Compte-rendu de projet – Golf

PELLOQUIN Rémi & PETIT Marjorie

1) Introduction

Le but du projet était d'implémenter un jeu de golf en utilisant une structure de quadtree afin de pouvoir identifier efficacement le déplacement de la balle lors de la partie. Ainsi, nous avons implémenté cette structure et de nombreuses méthodes demandées pour coller au mieux au sujet. Nous avons également gérer la lecture des fichiers .txt afin de pouvoir lire le terrain de golf et l'exploiter dans une partie.

2) Commandes de compilation et exécution en mode console

Tout d'abord, nous devions pouvoir lancer notre programme depuis un terminal sans passer par eclipse, ainsi, nous avons suivie les étapes ci-dessous :

Aller dans le dossier src du projet depuis un terminal
Bien mettre le fichier .txt à lire dans src
Puis taper « Javac Main.java » pour compiler
Et enfin, lancer avec la commande « java Main »

On aurait pu également exporter le projet depuis eclipse (File/Export) en JAR. Puis lancer le projet depuis un terminal avec la commande java "NomDuFichier.jar".

Nous avons créé un petit menu qui permet de tester les différentes méthodes demandées. Il suffit de taper le numéro correspondant dans la console, puis de suivre les instructions demandées pour exécuter notre jeu et ses tests.

3) Vue globale de notre approche

Pour commencer, nous avons géré la lecture du fichier texte contenant les polygones du terrain et les parcours de jeu. Voici nos fonctions :

```
//Fonction qui permet de traiter les données récupérées dans le fichier pour créer les polygones indiqués.
        fonction lirePoly() : ArrayList
        entier nb = nombre de polygone dans le fichier;
        liste de Polygones polys = nouvelle liste de Polygones;
        liste de Point pt = nouvelle liste de Point;
        chaine de caractère couleur = " ";
        entier i = 2;
DEBUT
        Tant que (nb différent de 0) faire
                si (le string lu est différent de V OU de le string lu est différent de C le string lu est différent de B
OU le string lu est différent de S OU le string lu est différent de J) alors
                        ajouter un nouveau point (i, i+1) à pt;
                sinon
                        couleur = string lu;
                        créer un nouveau polygone(pt, couleur);
                        ajouter ce polygone à polys;
                        nb = nb - 1;
                        vider() pt;
                fin si
                i = i + 2;
        fin tant que
        retourner la liste de polygone;
FIN
```

```
//Fonction qui va lire les polygones des parcours donnés dans le fichier
methode lirePolyParcours(entier i)
        liste de Polygones li = nouvelle liste de Polygones;
        entier comptmp = indice de la fin de lecture des polynomes
        comptmp = indice de debut des li;
DEBUT
          pour i allant de 0 à 3 faire
                    ajouter le polynome correspondant
                    comptmp ++
          fin pour
          retourner li
FIN
```

```
//Fonction qui va lire le point de depart du parcours i
fonction lireDepartParcours(entier i) : Point
          Point depart = null;
          entier comptmp = indice de la fin de lecture des polynomes;
DEBUT
          comptmp = indice du x du pt de depart du parcours num i
          depart = nouveau Point(    entier lu à l'indice comptmp, entier lu à l'indice comptmp + 1)
          return depart;
FIN
```

```
//Fonction qui va lire le point de d'arrive du parcours i
fonction lireArriveParcours(entier i) : Point
          Point arrive = null;
          entier comptmp = indice de la fin de lecture des polynomes;
DEBUT
          comptmp = indice du x du pt d'arrive du parcours num i;
          arrive = nouveau Point(entier lu à l'indice comptmp, entier lu à l'indice comptmp + 1);
          return arrive;
FIN
```

```
//Fonction qui récupère le par du parcours i (commence a 1)
fonction lireParParcours(entier i) : entier
          Entier Par;
          entier comptmp = indice de la fin de lecture des polynomes;
DEBUT
          comptmp = indice du par du parcours num i;
          Par = entier lu à l'indice comptmp;
          return Par;
FIN
```

Ensuite nous avons écrit une approche global pour pouvoir exécuter une partie de jeu. Ainsi, nous avons pu bien identifier les fonctions que nous devions coder et ce qu'elles devaien précisément faire. Ainsi nous obtenons :

```
Ecrire("nombre de triangle max par region : ")
Lire(entier n);
//Création de deux liste pour stocker les triangles et les polygones
liste de triangle triangles = nouvelle liste de triangle
liste de Polygones nosPoly = nouvelle liste de Polygones
nosPoly = lirePoly(); //on récupère la liste de nos polygones
//On fait la triangularisation
Pour tout ( Polygone P dans nosPoly) faire
        P.triangulation_totale();
        Pour tout (Triangle T dans  la liste des triangles de P) faire
                Ajouter T à triangles;
        fin pour tout
fin pour tout
Region R = nouvelle Region (0.0 , 0.0 ,10.0 ,10.0);
Quadtree Q = nouveau Quadtree (R , n, triangles);
Q.Contruction();
//lire le numero du parcours
int np;
Si (nb_parcours > 1) alors
        Lire("Il y a " + nb_parcours + " parcours. Lequel voulez vous faire ?")
        lire(entier np)
Sinon
        np = 1
Fin si
//on lit les éléments pour creer la balle
Point P = lireDepartParcours(np);
Point trou = lireArriveParcours(np);
entier par = lireParParcours(np);
Balle B = nouvelle Balle(P, Q ,trou , par)
//on verifie si la balle est dans le trou
Tant que (non B.balle_dans_trou())
        Ecrire("Vous etes ici : ");
        Afficher le Point de B
        Ecrire("Point de votre cible, x puis y :");
        lire(réel x)
        lire(réel y)
        Point Cible = nouveau Point (x,y);
        Point atte = B.calcul_point_aterisage(Cible);
        B.calcul_point_depart(atte);
Fin tant que
//on affiche le score
Ecrire(B.calcul_score())
```

4) Méthodes demandées

a) ConstructionQt

```
procédure ConstructionQT()
        entier : n //nombre de region max par region
        double : Xmin, Ymin , Xmax, Ymax //definir la region
DEBUT
        Si ( region.nombre_de_triangles() > n) alors
```

```
                Xmin = region.Xmin
                Ymin = region.Ymin
                Xmax = region.Xmax
                Ymax = region.Ymax
                Q1 = arbre(Xmin, Ymax/2, Xmax/2, Ymax)
                Q2 = arbre(Xmax/2, Ymax/2, Xmax, Ymax)
                Q3 = arbre(Xmax/2, Ymin, Xmax, Ymax/2)
                Q4 = arbre(Xmin, Ymin, Xmax/2, Ymax/2)
        fin si
FIN
```

```
fonction Quadtree arbre(double Xmin, double Ymin, double Xmax, double Ymax)
        entier : n //nombre de region max par region
DEBUT
        Region R = nouvelle Region(Xmin, Ymin, Xmax, Ymax)
        Si ( region.nombre_de_triangles() <= n) alors
                Quadtree Q = nouveau Quadtree(R , n, triangles)
                renvoyer Q
        Sinon
                Quadtree Q = new Quadtree(R, n, triangles);
                Q.Q1 = arbre(Xmin, (Ymin +Ymax)/2, (Xmax + Xmin)/2, Ymax)
                Q.Q2 = arbre((Xmax + Xmin)/2, (Ymin +Ymax)/2, Xmax, Ymax)
                Q.Q3 = arbre((Xmax + Xmin)/2, Ymin, Xmax, (Ymin +Ymax)/2)
                Q.Q4 = arbre(Xmin, Ymin, (Xmax + Xmin)/2, (Ymin +Ymax)/2)
                renvoyer Q
        fin si
FIN
```

Explication du fonctionnement :

ConstructionQT va devoir se renseigner sur sa région principale si celle-ci contient plus de n triangles alors il va associer à chaque fils un autre quadtree en divisant la région en 4 sous régions. Par une fonction récursive “arbre” qui opère de la même manière mais en renvoyant le quadtree si la région contient moins de n triangles le quadtree va se construire récursivement

Pourquoi notre algorithme est-il correct ?

Notre algorithme est correct puisqu’il répond bien au besoin de construire le quadtree, avec un découpage des régions correct et associe bien à chaque région un nombre de triangle correct.

Complexité

O(n log (n)) avec n nombre de triangles du terrain et log en base 4

Explication : on découpe en 4 et pour chaque découpage on calcul le nombre de triangle de la région qui est une fonction avec une complexité de O(n)

Pourquoi cet algorithme est aussi efficace que possible ?

Pour chaque découpage il faut calculer le nombre de triangles de la région qui est au minimum en O(n) et un quadtree se découpe en 4 donc au minimum au fait log4(n) fois le calcul du nombre de triangles. C’est la complexité de notre algorithme il est donc aussi efficace que possible

b) TriangulationTerrain

```
procédure triangulation_totale()
DEBUT
        Si (nombre de points du polygone == 3) alors
                on forme un triangle avec ces 3 points

        Sinon si ( nombre de points du polygone == 2) alors
                rien car fin de la triangulation

        Sinon
```

```
                triangulation() //sur tous les points du polygone
                triangulation_totale //avec les points restants
        fin si
FIN
```

```
procédure triangulation()
        liste de point : polytmp
        liste de droite : D
        liste de droite : segments
        entier : k
        boolean : inter
DEBUT
        Pour ( i = 0 polytmp.taille() ) faire
                Si ( triangle_vide( polytmp[i] , polytmp[i+1] , polytmp[i+2]) ) alors
                        D.ajoute(Droite(polytmp[i],polytmp[i+2])
                fin si
        fin pour
        Pour (toutes droite dans D) faire
                Si (interPropre(droite)%2 == 1) alors
                        Si ( segment.estvide() ) alors
                                segment.ajoute(droite)
                                k = position du point A de droite dans polytmp
                                creation_triangle(k)
                        Sinon
                                Pour (toutes droite dans segment) faire
                                        Si (droite.intersegment(segment) == vrai) alors
                                                inter = true
                                        fin si
                                fin pour
                                Si (inter == false ) alors
                                        segment.ajoute(droite)
                                        k = position du point A de droite dans polytmp
                                        creation_triangle(k)
                                fin si
                        inter = false
                        fin si
                fin si
        fin pour
        Pour ( tous point B de chaque triangle construit )  faire
                polytmp.supprime(B)
        fin pour
FIN
```

Explication du fonctionnement

Triangulation_totale() est une méthode récursive sur le nombre de points du polygone, si celui ci est de 3 alors on construits un dernier triangle avec ses 3 points , si il est de 2 impossible de plus trianguler le polygone, et sinon on triangule le polygone

Pour trianguler le polygone on stock d'abord toutes les droites de points i et i+2 qui peuvent possiblement former un triangle. Si un autre point du polygone se trouve dans le triangle des points i i+1 i+2 on sait d'office qu'on ne le tracera pas

Ensuite pour chacune des droites on va tester son nombre d'intersection propre avec le polygone. Si celui ci est impair alors on regarde que la droite ne coupe aucun triangle déjà tracé . Si c'est le cas alors on trace le triangle sans oublier de supprimer le point central pour la suite de la triangulation.

Pour triangulariser tout le terrain il suffit de trianguliariser totalement chaque polygone.

Pourquoi notre algorithme est-il correct ?

Notre algorithme est correct puisque pour chaque polygone il va bien étudier tous les cas possible de construction de triangle en se basant sur un calcul d'intersection propre correct.

Complexité: O(n4)

Pourquoi cet algorithme est aussi efficace que possible ?

On est obligé de parcourir tous les polygones pour les trianguliariser totalement et donc possiblement faire n appel trianguler, et on doit tester toutes les droites du polygones dans triangulariser qui pour chaque droite doit tester son nombre d'intersection propre qui test les intersection prores avec tous les points du polygones. Donc au minimum dans le pire des cas la complexité de triangulariser terrain est de l'ordre O(n4)
Ce qui est en accord avec notre algorithme donc aussi efficace que possible

c) CalculCoefficientsDroite

```
procédure calculcoefdroite(Point P1, Point P2)
DEBUT
        b=P2.x-P1.x //droite sous la forme by = ax + c
        a=P2.y-P1.y
        c=b*P1.y-a*P1.x
FIN
```

Explication du fonctionnement: Calcul par rapport au vecteur des deux points
Pourquoi notre algorithme est-il correct ? Formules mathématiques
Complexité: O(1)
Pourquoi cet algorithme est aussi efficace que possible ? trivial

d) TestIntersectionDeuxSegments

```
fonction interSegment(Droite D)
DEBUT
        Si (intersection(D)) alors //intersection compare les vecteurs directeurs des droites
                Point P = point_inter(D) //point d'intersection des droites
                Si(A.x == B.x et D.A.x == D.B.x et P.y < max(A.y,B.y) et P.y > min(A.y,B.y) et ( P.y < max(D.A.y,D.B.y)
et P.y > min(D.A.y,D.B.y))) alors
                        renvoyer true
                Sinon si(A.x == B.x et P.y < max(A.y,B.y) et P.y > min(A.y,B.y) et (P.x < max(D.A.x,D.B.x) et P.x >
min(D.A.x,D.B.x))) alors
                        renvoyer true
                Sinon si(D.A.x == D.B.x et P.x < max(A.x,B.x) et P.x > min(A.x,B.x) et (P.y < max(D.A.y,D.B.y) et P.y >
min(D.A.y,D.B.y))) alors
                        renvoyer true
                Sinon si(P.x < max(A.x,B.x) et P.x > min(A.x,B.x) et (P.x < max(D.A.x,D.B.x) et P.x >
min(D.A.x,D.B.x))) alors
                        renvoyer true
                Sinon
                        renvoyer false;
                fin si
        Sinon
                renvoyer false;
        fin si
FIN
```

Explication du fonctionnement

Si on a une intersection entre les deux droites on test si le point d'intersection est compris entre tous les points

Pourquoi notre algorithme est-il correct ?

Notre algorithme renvoie bien si le point d'intersection est compris entres les points qui forment les segments

Complexité: O(1) tout est constant

Pourquoi cet algorithme est aussi efficace que possible ? On peut pas mieux faire qu'un temps constant

e) TestRegionIntersecteTriangle

```
procedure Region_intersect_triangle()
        liste de triangle : triangles, triangle_de_region
        boolean present
DEBUT
        Pour (tout triangle T dans triangles) faire
                present = false
                Pour (toutes droites D de région) faire
                        Pour (toutes droites D' de triangles) faire
                                Si ( D.intersegment(D') ) alors
                                        present = true
                                fin si
                        fin pour
                fin pour
                Si (!present) alors
                        present = triangle_present_entier //si triangle entier dans region
                fin si
                Si (present) alors
                        triangle_de_region.ajouter(T)
                fin si
        fin pour
FIN
```

Explication du fonctionnement

Pour tous les triangles possibles on va tester si un côté de la région s'intersecte avec un coté du triangle

Pourquoi notre algorithme est-il correct ?

Notre algorithme teste bien pour tous les triangles du terrain ceux qui intersecte la région.

Complexité

O(n) , n nombre de triangles du terrain, car on les testes tous et comme la région a 4 coté et chaque triangle 3 le reste est en O(1)

Pourquoi cet algorithme est aussi efficace que possible ?

On est obligé de tester tous les triangles donc O(n) est la complexité minimale possible.

f) recherchepointriangle

```
fonction Triangle triangle_du_point(Point P)
        liste de Triangle : triangles
        boolean trouve = false
        entier i
DEBUT
        i = 0
        tant que (trouve = false) faire
                T =  triangles[i]
                trouve = T.contient_point(P)
                i++
        fin tq
        renvoyer T
FIN
```

Explication du fonctionnement : Pour tous les triangles de la régions on teste celui qui contient le point

Pourquoi notre algorithme est-il correct ?

Notre algorithme est correct puis que un seul triangle contient le point donc le premier qui le contient sera renvoyé.

Complexité : O(n) car on test sur n triangles qui fait contient_triangle en temps constant

Pourquoi cet algorithme est aussi efficace que possible ?

Obligé de parcours tous les triangles donc O(n) minimum ce qui correspond à notre algorithme

g) RecherchePointQT

```
fonction region recherchePtQt (Point P)
        quadtree fils : Q1, Q2, Q3, Q4
        Region du quadtree : R
DEBUT
        Si ( Q1 == null et Q2 == null et Q3 == null et Q4 == null) alors
                renvoyer R
        Sinon
                Si ( P.x <= (R.Xmax + R.Xmin)/2 et y <= (R.Ymin + R.Ymax)/2) alors
                        renvoyer recherche_ds_arbre(P, Q4)
                Sinon si ( P.x <= (R.Xmax + R.Xmin)/2 et y > (R.Ymin + R.Ymax)/2) alors
                        renvoyer recherche_ds_arbre(P, Q1)
                Sinon si ( P.x > (R.Xmax + R.Xmin)/2 et y <= (R.Ymin + R.Ymax)/2) alors
                        renvoyer recherche_ds_arbre(P, Q3)
                Sinon
                        renvoyer recherche_ds_arbre(P, Q2)
                fin si
        fin si
FIN
```

```
fonction region recherche_dans_arbre (Point P, Quadtree Q)
DEBUT
        Si ( Q.Q1 == null et Q.Q2 == null et Q.Q3 == null et Q.Q4 == null) alors
                renvoyer Q.R
        Sinon
                Si ( P.x <= (Q.R.Xmax + Q.R.Xmin)/2 et y <= (Q.R.Ymin + Q.R.Ymax)/2) alors
                        renvoyer recherche_ds_arbre(P, Q.Q4)
                Sinon si ( P.x <= (Q.R.Xmax + Q.R.Xmin)/2 et y > (Q.R.Ymin + Q.R.Ymax)/2) alors
                        renvoyer recherche_ds_arbre(P, Q.Q1)
                Sinon si ( P.x > (Q.R.Xmax + Q.R.Xmin)/2 et y <= (Q.R.Ymin + Q.R.Ymax)/2) alors
                        renvoyer recherche_ds_arbre(P, Q.Q3)
                Sinon
                        renvoyer recherche_ds_arbre(P, Q.Q2)
                fin si
        fin si
FIN
```

Explication du fonctionnement

On recherche un point dans le quadtree , si tous les fils de la région sont à null ça veut dire qu'on ne peut descendre plus bas dans l'arbre et donc le point le point appartient à la région de ce quadtree.

Sinon en fonction des coordonnées du point on se dirige dans l'un des quadtree fils. Avec une fonction récursive recherche_ds_arbre on va pouvoir parcourir tout le quadtree principal.

Pourquoi notre algorithme est-il correct ?

Notre algorithme est correct puis qu'il se dirige dans les régions en fonction des coordonnées du point il renverra donc forcément la région qui contient le point. Il s'arrête aussi uniquement qd il est rendu à une feuille.

Complexité

O(log n) n nombre de triangles de terrain et log en base 4.

Pourquoi cet algorithme est aussi efficace que possible ?

Pour la recherche d'un élément impossible d'avoir une complexité en O(1) donc notre algorithme est aussi

efficace que possible.

h) triangle_contient_point

```
fonction boolean triangle_contient_point(Point P)
DEBUT
        double x0 = A.x
        double y0 = A.y
        double a = B.x - x0;
        double b = B.y - y0;
        double c = C.x - x0;
        double d = C.y - y0;
        double det = a*d - b*c;
        double xP = -1.0 , xp; // xM coordonnées de M dans repère (A,AB,AC), xm coordonnées de M dans repe
d'origine
        double yP = -1.0 , yp; // yM coordonnées de M dans repère (A,AB,AC), ym coordonnées de M dans repre
d'origine
        boolean contient = false; //triangle pas dans la région
        xp = P.x;
        yp = P.y;
        xP = (c*(y0 - yp) + d*(xp - x0))/det;
        yP = (a*(yp - y0) + b*(x0 - xp))/det;
        Si ((xP + yP) <= 1 et (xP + yP) >= 0 et xP <= 1 et xP >= 0 et yP <= 1 et yP >= 0) //le point est dans le triangle
                contient = true
        fin si
        return contient
FIN
```

Explication du fonctionnement

On fait un changement de repère en se basant sur le triangle et on calcul les coordonnées du point dans ce nouveau repère et si elles respectent les conditions alors le point se situe dans le triangle.

Pourquoi notre algorithme est-il correct ? Propriétés mathématiques donc algorithme correct
Complexité: O(1) que du calcul
Pourquoi cet algorithme est aussi efficace que possible ? Temps constant on ne peux pas mieux faire

i) CalculePointAtterrissageBalle

```
fonction Point calcul_point_atterissage(Point cible)
        Point  : trou, P //P point d'ou on tire
        boolean : sable, green
        double : alpha, d
        Quadtree : Q
        liste de double : a, b //liste des valeur changeant aléatoirement alpha et d
        entier i
DEBUT
        Point atte
        Region R = Q.recherchePtQt(cible)
        Triangle T = R.triangle_du_point(Cible)
        Si (T.couleur == "S") alors
                sable = true
        Sinon si (T.couleur == "C") alors
                green = true
        fin si
        d = sqrt((Cible.x – P.x)² + (Cible.y – P.y)²)
        alpha = arctan((Cible.y – P.y) / (Cible.x – P.x))
        alpha = 180 * alpa / pi
        Si ((Cible.x – P.x) < 0) alors
                alpha =+ 180
        fin si
```

```
        Si (green et d <= 0.1) alors
                Point atte = trou
        fin si
        i = aleatoire ( 0, 22)
        d = d * b[i]
        Si (sable) alors
                d = d/2
        fin si
        i = aleatoire ( 0, 22)
        alpha += a[i]
        alpha = (Math.PI * alpha) /180;
        Si (abs(a[i]) > 10) alors
                Ecrire ( " Vous avez beaucoup devie" )
        Sinon
                Ecrire ( " Vous avez peu devie" )
        Si (alpha >= 0 et alpha < pi/2) alors
                atte.x = cos(alpha) * d + P.x
                atte.y = sin(alpha) * d + P.y
        Sinon si (alpha >= pi/2 et alpha < pi) alors
                atte.x = -cos(alpha) * d + P.x
                atte.y = sin(alpha) * d + P.y
        Sinon si (alpha >= -pi et alpha < -pi/2) alors
                atte.x = -cos(alpha) * d + P.x
                atte.y = -sin(alpha) * d + P.y
        Sinon
                atte.x = cos(alpha) * d + P.x
                atte.y = -sin(alpha) * d + P.y
        fin si
        Si (atte.x > 10) alors //gestion des cas sorti des limite du terrain
                atte.x = 10
        Sinon si ( atte.x < 0) alors
                atte.x = 0
        fin si
        Si (atte.y > 10) alors
                atte.y = 10
        Sinon si ( atte.y < 0) alors
                atte.y = 0
        fin si
        renvoyer atte
FIN
```

Explication du fonctionnement

On commence d'abord par regarder de quel surface on pars pour savoir si on est dans du sable ou le green
Ensuite on calcul la distance d qui est égale à la norme du vecteur (P, Cible) et l'angle alpha avec la tangente.
Il faut ensuite remettre alpha en degré. Comme ça nous renvoie un alpha compris en -90 et 90 il faut adapter pour le mettre en 0 et 360
Si d est inférieur à 0,1 et qu'on est dans le green alors on dit que la balle arrive dans le trou et donc atte = trou.
A d et alpha on ajoute une valeur tiré aléatoirement.
Enfin si on pars du sable on divise d par 2.
Ensuite en fonction de alpha par des calculs trigo on trouve les coordonnées de atte, en faisant attention qu'il ne dépasse pas du terrain.

Pourquoi notre algorithme est-il correct ?

Notre algorithme est correct puisqu'il calcule bien le point d'atterrissage en fonction de la surface de départ et du point cible

Complexité

O(n) avec n nombre de triangle, tout est en temps constant sauf rechercheQt et recherche triangle

_du_point, la première est en O(log n) et la deuxième en O(n) donc O(n) l'emporte

Pourquoi cet algorithme est aussi efficace que possible ?

Pour tester à quel triangle de la région appartient le point on est obligé de tous les tester donc minimum O(n) donc notre algorithme est aussi efficace que possible

j) CalculePointDepartBalle

```
procédure calcul_point_depart (Point atte)
        Point : P //point on se trouve avant de taper la balle
        Quadtree : Q
        entier : score
DEBUT
        Region R = Q.recherche(atte)
        Triangle T = R.triangle_du_point(atte)
        Si (T.couleur == "S") alors
                Ecrire("Vous atterrissez dans la foret, coup non valide, +1 de pénalité")
                score += 2
        Sinon si (T.couleur == "B") alors
                Ecrire("Vous atterrissez dans l'eau, coup non valide, +1 de pénalité")
                score += 2
        Sinon si (T.couleur == "B") alors
                Ecrire("Vous atterrissez dans le sable, compliqué d'en sortir, distance réduite au prochain coup")
                score += 1
                P.x = atte.x
                P.y = atte.y
        Sinon
                Ecrire ("lancé correct !")
                score +=1
                P.x = atte.x
                P.y = atte.y
        fin si
FIN
```

Explication du fonctionnement

En fonction de où la balle atterrit on applique les conséquence du terrain. On utilise le quadtree et la région pour trouver la surface. On augmente aussi le score en fonction du lancé.

Pourquoi notre algorithme est-il correct ?

Il applique bien les règles définies dans le sujet

Complexité

O(n) avec n nombre de triangle, tout est en temps constant sauf recherchePtQt et recherche triangle_du_point, la première est en O(log n) et la deuxième en O(n) donc O(n) l'emporte

Pourquoi cet algorithme est aussi efficace que possible ?

Pour tester à quel triangle de la région appartient le point on est obligé de tous les tester donc minimum O(n) donc notre algorithme est aussi efficace que possible

k) CalculScore

```
procédure entier calculscore()
        entier : score, par
DÉBUT
        renvoyer score - par
FIN
```

Explication du fonctionnement : regle du golf

Pourquoi notre algorithme est-il correct ? Car renvoie bien le score

Complexité : O(1)

Pourquoi cet algorithme est aussi efficace que possible ? Trivial

5) Conclusion

Tous nos jeux de test des fonctions demandées se trouvent dans le document pdf “JeuxTest_Petit_Pelloquin.pdf”

Ce projet nous a permis d’utiliser la structure de quadtree et de nous rendre compte de son efficacité sur un projet concret. Nous avons aussi testé l’affichage graphique mais nous avons manqué de temps pour pouvoir tout réaliser dans l’interface graphique. Ainsi, nous affichons seulement le terrain de golf, les points de départ et d’arrivées et la triangularisation des polygones du terrain. Pour le reste, nous utilisons la console en affichant notre menu, nos tests de méthode et nos résultats de jeu.
Ce projet nous a demandé beaucoup de temps et d’investissement pour pouvoir aboutir à la fin de notre jeu. Celui-ci fonctionne néanmoins correctement et nous pouvons lancer une partie. Malgré de nombreuses heures passées à coder, nous n’avons pas eu le temps de tout finir, en effet, il nous manque la gestion du cas où la balle tombe dans l’eau.