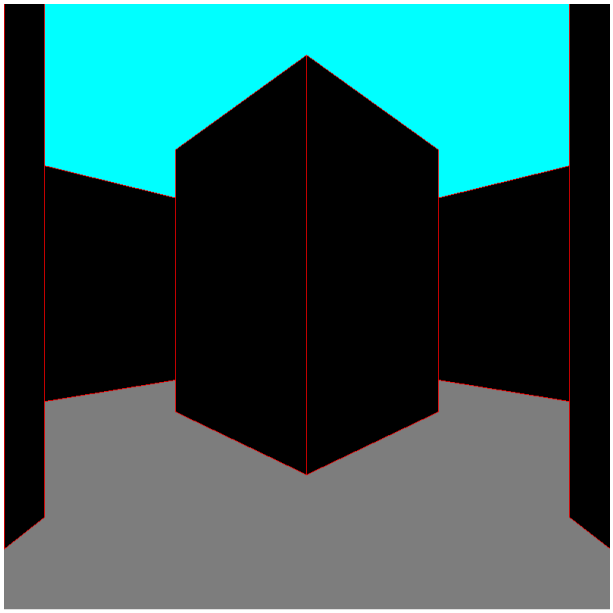
Projet de programmation de PFA

Moteur graphique *à la DOOM*

Mattias Roux
roux@lri.fr

Encadrement :

Stefania Dumbravă
dumbrava@lri.fr



2 mai – 15 mai 2016

Table des matières

Table des figures

1 Présentation du projet

Le but de ce projet est de vous faire créer un moteur de jeu (appelé aussi *game engine*). Certains de ces moteurs sont aujourd'hui très connus dans le monde vidéoludique. On peut notamment citer les moteurs *Source* de *Valve*, l'*Unreal Engine* d'*Epic Games* ou encore, et c'est ce moteur qui nous intéressera particulièrement, le *Doom Engine* aussi connu comme l'*id Tech 1* d'*id Software*.

Il est, bien évidemment, tout à fait illusoire de penser que vous allez recoder ici l'intégralité du moteur de jeu de *Doom*. Votre objectif va être d'implémenter une version minimale d'un moteur de jeu afin de vous faire une idée de la brique de base des jeux vidéos modernes et des algorithmes utilisés dans ses différentes strates.

La structure d'un moteur de jeu

Le moteur 3D 2	Arbres BSP* 2.1	Projections* 2.3	Clipping* 2.4	...
Le moteur physique 3	Déplacements*	Collisions*	Gravité	...
Le moteur de son	Musique	Ambiance		...
Autres	Gestion de l'IA	Gestion des textures		...

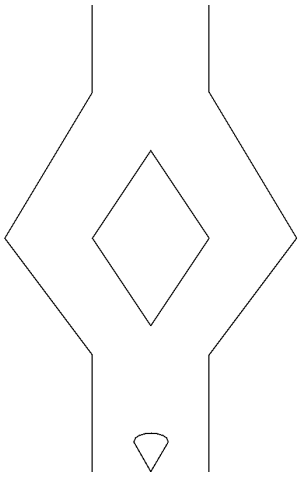
Les parties marquées d'une étoile (*) constituent la base du travail demandé pour ce projet. Les sections s'y référant servent surtout d'indication quant à l'ordre de développement de votre projet. Il est néanmoins tout à fait possible d'implémenter le moteur physique avant le passage à la 3D, par exemple.

Pour finir cette introduction, vous êtes invités, avant de commencer à coder, à lire l'integralité du sujet afin de vous faire une idée précise de ce qui vous est demandé et de bien organiser le travail à venir.

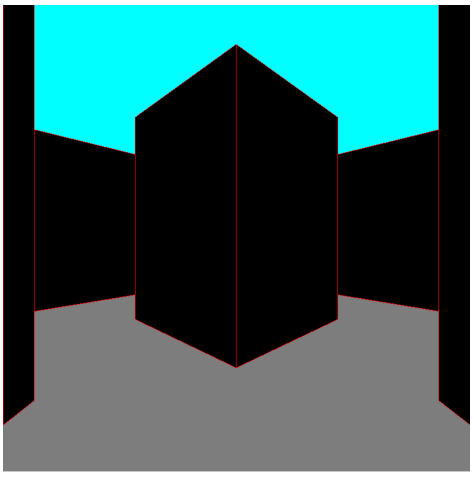
2 Le moteur 3D

Autant vous prévenir tout de suite, *Doom* n'est pas un jeu en 3D. C'est ce qu'on appelle un jeu en *2.5D* car le joueur peut se déplacer sur le plan (avant, arrière, gauche, droite etc.) mais ne peut regarder ni vers le haut ni vers le bas. Le monde de *Doom* est donc un plan qu'on projette sur un monde tridimensionnel.

Votre programme traite des fichiers représentant un monde en 2D et simule une troisième dimension en rajoutant la hauteur lors du rendu :



(a) Ce que voit le programme

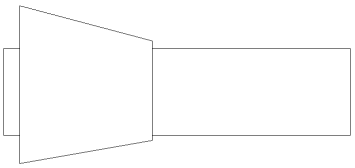


(b) Ce que voit le joueur

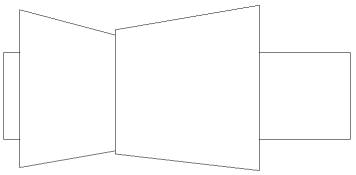
FIGURE 1 – De la 2D à la 3D

Pour obtenir cet affichage, nous allons mettre en œuvre l´algorithme du peintre. Cet algorithme consiste à dessiner les objets d´une scène des plus éloignés aux plus proches.

(a) Dessin du premier mur



(b) Dessin du deuxième mur



(c) Dessin du troisième mur

FIGURE 2 – Une exécution de l'algorithme du peintre

2.1 Les arbres BSP

Plusieurs techniques existent pour appliquer cet algorithme mais la plus courante est la partition binaire de l'espace. Comme son nom l'indique, cette partition sépare le monde en deux sous-mondes, chacun étant lui-même une partition binaire de l'espace. Cela revient à dire qu'avant de vous occuper d'afficher des murs, vous allez les organiser. L'algorithme pour créer un arbre BSP est le suivant :

Algorithme

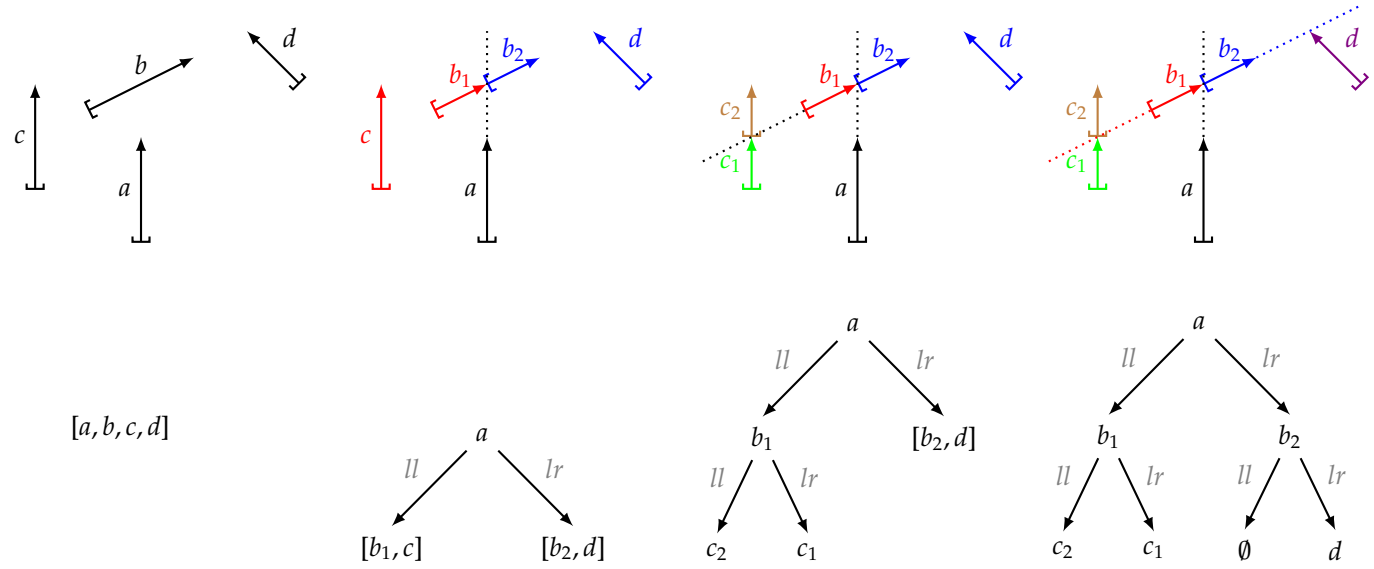
Soit l l'ensemble des segments orientés [a] à traiter,

- On choisit s dans l (si l est vide on renvoie un arbre vide)
- On sépare le reste des segments de l en deux parties, [b]

  ll : l'ensemble des segments qui sont à gauche de s

  lr : l'ensemble des segments qui sont à droite de s
- On recommence le même processus avec ll et lr et on récupère deux arbres BSP L et R
- On crée l'arbre BSP dont la racine est s, le sous-arbre gauche est L et le sous-arbre droit R

a. Il faut nécessairement définir une orientation aux segments puisque on va devoir définir ce qui est à gauche et à droite de ceux-ci.
b. Si un segment c est à la fois à droite et à gauche de s, on le sépare en deux sous-segments c_1 et c_2 tels que c_1 est à gauche de s et c_2 à droite.



(a) Etat initial du monde : quatre murs

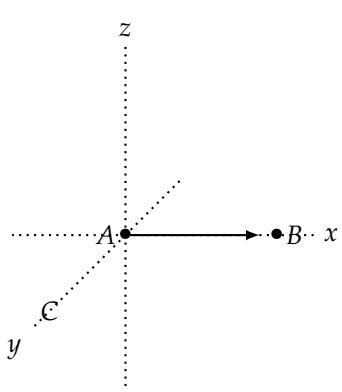
(b) Première partition par rapport au segment a, b est coupé en deux parties.

(c) Deuxième partition par rapport au segment b_1, c est coupé en deux parties.

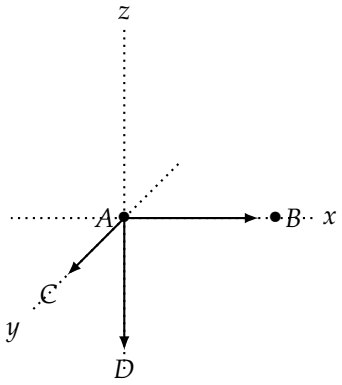
(d) Troisième et dernière partition par rapport au segment b_2. Tous les segments ont été traités.

FIGURE 4 – Exécution de l'algorithme de création de BSP

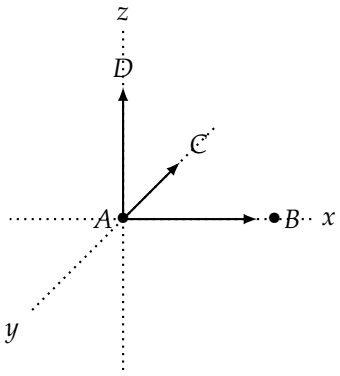
Maintenant que l'algorithme est fourni, il faut savoir comment déterminer si un segment est à gauche ou à droite d'un autre. Pour cela, nous allons déjà nous poser la question de la position d'un point par rapport à un segment et quoi de mieux qu'un peu de géométrie ?



(a) Soit la figure suivante
(C est à droite de $\overrightarrow{AB}$)



(b) Soit $\overrightarrow{AD} = \overrightarrow{AB} \times \overrightarrow{AC}$
On voit que la troisième coordonnée de $\overrightarrow{AD}$ (celle de l'axe (Oz)) est négative



(c) Si C est à gauche de $\overrightarrow{AB}$, la troisième coordonnée de $\overrightarrow{AD}$ est positive [1]

FIGURE 5 – Illustrations du produit vectoriel

Calculer cette coordonnée est relativement aisé. Il suffit pour cela de connaître les coordonnées dans le plan des trois points $A\begin{pmatrix} x_A \\ y_A \end{pmatrix}$, $B\begin{pmatrix} x_B \\ y_B \end{pmatrix}$ et $C\begin{pmatrix} x_C \\ y_C \end{pmatrix}$

On définit $z = (x_B - x_A) * (y_C - y_A) - (y_B - y_A) * (x_C - x_A)$ qui correspond à la troisième coordonnée de $\overrightarrow{AB} \times \overrightarrow{AC}$.

On peut donc conclure avec trois cas possibles :
$$\begin{cases} z < 0 & \text{Le point } C \text{ est à droite de } AB \\ z = 0 & \text{Le point } C \text{ est sur } AB \\ z > 0 & \text{Le point } C \text{ est à gauche de } AB \end{cases}$$

Indications

On donne les types suivant :

```
type Point.t = {x : int; y : int}
type Player.t = {pos : Point.t; angle : int} (* l'angle est en degrés *)
type Segment.t = {id : string; orig : Point.t; dest : Point.t; ci : float; ce : float}
type Bsp.t = E | N of Segment.t * Bsp.t * Bsp.t
```

1. On en conclut que si C est sur la droite AB, cette coordonnée sera nulle.

Précisions

Les champs `ci` et `ce` du type `segment` correspondent aux pourcentages de début et de fin du vrai segment :

soit s tel que		On définit		les vraies coordonnées de s sont donc	
	$s.orig = (x_o, y_o)$				$x_o = l_x \times p_i$
	$s.dest = (x_d, y_d)$		$l_x = x_d - x_o$		$x_d = l_x \times p_e$
	$s.ci = p_i$		$l_y = y_d - y_o$		$y_o = l_y \times p_i$
	$s.ce = p_e$				$y_d = l_y \times p_e$

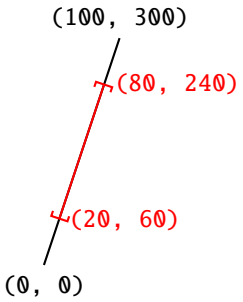


FIGURE 6 – Exemple de représentation d'un segment : `s = {orig = (0, 0); dest = (100, 300); ci = 0.2; ce = 0.8}`

Supposons maintenant que nous ayons deux vecteurs, $\overrightarrow{AB}$: $\begin{pmatrix} x_B - x_A \\ y_B - y_A \end{pmatrix}$ et $\overrightarrow{CD}$: $\begin{pmatrix} x_D - x_C \\ y_D - y_C \end{pmatrix}$ et qu'on veuille déterminer la position de $\overrightarrow{CD}$ par rapport à $\overrightarrow{AB}$. Il nous reste une chose à savoir, c'est si un point de $[CD]$ appartient à la droite (AB).

Pour cela on définit $d = (x_B - x_A) * (y_D - y_C) - (y_B - y_A) * (x_D - x_C)$ et, si $d \neq 0$, $c = \frac{-z}{d}$

On a cinq cas possibles :

$d = 0$	Le segment $[CD]$ et la droite (AB) sont colinéaires
$0 < c < 1$	Il existe un point de $[CD]$ sur la droite (AB)
$c < 0 \vee c > 1$	Il n'existe pas d'intersection entre $[CD]$ et (AB)
$c = 0$	C est sur la droite (AB)
$c = 1$	D est sur la droite (AB)

Une fois la position d'un point et l'intersection entre une droite et un segment comprises, on peut très bien déterminer la position d'un segment par rapport à un autre segment. Huit cas peuvent survenir

Position d'une extrémité	Gauche				Droite			
Cas	$d = 0$	$0 < c < 1$	$c < 0$ $c >= 1$	$c = 0$	$d = 0$	$0 < c < 1$	$c < 0$ $c >= 1$	$c = 0$
Position du segment	Gauche	Des deux côtés	Gauche	Position de l'autre extrémité [2]	Droite	Des deux côtés	Droite	Position de l'autre extrémité [2]

2. On suppose qu'un point situé sur le segment de référence est situé au bon endroit par rapport à l'autre point (par exemple, si une extrémité est située à gauche et l'autre sur le segment référence, alors on considère qu'elle est aussi située à gauche). Si les deux extrémités sont sur le segment référence, il suffit de choisir un côté par convention.

2.2 Implémentation

Modules fournis

Les modules Options, Parser et Main vous sont déjà fournis ainsi que certains fichiers de labyrinthes dans le répertoire `labyrinthes`.
La structure de ces fichiers doit être la suivante :

```
P : [0-9]+ [0-9]+ [0-9]+
([0-9]+ [0-9]+ [0-9]+ [0-9]+\n)*
```

Si vous appelez, par exemple, votre programme avec la commande

```
./projet_pfa lab1.lab
```

le point d'entrée () du module Main.main commence par appeler Parser.read_lab ci qui lui renvoie un couple (p, l) : (int * int * int) * (int * int * int * int) list, p représente les coordonnées (x, y, *angle*) du joueur et l une liste de coordonnées (x_o, y_o, x_d, y_d) correspondant aux murs de votre labyrinthe.

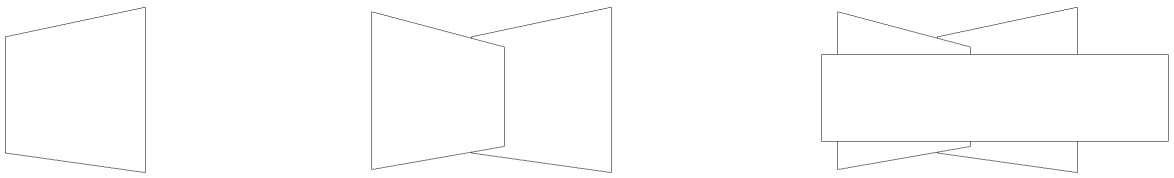
Travail à faire

- Implémentez le module Point en rapport avec sa signature qui est donnée dans la documentation (vous pouvez directement cliquer sur le nom du module),
- Implémentez le module Player en rapport avec sa signature,
- En vous aidant des formules précédentes, implémentez le module Segment,
- En vous aidant de l'algorithme précédent, implémentez le module Bsp.

Il est évident que les signatures sont des guides et votre implémentation pourra comporter d'autres fonctions, types etc. De plus, certaines fonctions présentes dans les signatures ne s'avèrent utiles que plus tard dans l'avancement du projet, il n'est donc pas nécessaire de les implémenter immédiatement.

Important

Une fois arrivés ici, prenez garde ! Votre partition est l'élément le plus important de votre programme. Le rendu 3D et la gestion des collisions, entre autres, s'appuient en grande partie dessus. Il vous est donc très fortement conseillé de vérifier qu'elle est parfaite sans quoi vous obtiendrez des bogues dont l'origine remontera au tout début de votre projet.



(a) Dessin du premier mur (b) Dessin du deuxième mur (c) Dessin du troisième mur

FIGURE 7 – Une exécution de l'algorithme du peintre sur une partition erronée

2.3 Opérations en 2D

Arrivés ici, vous devriez avoir un arbre de murs et un joueur.

Important

On représente

- Le joueur par le tuple (x_p, y_p, α)
- Un segment s par le tuple $(x_o^s, y_o^s, x_d^s, y_d^s)$ (coordonnées de départ et d'arrivée).

Pour l'instant, votre programme a assez peu d'informations qu'on peut représenter ainsi [3] :

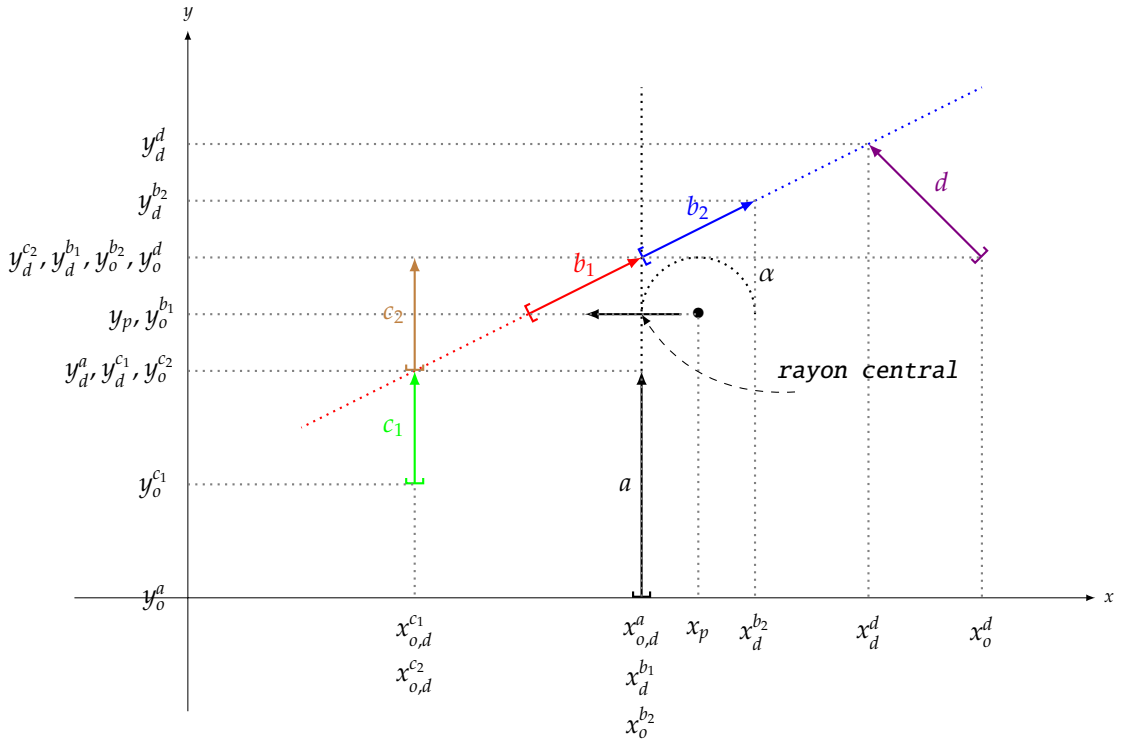


FIGURE 8 – État courant du programme après calcul du BSP correspondant au fichier

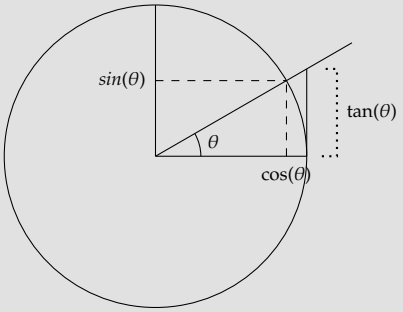
```
P : 450 250 180
400 0 400 200
300 250 500 350
200 100 200 300
700 300 600 400
```

3. Chaque figure de cette section représente une vue de dessus.

2.3.1 Translation et rotation

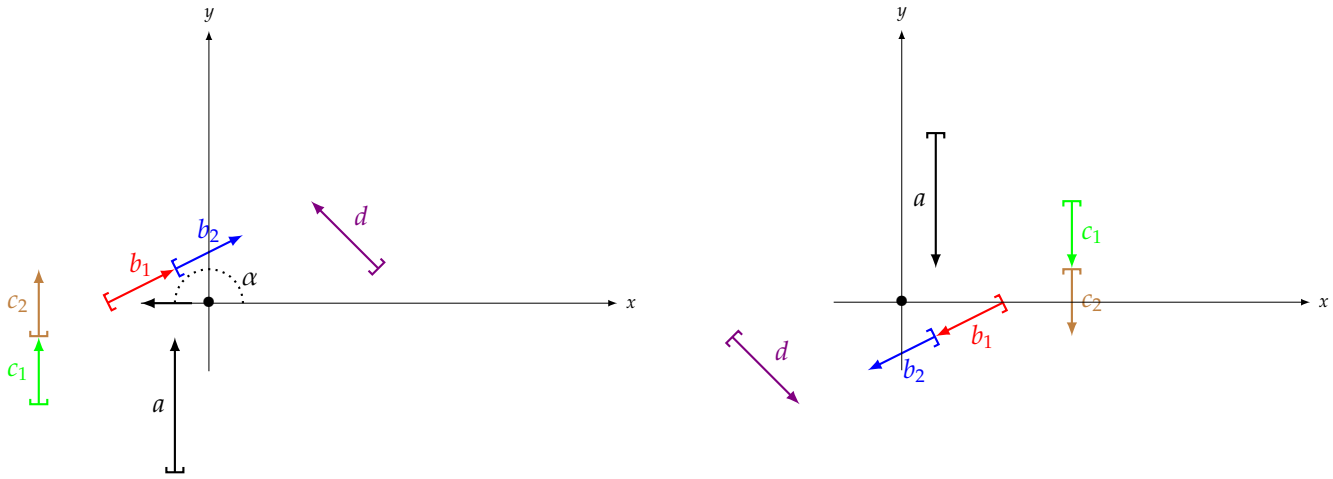
Indications

On rappelle certaines propriétés de trigonométrie. Tout résultat non expliqué est issu d'une de ces propriétés



La première chose que nous voulons faire, car ceci simplifiera le reste des opérations à venir, c'est de se centrer sur le joueur. Pour cela, il faut que les coordonnées x_p, y_p soient l'origine du nouveau repère et que le rayon central représente l'axe des abscisses.

Il faut donc faire une translation de vecteur $\overrightarrow{v}$: $\begin{pmatrix} -x_p \\ -y_p \end{pmatrix}$ puis une rotation plane de centre $(0,0)$ et d'angle $-\alpha$.



(a) translation de vecteur $\overrightarrow{v}$: $\begin{pmatrix} -x_p \\ -y_p \end{pmatrix}$

(b) rotation plane de centre $(0,0)$ et d'angle $-\alpha$

FIGURE 9 – Translation et rotation autour du joueur

Indications

Soient $\begin{cases} x_p, y_p, \alpha & \text{les coordonnées du joueur} \\ x_o^s, y_o^s, x_d^s, y_d^s & \text{les coordonnées d'un segment } s \end{cases}$

Alors le segment s' résultant de la translation de vecteur $\overrightarrow{v}$: $\begin{pmatrix} -x_p \\ -y_p \end{pmatrix}$ puis de la rotation plane de centre $(0,0)$ et d'angle $-\alpha$ a pour coordonnées

$$\begin{cases} x_o^{s'} &= (x_o^s - x_p) * \cos(-\alpha) - (y_o^s - y_p) * \sin(-\alpha) \\ y_o^{s'} &= (y_o^s - y_p) * \cos(-\alpha) + (x_o^s - x_p) * \sin(-\alpha) \\ x_d^{s'} &= (x_d^s - x_p) * \cos(-\alpha) - (y_d^s - y_p) * \sin(-\alpha) \\ y_d^{s'} &= (y_d^s - y_p) * \cos(-\alpha) + (x_d^s - x_p) * \sin(-\alpha) \end{cases}$$

2.3.2 Clipping

Regardons maintenant plus précisément la figure obtenue et profitons-en pour définir l'*angle d'un segment* :

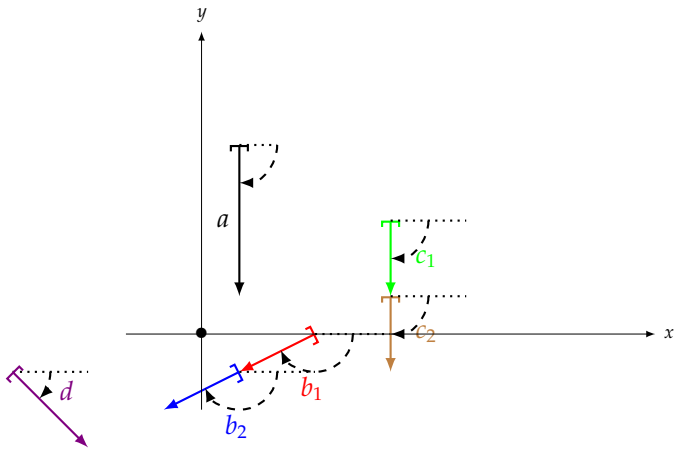


FIGURE 10 – État actuel du programme et définition de l'*angle d'un segment* [4]

L'objectif principal de tout moteur de rendu, est de faire le moins de calculs coûteux possibles. Comme on peut le voir dans la figure 10, il existe déjà des segments dont nous sommes sûrs qu'ils ne seront pas visibles par le joueur. Nous allons donc *clipper* les segments pour ne garder que ceux qui sont a priori visibles (on ne s'embarasse pas encore de la notion de champ de vision) pour supprimer tout ce qui se trouve *derrière* le joueur. *Derrière* est en italique car pour des raisons de simplification des calculs, il vaut mieux définir que tout ce qui est à une distance inférieure à 1 (plutôt que 0) se trouve *derrière* (pour de sombres raisons de divisions par 0 qui pourraient malheureusement survenir).

Algorithme

Soient $(x_o^s, y_o^s, x_d^s, y_d^s)$ les coordonnées d'un segment s et ta la tangente de l'angle du mur :

si $x_o^s < 1 \wedge x_d^s < 1$	alors on supprime ce segment des segments à traiter
sinon si $x_o^s < 1$ [a]	alors les nouvelles coordonnées du segment deviennent $(1, y_o^s + (1 - x_o^s) * ta, x_d^s, y_d^s)$
sinon si $x_d^s < 1$ [b]	alors les nouvelles coordonnées du segment deviennent $(x_o^s, y_o^s, 1, y_d^s + (1 - x_d^s) * ta)$
sinon	le segment ne change pas

On pensera aussi à faire en sorte que si le segment est trop loin on le rapproche sans faire d'autre changement (ce que fait notre œil naturellement, tous les objets trop éloignés ayant à peu près les mêmes proportions) car on risque de dépasser les entiers représentables en OCaml [c].

a. Donc $x_d^s \geq 1$
b. Donc $x_o^s \geq 1$
c. Ceci revient à dire que si $x_o^s > x_{max}$ alors les nouvelles coordonnées deviennent $(x_{max}, y_o^s, x_d^s, y_d^s)$ et idem pour $x_d^s > x_{max}$

4. Pour chaque segment s, son angle est l'arc tangente du rapport $\frac{y_d^s - y_o^s}{x_d^s - x_o^s}$.

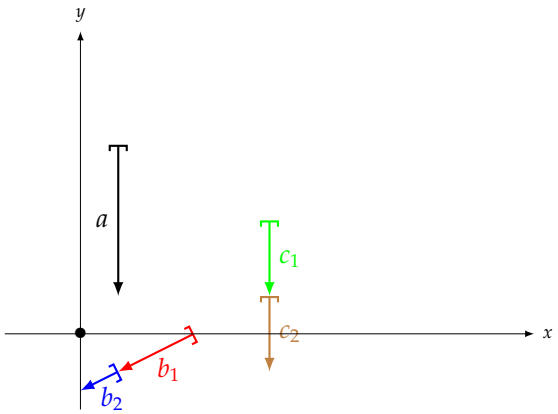


FIGURE 11 – État du programme après le clipping

2.3.3 Projection horizontale

Après cette partie, nous serons enfin arrivés à la fin du travail en deux dimensions. Il va falloir se concentrer sur le champ de vision du joueur donc sur la distance focale et la taille de notre *écran*.

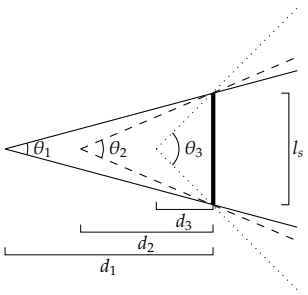


FIGURE 12 – Rapport entre la distance focale d_i, l'angle d'ouverture θ_i et la largeur de l'écran l_s

l_s ne variant pas (ou que très légèrement), on peut contrôler l'angle de vision en jouant sur la distance focale ou inversement. De façon générale, on préfère définir l'angle de vision et en déduire la distance focale grâce à l'équation suivante $d = \frac{l_s/2}{\tan(\theta/2)}$.

Nous devons par conséquent trouver la largeur des murs par rapport au joueur, à θ et à d. Pour cela, reprenons notre exemple où il en est et rajoutons le champ de vision ainsi que l'écran :

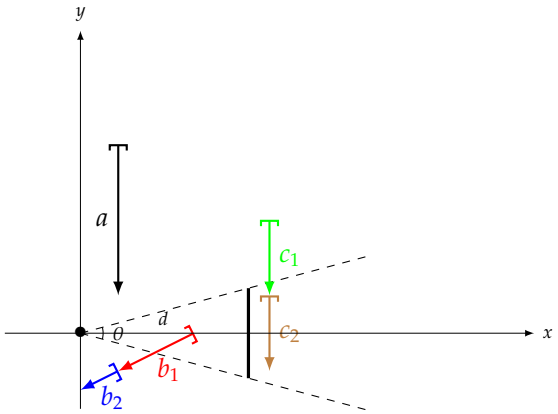
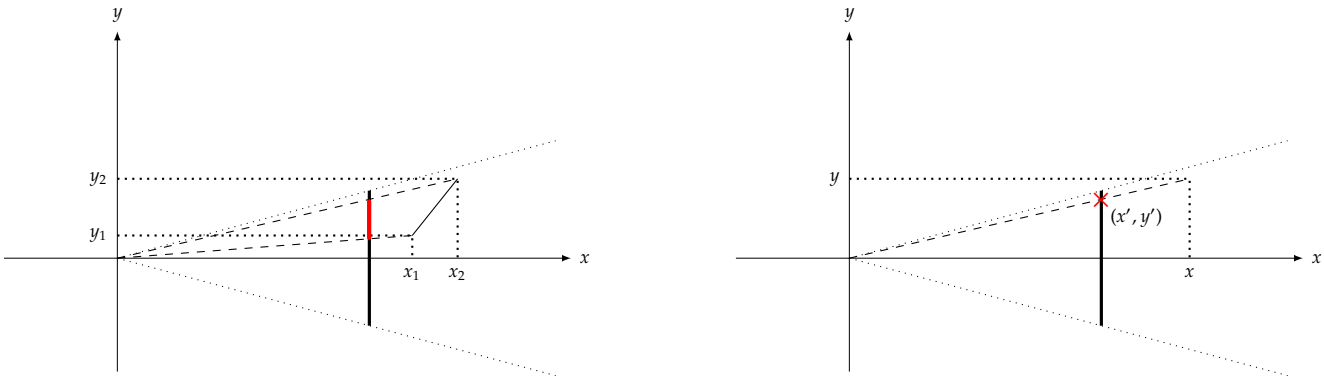


FIGURE 13 – État du programme avant projection horizontale

Quelle est, ici, la seule chose que nous savons ? Que l'abscisse d'un point projeté sur l'écran est égale à d. On peut représenter cela comme dans la figure 14



(a) Idée derrière la projection d'un segment

(b) Idée derrière la projection d'un point

FIGURE 14 – Projeté d'un segment et d'un point sur l'écran

En faisant disparaître la profondeur, on fait disparaître les abscisses des points. C'est le principe d'un jeu en trois dimensions. Étant donné que l'écran est en deux dimensions, on fait disparaître la profondeur qu'on simule par une projection conforme aux règles de la perspective pour garder la largeur et la hauteur. Nous nous concentrons donc ici sur la largeur d'un mur. Il nous suffit, pour un point (x, y), de calculer ses nouvelles coordonnées (x', y') sachant que $x' = sd$, on obtient, grâce à Thalès : $\frac{x'}{y'} = \frac{x}{y}$ donc $y' = \frac{y * x'}{x} = \frac{y * d}{x}$ et il suffit d'appliquer cette opération à chaque extrémités de chaque segment affichable.

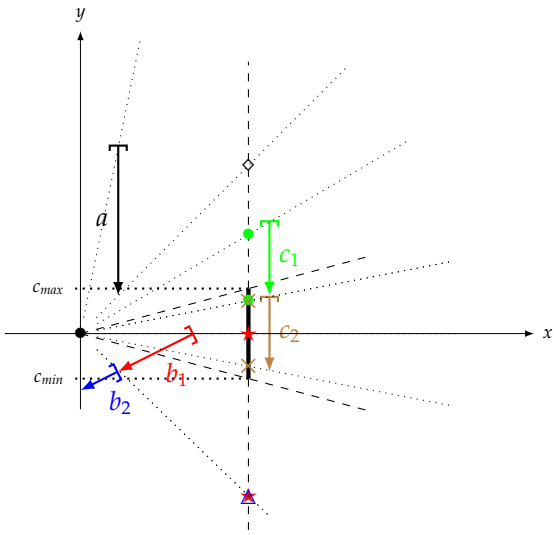


FIGURE 15 – État du programme après projection horizontale. Les ordonnées sont appelées *colonnes* car elles correspondent aux *colonnes* de pixels de l'écran.

Un petit ajustement est nécessaire, lié au fait que la colonne 0 de la figure correspond en réalité à la colonne centrale de l'écran. c_{max} doit ainsi correspondre en réalité à la colonne 0 et c_{min} à la colonne l_s. On obtient alors que $y' = \frac{l_s}{2} - \frac{y * d}{x}$

Certains segments peuvent être immédiatement éliminés, ce sont ceux dont les valeurs des extrémités des colonnes sont soit toutes deux négatives, soit toutes deux supérieures à l_s (ce qui correspond, dans notre exemple, aux segments a et b_2). Le segment c_2, lui, sera intégralement visible mais les segments b_1 et c_1 ne le seront qu'en partie. Néanmoins, il vaut mieux éviter de les clipper immédiatement car ces coordonnées vont nous aider pour la projection verticale.

2.4 Le passage à la 3D

Enfin, nous y voici arrivés, nous allons calculer la hauteur de nos segments ! Voici l'état de notre programme :

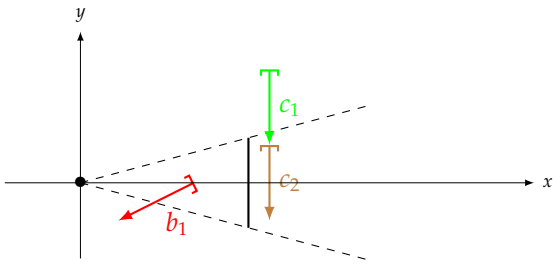
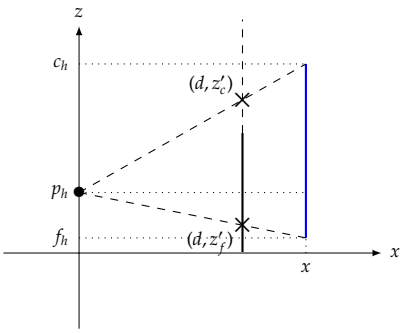


FIGURE 16 – État du programme après projection horizontale et filtrage des segments invisibles.

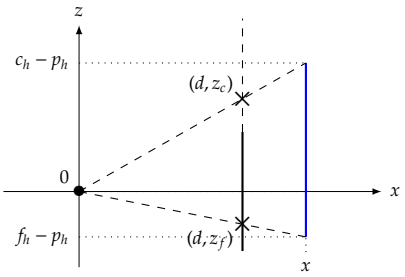
Nous allons abandonner la vue de dessus pour passer à une vue de côté en nous intéressant à l'extrémité d'un mur.

Pour cela nous avons besoin de trois nouvelles valeurs, $\begin{cases} p_h & \text{la hauteur des yeux du joueur} \\ f_h & \text{la hauteur du plancher (\textit{floor})} \\ c_h & \text{la hauteur du plafond (\textit{ceiling})} \end{cases}$

Les murs étant compris exactement entre le plancher et le plafond et l'écran étant centré sur les pieds du joueur, nous obtenons la figure suivante :



(a) Repère centré sur les pieds du joueur (repère originel)



(b) Repère centré sur les yeux du joueur (le centre de l'écran correspond donc à l'origine du repère).

FIGURE 17 – Projection d'un segment sur l'écran.

Comme on le voit sur la deuxième figure, tout cela tombe très bien. Pour obtenir la hauteur maximale du segment (z_c), par exemple, on remarque que $\frac{c_h - p_h}{z_c} = \frac{x}{d}$ donc $z_c = \frac{(c_h - p_h) * d}{x}$ et, de même, $z_f = \frac{(f_h - p_h) * d}{x}$. Il faut néanmoins se rappeler que le centre de l'écran est en réalité à la moitié de sa hauteur h_s donc on doit réajuster les segments en conséquence. On obtient finalement que pour une extrémité de segment, $z_c = \frac{h_s}{2} + \frac{(c_h - p_h) * d}{x}$ et $z_f = \frac{h_s}{2} + \frac{(f_h - p_h) * d}{x}$.

Il ne nous reste plus qu'à afficher le mur entier en clippant les parties dépassant à gauche et à droite.

Indications

Soit s tel que
- c_o est sa colonne de départ
- c_d est sa colonne d'arrivée
- z_{uo} est la hauteur maximale de son extrémité de départ
- z_{lo} est la hauteur minimale de son extrémité de départ
- z_{ud} est la hauteur maximale de son extrémité d'arrive
- z_{ld} est la hauteur minimale de son extrémité d'arrivée

alors

$$\delta_u = \frac{z_{ud} - z_{uo}}{c_d - c_o} \qquad \delta_l = \frac{z_{md} - z_{lo}}{c_d - c_o}$$

Ainsi :

Algorithme

$$\begin{cases} c_o < 0 \implies c_o \leftarrow 0, \quad z_{uo} \leftarrow z_{uo} - (c_o * \delta_u), & z_{lo} \leftarrow z_{lo} - (c_o * \delta_l) \\ c_o > l_s \implies c_o \leftarrow l_s, \quad z_{uo} \leftarrow z_{uo} - ((c_o - l_s) * \delta_u), & z_{lo} \leftarrow z_{lo} - ((c_o - l_s) * \delta_l) \end{cases}$$

$$\begin{cases} c_d < 0 \implies c_d \leftarrow 0, \quad z_{ud} \leftarrow z_{ud} - (c_d * \delta_u), & z_{ld} \leftarrow z_{ld} - (c_d * \delta_l) \\ c_d > l_s \implies c_d \leftarrow l_s, \quad z_{ud} \leftarrow z_{ud} - ((c_d - l_s) * \delta_u), & z_{ld} \leftarrow z_{ld} - ((c_d - l_s) * \delta_l) \end{cases}$$

On peut maintenant afficher nos murs en utilisant deux fonctions du module Graphics.

Fonctions utiles

val fill_poly : (int * int) array -> unit

Fills the given polygon with the current color. The array contains the coordinates of the vertices of the polygon.

val draw_segments : (int * int * int * int) array -> unit

Draws the segments given in the array argument. Each segment is specified as a quadruple (x0, y0, x1, y1) where (x0, y0) and (x1, y1) are the coordinates of the end points of the segment. The current point is unchanged.

2.5 Implémentation

Fourni

On rappelle la structure basique d'un programme utilisant Graphics :

```
let () =
  open_graph (Printf.sprintf " %dx%d" win_w win_h);
  auto_synchronize false;
  try
    while true do
      attente_dun_evenement ();
      fonctions_de_mise_a_jour_de_la_fenetre arg1 arg2 ...;
      synchronize ();
    done;
  with Exit -> close_graph (); exit 0
```

Travail à faire

- En vous aidant des formules et algorithmes précédents, implémentez le module Render

3 Moteur physique et optimisations

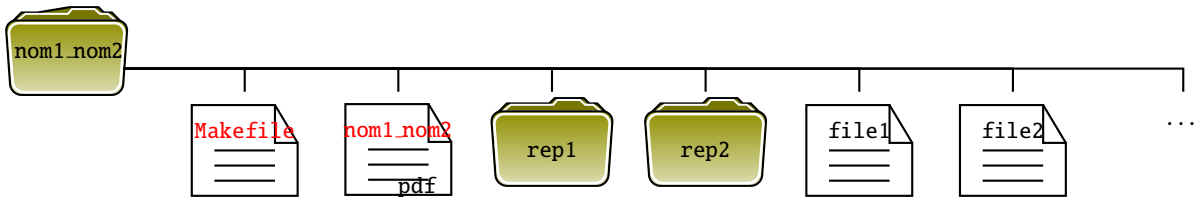
Le moteur physique est laissé à votre imagination mais il faut savoir que les connaissances mathématiques requises sont du même niveau que celles utilisées pour le moteur graphique. Quelques indications néanmoins :

- Les déplacements ne peuvent pas s'effectuer avec les touches fléchées du clavier qui ne sont pas reconnues
- Certaines options sont là pour vous aiguiller
- Pour les collisions, on peut créer une zone rectangulaire autour de chaque mur dont la taille sera judicieusement choisie et vérifier que notre joueur n'est pas dans cette zone (*cf.* équations de droite)
- Pensez, quand vous vous rendez compte que vous utilisez plusieurs fois le même résultat d'un calcul dans différentes fonctions ou appels de fonctions, à sauvegarder votre résultat. L'extensibilité des enregistrements est là pour ça
- N'hésitez pas à poser des questions et à discuter entre vous, la copie est interdite, pas l'entraide

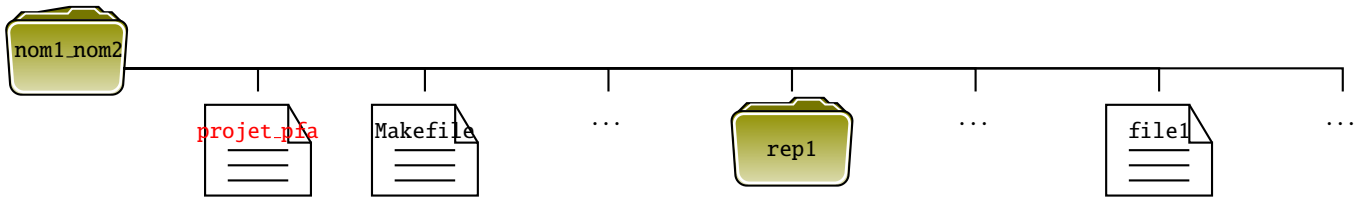
4 Modalités

- Ce projet est à faire en binôme et en OCaml.
- Vous devrez envoyer à votre chargé de TP, *i.e.* | Stefania Dumbravă / *ou* / Mattias Roux | une archive nommée `nom1_nom2.tgz` contenant un unique répertoire nommé `nom1_nom2` [5] et ayant pour sujet : *[Projet PFA] nom1 nom2*.
- Le répertoire `nom1_nom2` pourra contenir les répertoires et fichiers que vous voudrez mais il devra *impérativement* inclure :
  - un fichier `Makefile` tel que la commande `make` permette de compiler votre programme et d'obtenir un exécutable `projet_pfa`
  - le compte-rendu de votre projet nommé `nom1_nom2.pdf` listant les différentes commandes permettant d'exécuter le programme, une description succincte des fichiers importants de votre projet et de vos choix d'implémentation et toute autre information jugée utile. *Ce compte-rendu devra faire environ cinq pages.*

On veut donc un répertoire de cette forme :



tel que la commande `make` donne la configuration suivante :



Dates importantes

Dimanche 15 mai [a]	Envoi de l'archive
Mercredi 18 mai après-midi	Soutenances

a. Donc avant minuit

Indications

Vous serez évalués sur différents points :

- Propreté, clarté, modularité, optimisation, absence de bugs de votre code. Cela implique qu'il faut, par exemple, savoir commenter quand c'est nécessaire mais l'éviter quand c'est inutile.
- Comme dit dans la présentation (*cf* 1), ce projet est constitué d'une base attendue. Il vous est donc conseillé de bien l'implémenter.

5. obtenable avec la commande `tar czvf nom1_nom2.tgz nom1_nom2`

5 Documentation

5.1 Module `Trigo`

```
val dtan : int -> float
```

Tangente. Argument en degrés.

```
val dcos : int -> float
```

Cosinus. Argument en degrés.

```
val dsin : int -> float
```

Sinus. Argument en degrés.

```
val dacos : float -> float
```

Arc cosinus. L'argument doit être entre [-1.0, 1.0] et le résultat est en degrés.

5.2 Module `Options`

```
type tmode = TwoD | ThreeD
val mode : tmode (* Default : TwoD *)
```

On veut pouvoir observer le labyrinthe de dessus (pour, par exemple, vérifier que les collisions ont bien lieu).

```
val cin : Pervasives.in_channel
```

Canal de lecture sur le fichier contenant le labyrinthe traité.

```
val win_w : int (* Default : 800 *)
val win_h : int (* Default : 800 *)
```

Auteur et largeur de la fenêtre graphique.

```
val ceiling_h : int (* Default : win_h / 4 *)
val floor_h : int (* Default : 0 *)
val wall_h : int (* ceiling_h - floor_h *)
```

Hauteur du plafond et du plancher (un mur étant compris entre les deux, sa hauteur est égale à la différence). Ces valeurs sont calculées en fonction de `win_h` donc il faut modifier directement le fichier `options.ml` si on veut changer.

```
val fov : int (* Default : 60 *)
```

Angle de vision.

```
val step_dist : float (* Default : 10 *)
```

Distance parcourue entre deux pas.

```
val xmin : float (* Default : 1 *)
val xmax : float (* Default : 9000 *)
```

Distance minimale pour voir un objet et distance maximale à partir de laquelle tous les objets sont considérés comme étant à la même distance.

```
val scale : int (* Default : 5 *)
val minimap : bool (* Default : false *)
```

Options pouvant servir si on veut appliquer une carte vue de dessus lorsqu'on est en 3D mise à une certaine échelle.

```
val yaw_sensitivity : float (* Default : false *)
```

Sensibilité lors du déplacement de la souris sur l'axe horizontal.

```
val debug : bool (* Default : false *)
val debug_bsp : bool (* Default : false *)
```

Options permettant d'activer ou de désactiver des affichages plus précis afin de faciliter la correction des bugs.

```
type tlang = Fr | En
val lang : tlang (* Default : Fr *)
```

Option pour adapter les touches directionnelles aux différentes configurations de clavier.

5.3 Module `Point` : Point du plan

Un point représente à la fois un joueur, un personnage non joueur (PNJ) ou la coordonnée de départ ou d'arrivée d'un `Segment.t`.

```
type t = {x : int; y : int}
```

Il est déconseillé de créer des points autrement que par cette fonction car si la représentation de ceux-ci venait à changer, cela induirait de nombreuses modifications.

```
val new_point : int -> int -> t
```

Il est déconseillé de créer des points autrement que par cette fonction car si la représentation de ceux-ci venait à changer, cela induirait de nombreuses modifications.

5.4 Module `Segment` : Murs en 2D

```
type t = {
  id : string;
  porig : Point.t;
  pdest : Point.t;
  [...]
}
```

Le type des segments est laissé à votre discrétion. Il vous faudra néanmoins définir un point d'origine et un point d'arrivée afin de donner une orientation à vos segments mais vous serez amené à rajouter des champs au fur et à mesure de l'avancée du projet.

```
type tpos = L | R | C
```

Représente le fait qu'un point soit à gauche (`L`) ou à droite (`R`) ou aligné (`C`) avec un segment.

```
val new_segment : int -> int -> int -> int -> t
```

Un segment est défini par ses points d'origine et de destination.
Bien qu'il semble plus simple de ne pas passer par une fonction pour créer un nouveau segment, au fur et à mesure de l'avancement du programme on se rendra compte de la nécessité de n'avoir à modifier que la fonction de création.

```
val get_position : Point.t -> t -> tpos
```

`get_position p s` renvoie la position du point `p` par rapport au segment `s`. Attention, les segments sont supposés orientés (voir `Segment.t`).

```
val split_segment : t -> t -> t option * t option
```

`split_segment s1 s2` détermine si `s2` est situé à gauche, à droite ou des deux côtés de `s1` et renvoie un couple correspondant à la partie de `s2` se trouvant à gauche de `s1` et celle se trouvant à droite.

```
val split : t -> t list -> t list * t list
```

`split s sl` sépare la liste `sl` en deux sous-listes `sll` et `slr`. `sll` correspond à l'ensemble des segments se trouvant à gauche de `s` et `slr` à l'ensemble se trouvant à droite.

5.5 Module `Bsp` : Partition binaire de l'espace

Un arbre bsp est la représentation d'un ensemble de `Segment.t`[5.4] dans le plan.

```
type t = E | N of Segment.t * t * t
```

Un arbre est composé d'une racine qui sépare les `Segment.t` en deux parties, ceux qui sont à gauche de cette racine et ceux qui sont à droite. Chacune de ces parties est un arbre (un arbre peut être vide).

```
val parse : (Segment.t -> 'a) -> t -> Point.t -> unit
```

`parse f p bsp` parcourt l'arbre en fonction de la position du point `p` par rapport à chaque racine rencontrée. Si `p` est à gauche de la racine, on parcourt le sous-arbre gauche puis on applique `f` à la racine puis on parcourt le sous-arbre droit (en résumé, on applique `f` des murs les plus proches aux plus éloignés).

```
val rev_parse : (Segment.t -> 'a) -> t -> Point.t -> unit
```

`rev_parse f p bsp` parcourt l'arbre en fonction de la position du point `p` par rapport à chaque racine rencontrée. Si `p` est à gauche de la racine, on parcourt le sous-arbre droit puis on applique `f` à la racine puis on parcourt le sous-arbre gauche (en résumé, on applique `f` des murs les plus éloignés aux plus proches).

```
val iter : (Segment.t -> 'a) -> t -> unit
```

`iter f bsp` parcourt l'arbre entier en appliquant `f` à chaque segment. L'ordre de parcours n'est pas spécifié.

```
val build_bsp : Segment.t list -> t
```

`build_bsp sl` parcourt une liste de segments et renvoie un arbre BSP.

5.6 Module `Physic` : Moteur physique

Ce module a pour objectif de calculer les collisions entre un point (représentant, généralement, un joueur) et un des murs du labyrinthe.

```
val detect_collision : Point.t -> Bsp.t -> bool
```

`detect_collision p b` renvoie `vrai` si le point `p` entre en collision avec un segment de l'arbre `b` et faux sinon.

5.7 Module `Player` : Joueur et fonction de mise à jour

```
type t = {
  mutable pos : Point.t;
  mutable pa : int;
  [...]
}
```

Le joueur a une position représentée par un point et un angle correspondant à la direction de son regard. D'autres champs pourront être ajoutés ultérieurement.

```
val new_player : Point.t-> int -> t
```

```
type dir = Left | Right
val rotate : dir -> t -> unit
```

Fonction appelée par le moteur de rendu pour communiquer le changement d'angle du joueur.

```
type mv = MFwd | MBwd | MLeft | MRight
val move : mv -> t -> Bsp.t -> unit
```

Fonction appelée par le moteur de rendu pour communiquer le changement de position du joueur (en faisant attention aux éventuelles collisions).

5.8 Module `Parse_lab`

```
val read_lab : in_channel -> (int * int * int) * (int * int * int * int) list
```

Fonction prenant en entrée un canal de lecture sur un fichier représentant un labyrinthe et renvoyant les coordonnées nécessaires à la représentation du joueur et des murs.

5.9 Module Render : Moteur graphique 2D et 3D

```
val display : Bsp.t -> Player.t -> unit
```

Fonction d'affichage principale.