```
(*
```

PROJET MLSF - ENSIIE 2015

Thomas Sarboni

1. Echauffement sur les rectangles

Déclaration des types

```
*)
type coord = float * float ;;
type rect = R of coord * coord ;;

let coord0 : coord = ( 0., 0. ) ;;
let coord1 : coord = ( 1., 1. ) ;;
let coord2 : coord = ( 0.5, 0.5 ) ;;
let coord3 : coord = ( 1.5, 1.5 ) ;;
(*
```

1.1 Fonction make_rect

```
*)
(* Crée un rectangle à partir de coordonnées *)
let make_rect = fun ( x, y ) -> fun ( x', y' ) ->
   if x <= x' then
      if y >= y' then
         R ( ( x, y ), ( x', y' ) )
      else
         R ( ( x, y' ), ( x', y ) )
   else
      if y >= y' then
         R ( ( x', y ), ( x, y' ) )
      else
         R ( ( x', y' ), ( x, y ) )
;;

(* Tests *)
let rect0 = make_rect coord0 coord1 ;;
let rect1 = make_rect coord2 coord3 ;;
assert ( not ( rect0 = rect1 ) ) ;;
let coord00 : coord = ( 0., 1. ) ;;
let coord01 : coord = ( 1., 0. ) ;;
let rect00 : rect = R ( coord00, coord01 ) ;;
assert ( rect0 = rect00 ) ;;
(*
```

1.2 Fonctions de projection

```
*)
(* Retourne l'abcisse du côté gauche d'un rectangle *)
let rect_left = fun r ->
   let R ( ( x, y ), ( x', y' ) ) = r in x
;;

(* Retourne l'abcisse du côté droit d'un rectangle *)
let rect_right = fun r ->
   let R ( ( x, y ), ( x', y' ) ) = r in x'
;;

(* Retourne l'ordonnée du côté inférieur d'un rectangle *)
let rect_bottom = fun r ->
   let R ( ( x, y ), ( x', y' ) ) = r in y'
;;

(* Retourne l'ordonnée du côté supérieur d'un rectangle *)
let rect_top = fun r ->
   let R ( ( x, y ), ( x', y' ) ) = r in y
;;

(* Tests *)
assert ( ( rect_left rect0 ) = 0. ) ;;
assert ( ( rect_right rect0 ) = 1. ) ;;
assert ( ( rect_bottom rect0 ) = 0. ) ;;
assert ( ( rect_top rect0 ) = 1. ) ;;
(*
```

1.3 Fonctions de mesure

```
*)
(* Retourne la longueur d'un rectangle *)
let rect_length = fun r ->
   let R ( ( x, y ), ( x', y' ) ) = r in abs_float ( x' -. x )
;;

(* Retoure la hauteur d'un rectangle *)
let rect_height = fun r ->
   let R ( ( x, y ), ( x', y' ) )  = r in abs_float ( y -. y' )
;;

(* Tests *)
assert ( ( rect_length rect0 ) = 1. ) ;;
assert ( ( rect_height rect0 ) = 1. ) ;;
(*
```

1.4 Fonction rect_mem

```
*)
(* Indique si un point est dans un rectangle *)
let rect_mem = fun r -> fun ( xc, yc ) ->
    let R ( ( x, y ), ( x', y' ) ) = r in
    if x <= xc && xc <= x' then
        if y >= yc && yc >= y' then
            true
        else
            false
    else
        false
;;

(* Tests *)
assert ( rect_mem rect0 coord2 ) ;;
assert ( not ( rect_mem rect0 coord3 ) ) ;;
assert ( rect_mem ( R ( ( 0., 4. ), ( 6., 0. ) ) ) ( 0.8, 1.4 ) ) ;;
(*
```

1.5 Fonction rect_intersect

```
*)
(* Indique si l'intersection entre deux rectangles est non nulle *)
let rect_intersect = fun r -> fun r' ->
    let R ( ( a, b ), ( a', b' ) ) = r in
    let R ( ( x, y ), ( x', y' ) ) = r' in
    ( b' >= y' && b' <= y )
    || ( b >= y' && b <= y )
    || ( a' >= x' && a' <= x )
    || ( a >= x' && a <= x )
;;

(* Tests *)
assert ( rect_intersect rect0 rect1 ) ;;
(*
```

1.6 Fonction rect_split

```
*)
(* Indique le centre d'un rectangle (BONUS) *)
let rect_center = fun r ->
    let R ( ( x, y ), ( x', y' ) ) = r in
    let a = x +. ( rect_length r /. 2. ) in
    let b = y' +. ( rect_height r /. 2. ) in
    let g : coord = ( a, b ) in g
;;

(* Decoupe un rectangle en quatre à partir du centre *)
let rect_split = fun r ->
    let R ( ( x, y ), ( x', y' ) ) = r in
    let ( a, b ) = rect_center r in
    let r0 = make_rect ( x, y ) ( a, b ) in
    let r1 = make_rect ( a, y ) ( x', b ) in
    let r2 = make_rect ( a, b ) ( x', y' ) in
    let r3 = make_rect ( x, b ) ( a, y' ) in
    r0, r1, r2, r3
;;

(* Tests *)
let r0 = make_rect ( 0., 1. ) ( 0.5, 0.5 ) ;;
let r1 = make_rect ( 0.5, 1. ) ( 1., 0.5 ) ;;
let r2 = make_rect ( 0.5, 0.5 ) ( 1., 0. ) ;;
let r3 = make_rect ( 0., 0.5 ) ( 0.5, 0. ) ;;
let s0, s1, s2, s3 = rect_split rect0 ;;
assert ( r0 = s0 ) ;;
assert ( r1 = s1 ) ;;
assert ( r2 = s2 ) ;;
assert ( r3 = s3 ) ;;

(* Donne pour un rectangle et une coordonnée le rectangle 4x plus grand
 * ainsi que la direction vers laquelle on a étendu le rectangle (BONUS)
 * 0 = NordOuest ; 1 = NordEst ; 2 = SudEst ; 3 = SudOuest *)
let rect_duplicate = fun r -> fun ( a, b ) ->
    let R ( ( x, y ), ( x', y' ) ) = r in
    let h = rect_height r in
    let l = rect_length r in
    if ( b >= y' ) then
        if ( a <= x' ) then
            make_rect ( ( x -. l ), ( y +. h ) ) ( x', y' ), 0
        else
            make_rect ( x, ( y +. h ) ) ( ( x' +. l ), y' ), 1
    else
        if ( a >= x ) then
            make_rect ( x, y ) ( ( x' +. l ), ( y' -. h ) ), 2
        else
            make_rect ( ( x -. l ), y ) ( x', ( y' -. h ) ), 3
;;

(* Tests *)
let new_coord : coord = ( 0.75, 0.25 ) ;;
assert ( ( rect_duplicate s0 new_coord ) = ( rect0, 2 ) ) ;;
(*
```

2. La structure de données QuadTree

Déclaration des types

```
*)
type 'a quadtree =
    | Q of rect * 'a cell
and 'a cell =
    | Empty
    | Leaf of coord * 'a
    | Node of 'a quadtree * 'a quadtree * 'a quadtree * 'a quadtree
;;

let cell0 : 'a cell = Empty ;;
let quad0 : 'a quadtree = Q ( rect0, cell0 ) ;;

(* Creation d'un QuadTree correspondant à celui du poly *)

let c_empty : 'a cell = Empty ;;
let cA : coord = ( 2.1, 0.8 ) ;;
let cB : coord = ( 5.3, 3.2 ) ;;
let cC : coord = ( 0.4, 2.5 ) ;;
let cD : coord = ( 4.6, 2.2 ) ;;
let c_A : 'a cell = Leaf ( cA, 'A' ) ;;
let c_B : 'a cell = Leaf ( cB, 'B' ) ;;
let c_C : 'a cell = Leaf ( cC, 'C' ) ;;
let c_D : 'a cell = Leaf ( cD, 'D' ) ;;
let rect10 : rect = make_rect ( 0., 4. ) ( 6., 0. ) ;;
let rect11 : rect = make_rect ( 0., 4. ) ( 3., 2. ) ;;
let rect12 : rect = make_rect ( 3., 4. ) ( 6., 2. ) ;;
let rect13 : rect = make_rect ( 3., 2. ) ( 6., 0. ) ;;
let rect14 : rect = make_rect ( 0., 2. ) ( 3., 0. ) ;;
let rect15 : rect = make_rect ( 3., 4. ) ( 4.5, 3.) ;;
let rect16 : rect = make_rect ( 4.5, 4. ) ( 6., 3. ) ;;
let rect17 : rect = make_rect ( 4.5, 3. ) ( 6., 2. ) ;;
let rect18 : rect = make_rect ( 3., 3. ) ( 4.5, 2. ) ;;

let quad1 : 'a quadtree =
   Q ( rect10, Node(
      Q ( rect11, c_C ),
      Q ( rect12, Node(
         Q ( rect15, c_empty ),
         Q ( rect16, c_B ),
         Q ( rect17, c_D ),
         Q ( rect18, c_empty )
      ) ),
      Q ( rect13, c_empty ),
      Q ( rect14, c_A )
   ) )
;;
(*
```

2.7 Avantages de cette structure de données

Cette structure de données, comparée à un ensemble, présente l'avantage principal d'être dynamique.

En effet, l'espace occupé en mémoire peut évoluer au fur et à mesure de l'augmentation du niveau de détail.

Il n'est pas nécessaire de stocker l'ensemble du maillage.

L'économie sera d'autant plus importante si le plan contient des zones vides.

En contrepartie, l'accès à une case du plan nécessite le parcours de l'arbre alors que dans le cas d'un ensemble, on aurait pu directement accéder à la bonne case.

Enfin, si les points sont regroupés dans un espace réduit du plan on va aboutir à une situation ou l'arbre sera particulièrement déséquilibré et/ou son parcours va s'avérer coûteux.

2.8 Fonction boundary

```
*)
(* Donne pour un QuadTree le rectangle dans lequel il s'inscrit *)
let boundary = fun q ->
    match q with
    | Q ( r, _ ) -> r
;;

(* Tests *)
let rect2 = boundary quad0 ;;
assert ( rect2 = rect0 ) ;;
(*
```

2.9 Fonction cardinal

```
*)
(* Donne pour un QuadTree le nombre d'objets présents *)
let rec cardinal = fun q ->
    match q with
    | Q ( _, Empty ) -> 0
    | Q ( _, Leaf ( _, _ ) ) -> 1
    | Q ( _, Node ( f0, f1, f2, f3 ) ) ->
        cardinal f0 + cardinal f1 + cardinal f2 + cardinal f3
;;

(* Tests *)
assert ( cardinal quad0 = 0 ) ;;
assert ( cardinal quad1 = 4 ) ;;
(*
```

2.10 Fonction list_of_quadtree

```
*)
(* Donne pour un QuadTree la liste des objets présents *)
let rec list_of_quadtree = fun q ->
    match q with
    | Q ( _, Empty ) -> []
    | Q ( _, Leaf ( c, a ) ) -> [ ( c, a ) ]
    | Q ( _, Node ( f0, f1, f2, f3 ) ) ->
        let lf0 = list_of_quadtree f0 in
        let lf1 = List.append lf0 ( list_of_quadtree f1 ) in
        let lf2 = List.append lf1 ( list_of_quadtree f2 ) in
        List.append lf2 ( list_of_quadtree f3 )
;;

(* Tests *)
let list0 = [] ;;
let list1 = [ ( cC, 'C' ); ( cB, 'B' ); ( cD, 'D' ); ( cA, 'A' ) ] ;;
assert ( list0 = ( list_of_quadtree quad0 ) ) ;;
assert ( list1 = ( list_of_quadtree quad1 ) ) ;;
(*
```

2.11 Fonction insert

Fonctions préliminaires

```
*)
(* Donne pour un QuadTree son premier fils (BONUS) *)
let quad_get_first_child = fun q ->
   match q with
   | Q ( _, Empty ) | Q ( _, Leaf ( _, _ ) ) -> q
   | Q ( _, Node ( q0, _, _, _ ) ) -> q0
;;

(* Donne pour un QuadTree et une coordonnée le fils correspondant (BONUS) *)
let quad_get_child = fun q -> fun c ->
   match q with
   | Q ( _, Empty ) | Q ( _, Leaf ( _, _ ) ) -> q
   | Q ( _, Node ( q0, q1, q2, q3 ) ) ->
      if ( rect_mem ( boundary q0 ) c ) then q0
      else if ( rect_mem ( boundary q1 ) c ) then q1
      else if ( rect_mem ( boundary q2 ) c ) then q2
      else if ( rect_mem ( boundary q3 ) c ) then q3
      else q
;;

(* Donne pour un QuadTree et une coordonnée le QuaTree 4x plus grand (BONUS) *)
let quad_quad = fun q -> fun c ->
   let Q ( r, cell ) = q in
   let r', d = rect_duplicate r c in
   let r0, r1, r2, r3 = rect_split r' in
   let q0 = Q ( r0, Empty ) in
   let q1 = Q ( r1, Empty ) in
   let q2 = Q ( r2, Empty ) in
   let q3 = Q ( r3, Empty ) in
   if ( d = 0 ) then
      Q ( r', ( Node ( q0, q1, q, q3 ) ) )
   else if ( d = 1 ) then
      Q ( r', ( Node ( q0, q1, q2, q ) ) )
   else if ( d = 2 ) then
      Q ( r', ( Node ( q, q1, q2, q3 ) ) )
   else
      Q ( r', ( Node ( q0, q, q2, q3 ) ) )
;;

(* Tests *)
let rect04 = make_rect ( 0., 1. ) ( 2., -1. ) ;;
let quad04  = quad_quad quad0 ( 1.5, -0.5 ) ;;
assert ( ( boundary quad04 ) = rect04 ) ;;
assert ( ( quad_get_child quad04 ( 0.5, 0.5 ) ) = quad0 ) ;;

(* Donne pour un QuadTree le QuaTree séparé en quatre (BONUS) *)
let quad_split = fun q ->
   let Q ( r, _ ) = q in
   let r0, r1, r2, r3 = rect_split r in
   let q0 = Q ( r0, Empty ) in
   let q1 = Q ( r1, Empty ) in
   let q2 = Q ( r2, Empty ) in
   let q3 = Q ( r3, Empty ) in
   Q ( r, ( Node ( q0, q1, q2, q3 ) ) )
;;

(* Tests *)
let quad04' = Q ( rect04, Empty ) ;;
let quad04'' = quad_split quad04' ;;
assert ( ( quad_get_first_child quad04'' ) = quad0 ) ;;

(*
```

Fonction insert

```
*)

(* Insert un objet dans un QuadTree à partir d'un nom et de cordonnées *)
(* Il n'a pas été précisé dans le sujet si il fallait accepter des coordonnées
 * situées hors du rectangle initial et donc étendre le QuadTree. Je vais
 * partir du principe que cette opération est possible pour commencer. *)
let rec insert = fun q -> fun c -> fun n ->
    if not ( rect_mem ( boundary q ) c ) then
        (* ici on aurait pu retourner directement q pour ignorer les
         * objets ne faisant pas partie du rectangle initial *)
        let q' = quad_quad q c in insert q' c n
    else
        match q with
        | Q ( r, Empty ) ->
            Q ( r, Leaf ( c, n ) )
        | Q ( r, Leaf ( cl, nl ) ) ->
            let q' = insert ( quad_split q ) cl nl in
            insert q' c n
        | Q ( r, Node ( q0, q1, q2, q3 ) ) ->
            if ( rect_mem ( boundary q0 ) c ) then
                Q ( r, Node ( ( insert q0 c n ), q1, q2, q3 ) )
            else if ( rect_mem ( boundary q1 ) c ) then
                Q ( r, Node ( q0, ( insert q1 c n ), q2, q3 ) )
            else if ( rect_mem ( boundary q2 ) c ) then
                Q ( r, Node ( q0, q1, ( insert q2 c n ), q3 ) )
            else if ( rect_mem ( boundary q3 ) c ) then
                Q ( r, Node ( q0, q1, q2, ( insert q3 c n ) ) )
            else
                assert false
;;
(*
```

Quelque tests

```
*)

(* Tests *)
let cE = ( 4.5, 1. ) ;;
let cF = ( 0.8, 1.4 ) ;;
let c_E : 'a cell = Leaf ( cE, 'E' ) ;;
let c_F : 'a cell = Leaf ( cF, 'F' ) ;;
let rect19, rect20, rect21, rect22 = rect_split rect14 ;;

let quad2 : 'a quadtree =
   Q ( rect10, Node(
      Q ( rect11, c_C ),
      Q ( rect12, Node (
         Q ( rect15, c_empty ),
         Q ( rect16, c_B ),
         Q ( rect17, c_D ),
         Q ( rect18, c_empty )
      ) ),
      Q ( rect13, c_E ),
      Q ( rect14, Node (
         Q ( rect19, c_F ),
         Q ( rect20, c_empty ),
         Q ( rect21, c_A ),
         Q ( rect22, c_empty )
      ) )
   ) )
;;

let cG : coord = ( -1., 2.5 ) ;;
let c_G : 'a cell = Leaf ( cG, 'G' ) ;;
let rect10', _ = rect_duplicate rect10 cG
let rect23, rect24, rect25, rect26 = rect_split rect10' ;;

let quad3 : 'a quadtree =
   Q ( rect10', Node (
      Q ( rect23, Empty ),
      Q ( rect24, Empty ),
      Q ( rect25, Node (
         Q ( rect11, c_C ),
         Q ( rect12, Node (
            Q ( rect15, c_empty ),
            Q ( rect16, c_B ),
            Q ( rect17, c_D ),
            Q ( rect18, c_empty )
         ) ),
         Q ( rect13, c_E ),
         Q ( rect14, Node (
            Q ( rect19, c_F ),
            Q ( rect20, c_empty ),
            Q ( rect21, c_A ),
            Q ( rect22, c_empty )
         ) )
      ) ),
      Q ( rect26, c_G )
   ) )
;;

let quad1' = insert quad1 ( 4.5, 1. ) 'E' ;;
let quad1'' = insert quad1' ( 0.8, 1.4 ) 'F' ;;
assert ( quad1'' = quad2 ) ;;
let quad10 = insert quad1'' cG 'G' ;;
assert ( quad10 = quad3 ) ;;
(*
```

2.12 Fonction quadtree_of_list

```
*)
(* Donne pour une liste le rectangle qui la contient (BONUS) *)
let list_boundary = fun l ->
    let r = R ( ( 0., 0. ), ( 0., 0. ) ) in
    let min_max_list = fun rect -> fun ( ( a, b ), _ ) ->
        let R ( ( x, y ), ( x', y' ) ) = rect in
        let a' = ceil a in
        let b' = ceil b in
        R ( ( min x a', max y b' ), ( max x' a', min y' b' ) )
    in
    List.fold_left min_max_list r l
;;

(* Donne pour une liste de coordonnées le QuadTree correspondant *)
let quadtree_of_list = fun l ->
    let q = Q ( ( list_boundary l ), Empty ) in
    List.fold_left ( fun acc e -> let ( c, n ) = e in insert acc c n ) q l
;;

(* Tests *)
let quad11 = quadtree_of_list list1 ;;
assert ( ( quadtree_of_list list1 ) = quad1 ) ;;
(*
```

2.13 Fonction remove

Une première version utilisant une liste

```
*)
(* Donne pour un QuadTree et un élément un QuadTree privé de cet élément *)
let remove = fun q -> fun o ->
    let list_remove = fun acc -> fun e ->
        if ( e = o ) then
            acc
        else
            e::acc
    in
    let l = list_of_quadtree q in
    let l' = List.fold_left list_remove [] l in
    quadtree_of_list l'
;;

(* Tests *)
let list2 = [ ( cC, 'C' ); ( cB, 'B' ); ( cD, 'D' ) ] ;;
assert ( ( list_of_quadtree ( remove quad1 ( cA, 'A' ) ) ) = list2 ) ;;

(*
```

Une seconde version plus performante

```
*)

(* Supprime la subdivision d'un quadtree si celle-ci n'est pas utile (BONUS) *)
let clean_quad = fun quad ->
    match quad with
    | Q ( _, ( Empty | Leaf ( _, _ ) ) ) -> assert false
    | Q ( rect, Node ( q0, q1, q2, q3 ) ) ->
        match q0, q1, q2, q3 with
        | Q ( _, Empty ), Q ( _, Empty ), Q ( _, Empty ), Q ( _, Empty ) ->
            Q ( rect, Empty )
        | Q ( _, content ), Q ( _, Empty ), Q ( _, Empty ), Q ( _, Empty ) ->
            Q ( rect, content )
        | Q ( _, Empty ), Q ( _, content ), Q ( _, Empty ), Q ( _, Empty ) ->
            Q ( rect, content )
        | Q ( _, Empty ), Q ( _, Empty ), Q ( _, content ), Q ( _, Empty ) ->
            Q ( rect, content )
        | Q ( _, Empty ), Q ( _, Empty ), Q ( _, Empty ), Q ( _, content ) ->
            Q ( rect, content )
        | _ -> quad
;;

(* Donne pour un QuadTree et un élément un QuadTree privé de cet élément *)
(* Ne repasse pas par une liste (BONUS) *)
let rec remove' = fun q -> fun o ->
    let ( c, _ ) = o in
    match q with
    | Q ( r, Empty ) -> assert false
    | Q ( r, Leaf ( _, _) ) -> Q ( r, Empty )
    | Q ( r, Node ( q0, q1, q2, q3 ) ) ->
        if ( rect_mem ( boundary q0 ) c ) then
            let q' = Q ( r, Node ( ( remove' q0 o ), q1, q2, q3 ) ) in
            clean_quad q'
        else if ( rect_mem ( boundary q1 ) c ) then
            let q' = Q ( r, Node ( q0, ( remove' q1 o ), q2, q3 ) ) in
            clean_quad q'
        else if ( rect_mem ( boundary q2 ) c ) then
            let q' = Q ( r, Node ( q0, q1, ( remove' q2 o ), q3 ) ) in
            clean_quad q'
        else if ( rect_mem ( boundary q3 ) c ) then
            let q' = Q ( r, Node ( q0, q1, q2, ( remove' q3 o ) ) ) in
            clean_quad q'
        else
            assert false
;;

(* Tests *)
assert ( ( list_of_quadtree ( remove' quad1 ( cA, 'A' ) ) ) = list2 ) ;;
assert ( ( remove' quad3 ( cG, 'G' ) ) = quad2 ) ;;
(*
```

3. Représentation graphique d'un QuadTree et tests

Inclusion du fichier display.ml

```
*)
(* #use "display.ml" ;; *)

(* draw_data: (float * float * float) -> ('a -> string) -> coord * 'a
 *          -> Graphics.ceolor -> unit
 *)
let draw_data = fun (sx,sy,z) data_to_string data col ->
    let ((cx,cy), label) = data in
    let x = int_of_float (sx +. z *. cx) in
    let y = int_of_float (sy +. z *. cy) in
    let _ = Graphics.set_color col in
    let _ = Graphics.draw_circle x y 1 in
    let _ = Graphics.moveto (x+3) (y-2) in
    let _ = Graphics.draw_string (data_to_string label) in
    let _ = Graphics.set_color Graphics.black in
    ()
;;

(* draw_quadtree: float * float * float -> ('a -> string) -> 'a quadtree
 *              -> unit
 *)
let rec draw_quadtree = fun dparams data_to_string qt ->
    let sx,sy,z = dparams in
    let Q (r, qc) = qt in
    let x1 = int_of_float (sx +. z *. rect_left r) in
    let y1 = int_of_float (sy +. z *. rect_bottom r) in
    let x2 = int_of_float (sx +. z *. rect_right r) in
    let y2 = int_of_float (sy +. z *. rect_top r) in
    let _ = Graphics.set_color Graphics.blue in
    let _ = Graphics.draw_rect x1 y1 (x2-x1) (y2-y1) in
    let _ = Graphics.set_color Graphics.black in
    match qc with
    | Empty -> ()
    | Leaf (c,d) -> draw_data dparams data_to_string (c,d) Graphics.black
    | Node (nw,ne,se,sw) ->
            let _ = draw_quadtree dparams data_to_string nw in
            let _ = draw_quadtree dparams data_to_string ne in
            let _ = draw_quadtree dparams data_to_string se in
            let _ = draw_quadtree dparams data_to_string sw in
            ()
;;
(*
```

```
*)
(* wait_and_quit: unit -> unit
 *)
let wait_and_quit = fun () ->
    let _ = Graphics.wait_next_event [ Graphics.Key_pressed ] in
    Graphics.close_graph ()
;;

(* init: rect -> float * float * float
 *)
let init = fun r ->
    let _ = Graphics.open_graph "" in
    let _ = Graphics.set_color Graphics.black in
    let m = 10 in
    let w = Graphics.size_x () - 2 * m in
    let h = Graphics.size_y () - 2 * m in
    let zx = (float_of_int w) /. (rect_right r -. rect_left r) in
    let zy = (float_of_int h) /. (rect_top r -. rect_bottom r) in
    let z = min zx zy in
    let sx = (float_of_int m) -. z *. rect_left r in
    let sy = (float_of_int m) -. z *. rect_bottom r in
    (sx, sy, z)
;;

(* simple_test: 'a quadtree -> ('a -> string) -> unit
 *)
let simple_test = fun qt f ->
    let r = boundary qt in
    let dparams = init r in
    let _ = draw_quadtree dparams f qt in
    wait_and_quit ()
;;
(*
```

3.14 Commentaire du fichier display.ml

La fonction simple_test prend un quadtree en paramètre ainsi qu'une fonction de conversion en chaine des données de ce quadtree et effectue les actions suivantes :

- Récuération du rectangle correspondant au QuadTree
- Utilisation de la fonction init pour générer le cadre du graphe
- Dessin récursif du QuadTree grace à la fonction draw_quadtree
- Attente de la saisie d'un touche grace à la fonction wait_and_quit

La fonction init permet d'initialiser la fenêtre d'affichage en effectuant les actions suivantes :

- Ouverture de la fenêtre d'affichage
- Définition de la couleur d'écriture à noir
- Définition d'une marge de 10px
- Calcul de la taille de la fenêtre
- Déduction de l'échelle d'affichage en pixels
- Et des coordonnées de l'origine du graphe

La fonction draw_quadtree permet de dessiner un rectangle en effectuant les actions suivantes : *Récupération des coordonnées des extrémités du rectangle* Définition de la couleur d'écriture en bleu *Dessin du rectangle (args : coordonnées à l'origine et longueur du côté)* Définition de la couleur d'écriture en noir *Si on est une feuille : appel de la fonction draw_data* Si on est un noeud : appel de draw_quadtree sur les sous-arbres * Si on est vide, on renvoie ()

La fonction draw_data permet de dessiner les données contenues dans une feuille.

Elle prend en paramètre la fonction permettant de convertir les données en chaine de caractères.

Par exemple, pour un quadtree de char, on peut passer Char.escaped, pour un entier, string_of_int.

Elle prend également en paramètre la couleur à utiliser, noir dans ce cas.

Elle effectue les actions suivantes :

- Récupération des coordonnées du point
- Déduction des coordonnées d'affichage
- Affichage d'un cercle de rayon 1px
- Déplacement à côté du cercle
- Ecriture des données correspondant au point

3.15 Adaptation de la fonction simple_test pour tester le code du projet

Fonction simple_test' : stratégie de test

La fonction simple_test' effectue les tâches suivantes :

- Creation du quadtree d'exemple à l'aide de la fonction quadtree_of_list
- Insertion d'un objet dans une case vide
- Insertion d'un objet dans une case occupée
- Insertion d'un objet hors des limites du quadtree
- Suppression des trois objets créés précédemment
- Le quad tree final doit être le même que le premier
- Le quadtree doit être optimal à chaque étape

```
*)
let wait_and_quit' = fun () ->
    Graphics.wait_next_event [ Graphics.Key_pressed ]
;;

(* Fonction simple_test' : test des fonctions avec l'arbre du poly *)
let simple_test' = fun l -> fun f ->
    let q1 = quadtree_of_list l in
    let r = boundary q1 in
    let dparams = init r in
    let _ = draw_quadtree dparams f q1 in
    let _ = wait_and_quit' () in
    let q2 = insert q1 cE 'E' in
    let _ = draw_quadtree dparams f q2 in
    let _ = wait_and_quit' () in
    let q3 = insert q2 cF 'F' in
    let _ = draw_quadtree dparams f q3 in
    let _ = wait_and_quit' () in
    let q4 = insert q3 cG 'G' in
    let r' = boundary q4 in
    let dparams = init r' in
    let _ = Graphics.clear_graph () in
    let _ = draw_quadtree dparams f q4 in
    let _ = wait_and_quit' () in
    let q5 = remove' q4 ( cG, 'G' ) in
    let dparams = init r in
    let _ = Graphics.clear_graph () in
    let _ = draw_quadtree dparams f q5 in
    let _ = wait_and_quit' () in
    let q6 = remove' q5 ( cF, 'F' ) in
    let _ = Graphics.clear_graph () in
    let _ = draw_quadtree dparams f q6 in
    let _ = wait_and_quit' () in
    let q7 = remove' q6 ( cE, 'E' ) in
    let _ = Graphics.clear_graph () in
    let _ = draw_quadtree dparams f q7 in
    let _ = wait_and_quit' () in
    Graphics.close_graph ()
;;

(* Tests *)
(* simple_test' list1 Char.escaped ;; *)

(* random_test : Donne un Int QuadTree composé de 100 éléments aléatoires *)
let random_test = fun () ->
    let _ = Random.self_init in
    let q = ref ( Q ( R ( ( 0., 1. ), ( 1., 0. ) ), Empty ) ) in
    for i = 0 to 99 do
        let a = Random.float 99. in
        let b = Random.float 99. in
        q := insert !q ( a, b ) i
    done ;
    !q
;;

(* Tests *)
let quad_r = random_test () ;;
assert ( ( cardinal quad_r ) = 100 ) ;;
(* simple_test quad_r string_of_int ;; *)
(*
```

3.16 Modification de la

fonction draw_quadtree pour utiliser rect_length

```
*)
(* draw_quadtree' : Affichage d'un QuadTree en utilisant rect_length *)
let rec draw_quadtree' = fun dparams data_to_string qt ->
    let sx,sy,z = dparams in
    let Q (r, qc) = qt in
    let x1 = int_of_float (sx +. z *. rect_left r) in
    let y1 = int_of_float (sy +. z *. rect_bottom r) in
    let length = int_of_float ( z *. ( rect_length r ) ) in
    let height = int_of_float ( z *. ( rect_height r ) ) in
    let _ = Graphics.set_color Graphics.blue in
    let _ = Graphics.draw_rect x1 y1 length height in
    let _ = Graphics.set_color Graphics.black in
    match qc with
    | Empty -> ()
    | Leaf (c,d) -> draw_data dparams data_to_string (c,d) Graphics.black
    | Node (nw,ne,se,sw) ->
            let _ = draw_quadtree' dparams data_to_string nw in
            let _ = draw_quadtree' dparams data_to_string ne in
            let _ = draw_quadtree' dparams data_to_string se in
            let _ = draw_quadtree' dparams data_to_string sw in
            ()
;;

(* simple_test'' : Affichage d'un QuadTree utilisant draw_quadtree' (BONUS) *)
let simple_test'' = fun qt f ->
    let r = boundary qt in
    let dparams = init r in
    let _ = draw_quadtree' dparams f qt in
    wait_and_quit ()
;;

(* Tests *)
(* simple_test'' quad_r string_of_int ;; *)
(*
```

Le problème est le suivant :

Utilser rect_length provoque un arrondi lors du calcul de la longueur.

Celà provoque une légère incohérence entre les rectangle contenants et les rectangles contenus, ce qui se traduit à l'affichage des varaiations de taille de marge.

4. Placement du disque

4.17 Fonction collision_disk_point

```
*)
(* collision_disk_point : Indique pour un cercle et point si le point fait
 * partie du cercle *)
let collision_disk_point = fun ( ( a, b ), r ) -> fun ( x, y ) ->
    ( x -. a ) ** 2. +. ( y -. b ) ** 2. <= r ** 2.
;;

(* Tests *)
let c1 = ( ( 1., 1. ), 1. ) ;;
let p1 = ( 0.5, 0.5 ) ;;
assert ( collision_disk_point c1 p1 ) ;;
(*
```

4.18 Fonction clip

```
*)

(* Donne pour une liste de coordonnées le QuadTree correspondant
 * mais cette fois on spécifie la taille initiale et on ajoute que les
 * points qui sont dans ce rectangle (BONUS) *)
let quadtree_of_list' = fun l -> fun r ->
    let q = Q ( r, Empty ) in
    let add_if_member = fun q -> fun ( c, n ) ->
        if rect_mem r c then
            insert q c n
        else
            q
    in
    List.fold_left add_if_member q l
;;

(* clip : Donne pour un quadtree q et un rectangle r, un QuadTree correspondant
 * aux éléments de q dans r  *)
let clip = fun q -> fun r ->
    let l = list_of_quadtree q in
    quadtree_of_list' l r
;;

(* Tests *)
assert ( boundary ( clip quad1 ( boundary quad1 ) ) = boundary quad1 ) ;;
assert ( list_of_quadtree ( clip quad1 ( boundary quad1 ) ) = list1 ) ;;
assert ( clip quad1 ( boundary quad1 ) = quad1 ) ;;
assert ( clip quad2 ( boundary quad2 ) = quad2 ) ;;
assert ( clip quad3 ( boundary quad3 ) = quad3 ) ;;
let lqc1 = list_of_quadtree ( clip quad3 ( boundary quad1 ) ) ;;
let lqc2 = list_of_quadtree quad1 ;;
assert ( lqc1 = lqc2 ) ;;
(*
```

4.19 Fonction collision_disk

```
*)
(* collision_disk : Donne pour un QuadTree et un disque, la liste des points
 * appartenant au disque *)
let collision_disk = fun q -> fun d ->
   let ( ( a, b ), r ) = d in
   let q' = clip q ( R ( ( a -. r, b +. r ), ( a +. r, b -. r ) ) ) in
   let l = list_of_quadtree q' in
   let add_disk_member = fun acc -> fun e ->
      let (c, _) = e in
      if ( collision_disk_point d c ) then
         e::acc
      else
         acc
   in
   List.fold_left add_disk_member [] l
;;

(* Tests *)
assert ( collision_disk quad1 ( cA, 1. ) = [ ( cA, 'A' ) ] ) ;;
(*
```

4.20 Commentaire du fichier simulation1.ml

La fonction simulation_placement prend un quadtree en paramètre ainsi qu'une fonction de conversion en chaine des données de ce quadtree et effectue les actions suivantes :

- Récuération du rectangle correspondant au QuadTree
- Utilisation de la fonction init pour générer le cadre du graphe
- Dessin récursif du QuadTree grace à la fonction draw_quadtree
- Récupèration d'un disque grace à la fonction get_disk
- Affichage du disque grace à la fonction draw_disk_with_collisions
- Attente de la saisie d'un touche grace à la fonction wait_and_quit

La fonction get_disk prend en paramètre les coordonnées à l'origine et la largeur de l'espace d'affichage et effectue les actions suivantes :

- Attente de l'appui sur le bouton de la souris
- Récupération de la positions de la souris comme centre du disque

- Attente du relachement du bouton de la souris
- Récupération de la position de la souris comme extrémité du disque
- Calcul du rayon du disque
- Retourne le disque ainsi généré

La fonction draw_disk_with_collisions prend en paramètre un quadtree et un disque et effectue les actions suivantes :

- Récupération la liste des points contenus dans le disque
- Définition de la couleur verte si la liste est vide
- Définition de la couleur jaune si la liste n'est pas vide
- Affichage du disque grace à la fonction draw_disk
- Affichage des points de la liste en rouge
- Retourne true si la liste est vide et false sinon

La fonction draw_disk prend en paramètres un disque, une couleur et un booléen et effectue les actions suivantes :

- Positionnement du disque dans l'espace d'affichage
- Définition de la couleur à rouge
- Affichage d'un disque si le booléen est à true
- Affichage d'un cercle si le booléen est à false

Inclusion du fichier simulation1.ml

```
*)
(* #use "simulation1.ml" *)

(* get_point: float * float * float -> coord
 *)
let get_point = fun (sx,sy,z) ->
    let st = Graphics.wait_next_event [ Graphics.Button_down ] in
    let x = (float_of_int st.Graphics.mouse_x -. sx) /. z in
    let y = (float_of_int st.Graphics.mouse_y -. sy) /. z in
    x, y
;;

(* get_disk: float * float * float -> coord * float
 *)
let get_disk = fun (sx,sy,z) ->
    let st = Graphics.wait_next_event [ Graphics.Button_down ] in
    let x = (float_of_int st.Graphics.mouse_x -. sx) /. z in
    let y = (float_of_int st.Graphics.mouse_y -. sy) /. z in
    let st' = Graphics.wait_next_event [ Graphics.Button_up ] in
    let x' = (float_of_int st'.Graphics.mouse_x -. sx) /. z in
    let y' = (float_of_int st'.Graphics.mouse_y -. sy) /. z in
    let r = sqrt ((x-.x')**2.0 +. (y-.y')**2.0) in
    (x, y), r
;;

(* draw_disk: float * float * float -> (coord * float) -> Graphics.color
 *           -> bool -> unit
 *)
let draw_disk = fun (sx,sy,z) ((x,y),r) col full ->
    let xr = int_of_float (sx+.z*.x) in
    let yr = int_of_float (sy+.z*.y) in
    let rr = int_of_float (z*.r) in
    let _ = Graphics.set_color col in
    (if full then Graphics.fill_circle else Graphics.draw_circle) xr yr rr
;;

(* draw_disk_with_collisions: float * float * float -> ('a -> string)
 *                          -> 'a quadtree -> coord * float -> bool
 *)
let draw_disk_with_collisions = fun dparams f qt disk ->
    let l = collision_disk qt disk in
    let b, col =
        match l with
        | [] -> true, Graphics.green
        | _ -> false, Graphics.yellow
    in
    let _ = draw_disk dparams disk col true in
    let _ = List.map (fun data -> draw_data dparams f data Graphics.red) l in
    b
;;

(* simulation_placement: 'a quadtree -> ('a -> string) -> unit
 *)
let simulation_placement = fun qt f ->
    let dparams = init (boundary qt) in
    let _ = draw_quadtree dparams f qt in
    let disk = get_disk dparams in
    let _ = draw_disk_with_collisions dparams f qt disk in
    wait_and_quit ()
;;

(* Tests *)
(* simulation_placement quad1 Char.escaped ;; *)
(*
```

5. Déplacement du disque et détection de collision

5.21 Fonction collision_trail_point

```
*)
(* Indique pour deux points, un rayon et un autre point si le dernier point
 * fait partie de la bande de rayon r centrée sur le segment reliant les
 * deux autres *)
let collision_trail_point = fun p -> fun q -> fun r -> fun m ->
    let ( a, b ) = m in
    let ( x, y ) = p in
    let ( x', y' ) = q in
    let pmpq = ( a -. x ) *. ( x' -. x ) +. ( b -. y ) *. ( y' -. y ) in
    let qmqp = ( a -. x' ) *. ( x -. x' ) +. ( b -. y' ) *. ( y -. y' ) in
    let n = abs_float ((y' -. y) *. a -. (x' -. x) *. b +. x' *. y -. y' *. x) in
    let d = sqrt ((x' -. x) ** 2. +. (y' -. y) ** 2.) in
    if ( n /. d <= r && pmpq >= 0. && qmqp >= 0. ) then
        true
    else
        false
;;

(* Tests *)
assert ( not ( collision_trail_point cD cB 1. cA ) ) ;;
assert ( collision_trail_point cA cB 1. cD ) ;;
(*
```

5.22 Fonction collision_trail

```
*)
(* collision_trail : Donne pour un quadtree, deux points et un rayon la liste des
 * points balayés par le déplacement d'un disque entre les deux points *)
let collision_trail = fun quad -> fun p -> fun q -> fun r ->
   let l = list_of_quadtree quad in
   let add_member = fun acc -> fun e ->
      let (c, _) = e in
      if ( collision_disk_point ( p, r ) c
         || collision_disk_point ( q, r ) c
         || collision_trail_point p q r c )
      then
         e::acc
      else
         acc
   in
   List.fold_left add_member [] l
;;

(* Tests *)
let list3 = [ ( cB, 'B' ); ( cC, 'C' ) ] ;;
assert ( collision_trail quad1 cC cB 0.5 = list3 ) ;;
assert ( collision_trail quad1 cC cB 1. = ( cD, 'D' )::list3 ) ;;
(*
```

5.23 Commentaire du fichier simulation2.ml

La fonction simulation_move prend un quadtree en paramètre ainsi qu'une fonction de conversion en chaine des données de ce quadtree et effectue les actions suivantes :

- Récuération du rectangle correspondant au QuadTree
- Utilisation de la fonction init pour générer le cadre du graphe
- Dessin récursif du QuadTree grace à la fonction draw_quadtree
- Récupèration d'un disque grace à la fonction get_disk:
- Affichage du disque grace à la fonction draw_disk_with_collisions
- Si pas de collision :
- Récupération des coordonnées du curseur de souris lors d'un clic
- Récupération du rayon de disque de départ
- Effacement du graphique

- Dessin d'un cercle vert en lieu et place du disque de départ
- Dessin du quadtree
- Dessin du chemin grace à la fonction draw_trail_with_collisions
- Dessin du disque d'arivée grace à la fonction draw_disk_with_collisions
- Sinon :
- Attente de la saisie d'un touche grace à la fonction wait_and_quit

La fonction draw_trail_with_collisions prend un quadtree en paramètre ainsi qu'un disque de départ et un point d'arrivée et effectue les actions suivantes : *Recupération des points inclus grace à la fonction collision_trail* Dessin du chemin en magenta * Dessin des points inclus en cyan

Inclusion du fichier simulation2.ml

```
*)
(* draw_trail_with_collisions: float * float * float -> ('a -> string)
 *                          -> 'a quadtree -> coord * float -> coord -> bool
 *)
let draw_trail_with_collisions =  fun dparams f qt ((x0,y0), r) (x1,y1) ->
    let (sx,sy,z) = dparams in
    let a = y1 -. y0 in
    let b = x0 -. x1 in
    let alpha = 2.0 *. atan (a /. (sqrt (a ** 2.0 +. b ** 2.0) -. b)) in
    let l = collision_trail qt (x0,y0) (x1,y1) r in
    let _ = Graphics.set_color Graphics.magenta in
    let x0r = int_of_float (sx +. z *. (x0 -. r *. sin alpha) ) in
    let y0r = int_of_float (sy +. z *. (y0 +. r *. cos alpha) ) in
    let x1r = int_of_float (sx +. z *. (x1 -. r *. sin alpha) ) in
    let y1r = int_of_float (sy +. z *. (y1 +. r *. cos alpha) ) in
    let x0r' = int_of_float (sx +. z *. (x0 +. r *. sin alpha) ) in
    let y0r' = int_of_float (sy +. z *. (y0 -. r *. cos alpha) ) in
    let x1r' = int_of_float (sx +. z *. (x1 +. r *. sin alpha) ) in
    let y1r' = int_of_float (sy +. z *. (y1 -. r *. cos alpha) ) in
    let _ = Graphics.moveto x0r y0r in
    let _ = Graphics.lineto x1r y1r in
    let _ = Graphics.lineto x1r' y1r' in
    let _ = Graphics.lineto x0r' y0r' in
    let _ = Graphics.lineto x0r y0r in
    let _ = List.map (fun data -> draw_data dparams f data Graphics.cyan) l in
    l = []
;;

(* simulation_move: 'a quadtree -> ('a -> string) -> unit
 *)
let simulation_move = fun qt f ->
    let dparams = init (boundary qt) in
    let _ = draw_quadtree dparams f qt in
    let disk = get_disk dparams in
    if draw_disk_with_collisions dparams f qt disk
    then
        let (xd,yd) = get_point dparams in
        let r = snd disk in
        let _ = Graphics.clear_graph () in
        let _ = draw_disk dparams disk Graphics.green false in
        let _ = draw_quadtree dparams f qt in
        let _ = draw_trail_with_collisions dparams f qt disk (xd,yd) in
        let _ = draw_disk_with_collisions dparams f qt ((xd,yd),r) in
        wait_and_quit ()
    else
        wait_and_quit ()
;;
(*
```

Tests

```
*)
(* Tests *)
(* simulation_move quad1'' Char.escaped ;; *)
(*
```

Bonus : Fonction simulation_move recursive

```
*)
(* Fonction simulation_move recursive (BONUS) *)
(* simulation_move' : Poursuit le déplacement jusqu'à rencontrer une collision *)
let rec simulation_move' = fun dparams -> fun qt -> fun f -> fun disk ->
    if draw_disk_with_collisions dparams f qt disk
    then
        let ( xd, yd ) = get_point dparams in
        let r = snd disk in
        let _ = Graphics.clear_graph () in
        let _ = draw_disk dparams disk Graphics.green false in
        let _ = draw_quadtree dparams f qt in
        if draw_trail_with_collisions dparams f qt disk ( xd, yd ) then
            simulation_move' dparams qt f ( ( xd, yd ), r )
        else
            wait_and_quit ()
    else
        wait_and_quit ()
;;

(* simulation_move'' : Dessine le disque initial puis passe la main à la fonction
 * recursive *)
let simulation_move'' = fun qt -> fun f ->
    let dparams = init (boundary qt) in
    let _ = draw_quadtree dparams f qt in
    let disk = get_disk dparams in
    simulation_move' dparams qt f disk
;;

(* Tests *)
simulation_move'' quad1'' Char.escaped ;;
(*
```

*)