aléatoirement, l'axe y représente la qualité calculé la courbe a été faite après calcul sur 40 fichiers distincts:

On peux voir sur cet exemple que la qualité moyenne est de 0,18 environs ce qui correspond au calcul précédent effectué, nous pouvons aussi remarquer que sur certains fichier cette valeur baisse et atteint même pour un fichier moins de 0.05 ! ce qui est une excellente qualité. Donc on peux constater que comme pour l'algorithme Toussaint, ici aussi la qualité varie en fonctions de l'ensemble des points donnée, et donc peut être bonne proche de 0.05 comme elle peux être beaucoup moins bonne et atteindre 0.57 environs pour d'autres fichier. Cependant la qualité moyenne reste acceptable et n'est pas très loin de la meilleur valeur trouvé.



4 Algorithmes Toussaint/Ritter

Cette section traite la comparaison entre ces deux algorithmes, afin de voir lequel peux correspondre le mieux, afin d'avoir la meilleur rapport entre le temps d'exécution et la qualité du résultat et degré de précision.

4.1 Comparaison des qualités

Quant tenu des qualités moyennes trouvés pour chaque algorithme dans les sections précédentes. Nous pouvons voir que l'algorithme Ritter est meilleur que l'algorithme Toussaint en terme de qualité, car sa qualité moyenne est inférieure à celle de Toussaint. Ce qui permet donc d'avoir une meilleur précision lors de l'étude des collisions par exemple, car l'aire du cercle est plus proche de l'aire de l'enveloppe convexe que l'aire du rectangle minimum.

Nous pouvons voir aussi que le meilleur des cas pour notre jeu de test donne l'avantage à Ritter aussi (0,05 conte 0,1 pour Toussaint), cependant le pire cas donne l'avantage à Toussaint avec une assez grande avance (0,37 contre 0,57 pour Ritter). Cependant c'est la moyenne qui nous donne le résultat qui nous intéresse et duquel on peux donner le jugement final donnée dans la paragraphe précédent.

Donc en moyenne la qualité de l'algorithme Ritter est meilleure, et donne donc plus de précision pour le calcul des collisions.

4.1 Temps d'exécution

La complexité des deux algorithmes est en **O(n)**, donc déjà en terme de complexité temporelle les deux algorithmes sont égaux.

En regardant de plus près nous pouvons voir que pour un grand nombre de points, l'algorithme Toussaint est plus rapide. Ceci car toutes les opérations effectués dans une boucle dans cet algorithme s'effectuent en **O(1)**, ceci que ça soit pour la fonction Graham ou le reste de l'algorithme. Or pour l'algorithme Ritter, nous effectuons une deuxième boucle imbriqué dans un première boucle, s'exécutant tant que la liste de points n'est pas vide. Donc le total de cette première boucle ayant une complexité **O(m*n)** qui est

transformé en **O(n)** vu que m<n. Cependant ce facteur « **m** » intervient lorsque les données sont très grandes, donc lorsque nous avons un très grand nombre de points ce facteur « **m** » fait de Ritter un algorithme plus lent que Toussaint.

5 Conclusion

Nous avons donc vu que l'algorithme Toussaint est en moyenne moins bon en terme de qualité que l'algorithme Ritter, et donc manque de précision par rapport à ce dernier. Cependant, en terme de vitesse d'exécution sur un nombre très importants de données (points), l'algorithme Toussaint est plus rapide que Ritter. Et la différence entre les deux qualités n'est pas très grande.

Pour conclure, On utilise l'algorithme Ritter si le nombre de points n'est pas très important, sinon on utilise l'algorithme Toussaint qui donne un résultat ayant une qualité légèrement inférieure à celle de Ritter mais plus rapide. Et pour un grands nombre de points tel que dans les jeux vidéos le temps d'exécution, tel que le temps de réaction par exemple à l'attaque d'un adversaire, est beaucoup plus importante que ce léger écart de précision.

Annexe

Définitions et formules

Dans cette section, je défini quelques définitions et formules mathématiques permettant de comprendre l'implémentation de mon algorithme afin de pouvoir bien comprendre l'implémentation de cet algorithme citée dans la suite.

- Un cercle est une courbe plane fermée constituée des points situés à égale distance d'un point nommé centre. La valeur de cette distance est appelée rayon du cercle. Celui-ci étant infiniment variable, il existe donc une infinité de cercles pour un centre quelconque, dans chacun des plans de l'espace.

La classe **Circle** dans le package « tools » joue ce rôle dans ce projet.

- Une ligne ici est une droite théoriquement infini tracé ayant un Point contenue dans cette droite et un vecteur directeur qui est un point sur un plan 2D ayant donc deux coordonnés et définissant le sens de cette droite dans une plan 2D.

La classe **Ligne** dans le package « tools » joue ce rôle dans ce projet.

- Un Segment est une ligne tracé reliant deux points dans un plan.

La classe **Segment** dans le package « tools » joue ce rôle dans ce projet, elle contient deux attributs a et b représentant les deux points.

- Un Rectangle est un quadrilatère dont les quatre angles sont droits, ici un rectangle est constitué de quatre segments chaque segment se croise avec un autre segment dans chacun de ses points.

La classe **Rectangle** dans le package tools joue ce rôle dans ce projet.

- La distance entre deux points dans un plan 2D :

$$AB = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}.$$

- Le produit vectoriel entre un segment constitué de deux points ‘p’ et ‘q’ et un point ‘r’:

pv = ((q.x - p.x) * (r.y - p.y)) - ((q.y - p.y) * (r.x - p.x))

La méthode **produit_vectoriel** dans la classe **Graham.java** dans le package « algorithms » implémente cette formule.

- Le calcul d’un angle entre deux droites (ici deux instances de la classe Ligne) :

Soit v1 le vecteur directeur de la droite 1 et v2 le vecteur directeur de la droite 2, le résultat est donc :

angle = |atan(((v1.y * v2.x - v2.y * v1.x) / (v1.x * v2.x + v1.y * v2.y)))|

- La rotation d’une droite d’un certain angle s’effectue comme suit :

Soit v le vecteur directeur de la droite, angle l’angle de rotation et deux valeurs décimales ‘x’ et ‘y’, et ‘vn’ le nouveau vecteur directeur de la droite après la rotation.

x = (v.x * cos(angle)) + (v.y * sin(angle))
y = (-(v.x * sin(angle))) + (v.y * cos(angle))
vn = Un nouveau point de coordonnée x et y

- L’intersection entre deux droites l1 et l2 :

Soit la valeur décimale ’t’ et et les points ‘q’, ‘p’, ‘a’, ’s’, ‘r’, ’n’ avec n le point d’intersection entre les deux droites:

q = un point de coordonnées (l1.PointDroite.x, l1.PointDroite.y)
p = un point de coordonnées (l2.PointDroite.x, l2.PointDroite.y)
a = un point de coordonnées (q.x - p.x, q.y - p.y)
s = le vecteur directeur de l1
r = le vecteur directeur de l2
t = (a.x * s.y - a.y * s.x) / (r.x * s.y - r.y * s.x)
n = nouveau point de coordonnées ((p.x + (t * r.x)) , (p.y + (t * r.y)))

References

Dans cette section je présente certains liens qui m'ont servis à retrouver quelques formules mathématiques ou quelques indications pour l'implémentation de l'algorithme.

Calcul de l'angle entre deux droites :
http://integraledesmaths.free.fr/idm/PagePrincipale.htm#http://integraledesmaths.free.fr/idm/GeoAPAngDro.htm

Calcul du point d'intersection de deux segments :
http://openclassrooms.com/forum/sujet/calcul-du-point-d-intersection-de-deux-segments-21661

Rotation vectorielle :
http://fr.wikipedia.org/wiki/Rotation_vectorielle

Toussaint :
http://web.cs.swarthmore.edu/~adanner/cs97/s08/pdf/calipers.pdf
https://geidav.wordpress.com/2014/01/23/computing-oriented-minimum-bounding-boxes-in-2d/