Groupe 584I X31I020

Projet Golf

Algorithmique et Structures de Données 3

LE BADEZET Benoît
PLOUHINEC Glenn

Description du projet:

Le but de ce projet était de programmer un jeu de golf, avec un terrain, des tracés :
Un terrain est représenté par une liste de polygones délimitants les différentes surface de jeu (fairway, green, rough, bunker, water, etc), et une liste contenant les différents Tracés du Terrain.

Un tracé (ou un Trou) est représenté par les fairways et le green qu'il contient (les polygones) ainsi que le point de départ, le point d'arrivé et le nombre de coups pour faire un Par.
Un polygone est représenté par une liste de points, sa couleur (la surface).
Un triangle hérite de polygone mais possède un nombre de point bien défini, soit 3.
Un carré hérite de polygone mais possède un nombre de point bien défini, soit 4.

Pour mener à bien ce projet nous avons dû réaliser un quadtree, qui est une structure de données permettant de partitionner un espace en deux dimensions, en le découpant récursivement en quatre noeuds. Nous avons aussi réalisé une triangulation de polygones concaves et convexes afin de simplifier au mieux la recherche de la balle sur le terrain.

Un affichage graphique a été produit pour mieux visualiser le jeu de golf et les méthodes faisant appel à des notions géométriques. De plus, des méthodes semblables à un analyseur syntaxique permettent de lire les fichiers "DescriptionFIgureGolf2.txt", "TestGolf3.txt" , "TestGolfConcave.txt", "TestGolfConvexe.txt" ont été implémentées.

Ayant rencontré de nombreuses difficultés pour les méthodes de triangulation du terrain de golf, et d'autres problèmes d'approximations des nombres de type double, nous n'avons pas pu terminer ce projet entièrement. La triangulation fonctionne désormais correctement, mais les problèmes d'approximations intervenant dans le calcul des intersections de segments notamment, nous ont donné bien du mal et causé de nombreux bogs. Ainsi, après avoir terminé le TP2, nous avons pu effectuer une représentation graphique du terrain, lire dans les fichiers, trianguler le terrain, et construire le quadtree. Mais aucune représentation du golfeur, des scores ou de la balle n'a pu être faite.

Compilation et exécution :

La compilation et l'exécution de notre jeu de golf s'effectue, en se positionnant dans le dossier golf/ et en tapant dans un terminal :

javac java/*.java -d class/ && java -cp class NomduFichier.txt

Le menu offrant toutes les modalités des jeux de tests n'ayant pas été terminé ce dernier offre malgré tout la possibilité en tapant :

0 : Permet de fermer l'application

1 : Afficher graphiquement la triangulation du terrain
2 : Afficher la représentation graphique du quadtree
3 : Permet de tester la méthode RecherchePointTriangle()
4 : Permet d'afficher le premier tracé
5 : Permet d'afficher le second tracé s'il existe
6 : Permet de réinitialiser l'affichage

Note : Une triangulation d'un polygone *P* est une partition de *P* en un ensemble de triangles qui ne se recouvrent pas, et dont l'union est *P*. On peut donc trianguler tous les polygones d'un terrain en une liste de triangles.

I - Les classes

Point :

Un objet Point est composé de deux attributs de type double : **x** et **y**, qui représentent ses coordonnées Abscisse et Ordonnée

Polygone :

Un objet Polygone est composé d'un ArrayList de Point, **sommets**, délimitant la surface du polygone, et un caractère **col** désignant la couleur (et sa surface) de ce polygone.
Cette classe contient une méthode de triangulation qui retourne la liste des triangles qui partitionnent ce polygone.
Deux classes héritent de Polygone, il s'agit des classes Triangle et Carré, leur nombre de Point est juste limité (3 pour les triangles et 4 pour les carrés).

Tracé :

Un objet tracé est composé d'un ArrayList de Polygone **fairgreen** représentants ses fairways et son green, ainsi que deux point **depart** et **trou** représentants les coordonnées de départ et d'arrivée du tracé, et un entier **par** qui représente le nombre de coup pour réussir un par.

Terrain :

Un objet terrain est composé d'un ArrayList de Polygone **polygones** représentant toute la surface de jeu, ainsi qu'un ArrayList de Tracé **traces** contenant tous les tracés disponibles sur ce terrain.
Cette classe contient une méthode de lecture de fichier afin de créer un terrain de manière plus aisée.

Quadtree :

Un objet Quadtree est composé d'un Point **origine** et d'un double **taille** qui permettent de déterminer la surface de la région. Quadtree contient aussi un Arraylist de Quadtree **noeud** contenant ses fils s'il en existe ainsi qu'un ArrayList de Triangle **triangles** constituant tous les triangles qui sont dans sa région.

II - Les fonctions principales

La fonction TriangulationTerrain()

```
/* Dans la classe Terrain */

fonction TriangulationTerrain() : ArrayList<Triangle>
Début
        ArrayList<Triangle> res  <--  new ArrayList<Triangle>()

        Pour tout Polygone p de polygones faire
            res.addAll(p.triangule())
        fin Pour

        retourner res
Fin



/* Dans la classe Polygone */

fonction triangule() : ArrayList<Triangle>
Début
    ArrayList<Triangle> res  <--  new ArrayList<Triangle>()
    Object o  <--  sommets.clone()
    ArrayList<Point> clone  <-- (ArrayList<Point>)o
    int ind
    int modulo

    Tant Que clone.size() > 3 faire
      modulo  <--  clone.size()
      ind  <-- trouveOreille(clone)
      res.add(new Triangle(clone.get((ind-1+modulo) mod modulo), clone.get(ind), clone.get((ind+1) mod modulo), col ) )
      clone.remove(ind)
    fin TQ
    res.add(new Triangle(clone.get(0), clone.get(1), clone.get(2), col))
    retourner res
Fin
```

```
37  fonction trouveOreille(ArrayList<Point> p) : int
38  Début
39      boolean oreille  <--  FAUX
40      int k  <--  p.size()
41      int i  <--  0
42      int j
43      Segment s
44      Droite d
45      Segment sj
46      Triangle t
47
48      Tant Que i < k ET oreille = FAUX faire
49          d  <--  new Droite(p.get((i-1+k) mod k), p.get((i+1) mod k) )
50          s  <--  new Segment(p.get((i-1+k) mod k), p.get((i+1) mod k))
51          t  <--  new Triangle(p.get((i-1+k) mod k),p.get(i), p.get((i+1) mod k))
52          j  <-- (i+2) mod k
53          oreille  <--  (d.appartient(p.get(i)) = -1)
54          oreille  <--  oreille ET (t.appartient(p.get((i+2) mod k)) = -1)
55          oreille  <--  oreille ET (t.appartient(p.get((i-2+k) mod k)) = -1)
56
57          Tant Que oreille = true ET (j+2) mod k ≠ i  faire
58              oreille  <--  oreille ET (t.appartient(p.get(j mod k)) = -1)
59              sj  <--  new Segment(p.get(j mod k), p.get((j+1) mod k))
60              oreille = oreille ET (s.inter(sj) = null)
61              j  <-- j + 1
62          Fin TQ
63
64        i <-- i + 1
65      Fin TQ
66      retourner (i-1+k) mod k;
67  Fin
```

Fonctionnement : Pour chaques polygone, avec la méthode *triangule*, tant qu'il nous reste plus de 4 sommets, on va retirer l'oreille préalablement trouvée avec le méthode *trouveOreille* qui va tester si, pour un point i du polygone, le segment entre le point précédent et le point suivant i, est a l'intérieur de ce polygone et n'est pas coupé par un autre segment de ce polygone. On stocke tous ces triangles dans un ArrayList que l'on retourne a la fin.
Cet algorithme est correct pour tous les cas que nous avons pu tester, cependant il est possible qu'à cause d'une erreur d'approximation du point d'intersection de deux segments, l'algorithme trouve une oreille erronée car un segment secant existerait mais ne serait pas vu.

Complexité : $O(m*n^3)$ avec m le nombre de polygone et n la taille (le nombre de points) des polygone
Cette complexité n'est pas optimale du fait que nous utilisons deux fonctions indépendantes qui repartirons du début du polygone a chaques appel, on pourrait descendre jusqu'à du $O(n^2)$ avec la méthode des oreille voir du $O(n \log(n))$ avec une autre façon de trianguler.

La méthode ConstructionQT()

```
Procédure ConstructionQT(ArrayList<Triangle> liste, Quadtree arbre)
Début
    arbre.origine <-- new Point(0,0);
    arbre.taille <-- Constantes.nbCases;
    arbre.noeuds <-- new ArrayList<Quadtree>();
    arbre.triangles <-- new ArrayList<Triangle>();
    arbre.ajouterListeTriangle(liste, arbre);
Fin

Procédure ajouterListeTriangle(ArrayList<Triangle> liste, Quadtree arbre)
Début
    Pour i allant de 1 à liste.size() faire
        arbre.ajouterTriangle(liste.get(i), arbre);
    Fin Pour
Fin

Procédure ajouterTriangle(Triangle t, Quadtree arbre)
Début
    Si arbre.estFeuille() alors

        Si arbre.TestRegionIntersecteTriangle(t, arbre) OU arbre.TriangleDansRegion(t, arbre) alors
            arbre.triangles.add(t);
        Fin si

        Si arbre.triangles.size() >= NbTrianglesMax ET arbre.profondeur < ProfondeurMax alors
            X <-- arbre.origine.getX();
            Y <-- arbre.origine.getY();
            nvTaille <-- arbre.taille/2;

            Quadtree zone1 <-- new Quadtree(new Point(X, Y + nvTaille), nvTaille, arbre.triangles);
            Quadtree zone2 <-- new Quadtree(new Point(X + nvTaille, Y + nvTaille), nvTaille, arbre.triangles);
            Quadtree zone3 <-- new Quadtree(new Point(X + nvTaille, Y), nvTaille, arbre.triangles);
            Quadtree zone4 <-- new Quadtree(new Point(X, Y), nvTaille, arbre.triangles);
            arbre.noeuds.add(zone1);
            arbre.noeuds.add(zone2);
            arbre.noeuds.add(zone3);
            arbre.noeuds.add(zone4);

            arbre.triangles.clear();
        Fin Si
    Sinon
        Pour tout Quadtree q de arbre.noeuds faire
            q.ajouterTriangle(t);
        Fin Pour
    Fin Si
Fin
```

Fonctionnement : Une fois le terrain triangulé, on passe en paramètre l'ensemble des triangles obtenus à la fonction ConstructionQT(), celle-ci fait alors appel à ajouterListeTriangle(), qui nous permet d'ajouter chaque triangle au quadtree un à un. On vérifie alors dans un premier temps si le triangle se trouve dans la région du quadtree en se servant implicitement de méthodes pour vérifier si un triangle se trouve dans un carré, ou bien si un triangle intersecte un carré, le carré représente alors la région du quadtree, et on ajoute le triangle à la liste des triangles du quadtree courant si ces conditions sont alors vérifiées. Dans un second temps, on vérifie alors si le nombre de triangles contenu dans la liste est inférieure à la constante nommée ici "NbTrianglesMax", et que l'arbre ne soit pas trop profond, on contrôle la profondeur de l'arbre car il est possible que sans cette vérification, l'arbre s'allonge continuellement s'il existe de trop nombreux triangles avec un point commun et surcharge la mémoire inutilement. Suite à cette vérification, on effectue donc un découpage du quadtree s'effectuant selon les coordonnées du Quadtree courant, en faisant appel à un constructeur prenant en paramètre un point d'origine, une taille, et la liste courante qui va alors être de nouveau répartie parmi les 4 nouveaux quadtrees, et on vide ensuite la liste afin d'éviter de surcharger la mémoire.

L'algorithme semble à première vu correct, seulement, au cours de quelques tests durant le projet, on s'aperçoit que quelques triangles manquent à l'appel, l'explication la plus

plausible serait un problème dû aux approximations de la fonction testant l'intersection de deux segments. Ce problème se règle parfois en réduisant la constante de ProfondeurMax.

Complexité : O(nbTriangles*ProfondeurMax*NbTrianglesMax)
On ne peut sans doute pas faire plus optimal.

La méthode CalculCoefficientsDroite()

```
1  Procédure CalculCoefficientsDroite(Point p1, Point p2, Droite d)
2  Début
3      Précondition : p1 != p2
4
5      d.a <-- (p2.getY() - p1.getY()) * (-1);
6      Si d.a = -0.0 alors
7          d.a <-- 0;
8
9      d.b <-- p2.getX() - p1.getX();
10     Si d.b = -0.0 alors
11         d.b <-- 0;
12
13     d.c <-- (p1.getX()*a + p1.getY()*b) * (-1);
14     Si d.c = -0.0 alors
15         d.c <-- 0;
16 Fin
```

Fonctionnement : On utilise le vecteur directeur de la droite avec p1 et p2 pour déterminer a et b,quant à c, il se détermine avec un des deux point (ici p1) en résolvant l'équation
$-c = a * x + b * y$
L'algorithme proposé est correct car utilise des notions mathématiques vues au lycée, et qu'il n'y a pas de division par 0.

Complexité : O(1)
On ne peut pas faire plus optimal.

La méthode TestIntersectionDeuxSegments()

```
1  Fonction TestIntersectionDeuxSegments(Segment s1, Segment s2): Point
2  Début
3
4      Droite d1 <-- new Droite(s1.p1, s1.p2);
5      Droite d2 <-- new Droite(s2.p1, s2.p2);
6      Point intersection <-- d1.intersection(d1);
7
8      Si s1.appartient(intersection) ET s2.appartient(intersection) alors
9        Si s1.p1 = intersection alors
10         Ecrire(C'est p1);
11       Sinon si s1.p2 = intersection alors
12         Ecrire(C'est p2);
13       Fin si
14       retourner intersection;
15     Fin si
16
17     retourner null;
18
19 Fin
```

Fonctionnement : Pour déterminer le point d'intersection on utilise l'intersection des deux droites qui prolongent ces segments pour calculer leur intersection a l'aide de l'équation cartésienne, ensuite on teste si ce point appartient à ces deux segments en vérifiant si les coordonnées de ce point sont comprises entre celles de ces segments.

Cet algorithme est correct, néanmoins lors du calcul de l'intersection de droites, les divisions multiples peuvent provoquer une erreur d'arrondis pour les coordonnées du Point qui ne sera alors pas sur le segment à cause d'un décalage sur les valeurs. Malgré le temps passé à réfléchir à une solution pour ce problème, nous n'en avons pas trouvé d'efficace.

Complexité : O(1)
On ne peut pas faire plus optimal.

La méthode TestRegionIntersecteTriangle()

```
1  Fonction TestRegionIntersecteTriangle(Triangle t, Quadtree arbre): boolean
2  Début
3      Carre c  <-- new Carre(arbre.origine, arbre.taille);
4      return TriangleIntersecteCarre(t, c);
5  Fin
6
7  Fonction TriangleIntersecteCarre(Triangle t, Carre c): boolean
8  Début
9      Segment ab <-- new Segment(c.getPoint(0), c.getPoint(1));
10     Segment bc <-- new Segment(c.getPoint(1), c.getPoint(2));
11     Segment cd <-- new Segment(c.getPoint(2), c.getPoint(3));
12     Segment da <-- new Segment(c.getPoint(3), c.getPoint(0));
13
14     res <-- 0;
15     res <-- res + TriangleIntersectionSegment(ab, t);
16     res <-- res + TriangleIntersectionSegment(bc, t);
17     res <-- res + TriangleIntersectionSegment(cd, t);
18     res <-- res + TriangleIntersectionSegment(da, t);
19
20     retourner res >= 2;
21 Fin
```

Fonctionnement : Nous utilisons la méthode *TriangleIntersecteCarre* qui va incrémenter l'entier *res* du nombre d'intersections entre chaques segment du carré et chaque segment du triangle (via la méthode *TriangleIntersectionSegment*), on retourne un booléen qui dépend de si au total on a deux ou plus intersections entre les deux figures. On évite les cas spéciaux tel que "le sommet du carré ou triangle est en contact avec un côté" en n'incrémentant *res* que d'un au lieu de deux dans ces cas.

Cette algorithme est correct car avec de nombreux tests nous n'avons pas rencontré d'erreur, cependant l'approximation d'intersection de segment peut encore une fois poser problème.

Compléxité : O(1)
On ne peut pas faire plus optimal.

La méthode TestTriangleContientPoint()

```
Fonction TestTriangleContientPoint(Point p, Triangle t): entier
Début
    Droite ab <-- new Droite(t.getPoint(0), t.getPoint(1));
    Droite bc <-- new Droite(t.getPoint(1), t.getPoint(2));
    Droite ca <-- new Droite(t.getPoint(2), t.getPoint(0));

    Segment AB <-- new Segment(t.getPoint(0), t.getPoint(1));
    Segment BC <-- new Segment(t.getPoint(1), t.getPoint(2));
    Segment CA <-- new Segment(t.getPoint(2), t.getPoint(0));

    //Détermine si le point est strictement à l'intérieur du triangle
    boolean inter <-- ab.PointAppartientDroite(p) = ab.PointAppartientDroite(t.getPoint(2));
    inter <-- inter ET bc.PointAppartientDroite(p) = bc.PointAppartientDroite(t.getPoint(0));
    inter <-- inter ET ca.PointAppartientDroite(p) = ca.PointAppartientDroite(t.getPoint(0));

    Si p = t.getPoint(0) OU p = t.getPoint(1) OU p = t.getPoint(2) alors
        Ecrire(Le point est sur un sommet);
        retourner 0;
    Sinon si AB.PointAppartientSegment(p) OU BC.PointAppartientSegment(p) OU CA.PointAppartientSegment(p) alors
        Ecrire(Le point est sur un segment);
        retourner 1;
    Sinon si inter alors
        Ecrire(Le point est dans le triangle);
        retourner 2;
    Sinon
        Ecrire(Le point est en dehors du triangle);
        retourner -1;
    Fin si
Fin
```

Fonctionnement : Nous déterminons si p est égal à un sommet du triangle (grâce à la redéfinition de *equals*), si oui nous retournons 0, ensuite nous vérifions si ce point appartient à un des segments du triangle, si oui nous retournons 1, enfin, pour chaque droite formé par les côtés du triangle, si le point p appartient au même demi-plan que le troisième sommet, le point se trouve alors dans le triangle et on retourne 2. Si nous ne sommes dans aucun de ces cas alors p n'appartient pas au triangle et nous retournons alors -1.
Cet algorithme est correct car ce sont des mathématiques basiques.

Complexité : O(1)
On ne peut pas faire plus optimal.

La méthode RecherchePointQT(Point p)

```
fonction RecherchePointQT(Point p) : Quadtree
Début
    Si estFeuille()  alors
        retourner this;
    Sinon
        Si p.getX() <= (origine.getX() + taille/2)  alors          //recherche zone 1 et 4
            Si p.getY() >= (origine.getY() + taille/2)  alors      //recherche zone 1
                retourner noeuds.get(0).recherchePointQT(p)
            Sinon                                                  //recherche zone 4
                retourner noeuds.get(3).recherchePointQT(p)
            Fin Si

        Sinon                                                      //recherche zone 2 et 3
            Si p.getY() >= (origine.getY() + taille/2)             //recherche zone 2
                retourner noeuds.get(1).recherchePointQT(p)

            Sinon                                                  //recherche zone 3
                retourner noeuds.get(2).recherchePointQT(p)
            Fin Si
        Fin Si
    Fin Si
Fin
```

Fonctionnement : On effectue ici une recherche récursive du point en fonction de ses coordonnées : on détermine s'il se situe à droite ou à gauche du quadtree (respectivement les zones 1-4, et 2-3) en fonction de l'axe X, et on affine la recherche pour savoir s'il se trouve plutôt en bas ou en haut dans le quadtree, selon l'axe Y.
Cet algorithme est correct et très efficace, car il possède une complexité moindre, et d'après nos tests il trouve toujours la zone correspondant au point recherché.

Complexité : O($log_4(n)$)

La méthode RecherchePointTriangle(Point p)

```
1  fonction RecherchePointTriangle(Point p) : Triangle
2  Début
3          ArrayList<Triangle> NvListeTri  <--  new ArrayList<Triangle>()
4          Triangle tri
5
6          Pour i allant de 0 à triangles.size() - 1   faire
7              Si triangles.get(i).appartient(p) ≠ -1  alors
8                  NvListeTri.add(triangles.get(i))
9              Sinon
10         Fin Pour
11
12         Si NvListeTri.size() = 1   alors
13             tri  <--  NvListeTri.get(0);
14         Sinon
15             tri  <--  prioriteMax(NvListeTri)       // Retourne le triangle le plus prioritaire en fonction de la couleur
16         Fin Si
17         retourner tri
18 Fin
```

Fonctionnement : On recherche dans les triangles occupant la région lesquels possèdent le point p (plusieurs possibles si p sur un sommet ou un côté) et on détermine lequel de ces triangle est le plus prioritaire en fonction de sa couleur (surface).
Le fait que cet algorithme est correct découle du fait que les méthodes qu'il utilise sont correctes elles aussi.

Complexité : O(n) avec n le nombre de triangles présents dans la région
Un ne peut pas faire plus optimal car nous devons vérifier dans tous les triangles ce qui demande un parcours.

Conclusion :

Ce projet nous a permis de mettre en pratique des notions vues en cours telles que le Quadtree vu en CM et TD, cependant leur implémentation n'est pas si facile, en effet nous nous sommes heurtés à quelques difficultés et contraintes qui on fait que nous avons manqué de temps pour terminer ce projet comme nous l'aurions souhaité.

Hormis les difficultés, nous avons eu l'occasion d'améliorer notre esprit d'équipe en échangeant avec les autres groupes de travail, ce qui nous a permis d'éclaircir notre point de vue sur certains problèmes.

La plus grande difficulté inter-binôme étant un conflit sur l'indentation du code, on peut remarquer que certaines parties sont parfois codées avec une indentation différente, et de nombreuses autres ont été corrigées par un l'un de nous.