TP n°8

Graphisme

Le but de ce TP est de mettre en oeuvre le jeu de plateau *Puissance 4*, en soignant en particulier l'interface graphique du jeu [1]. Votre programme permettra à deux joueurs humains de jouer entre eux. L'extension "utilisateur contre ordinateur" est traitée dans la deuxième partie de la feuille.

Règles du jeu

Puissance 4 est un jeu de plateau à deux joueurs. Il se joue sur un plateau vertical de 7×6 cases (Figure 1). Chaque joueur dispose de 21 pions, rouges ou jaunes. Rouge joue le premier, et les joueurs jouent tour-à-tour. Le but du jeu est d'aligner quatre pions de sa couleur, verticalement, horizontalement, ou en diagonale.

Lorsque c'est son tour, le joueur p choisit une colonne qui n'est pas encore pleine, et y insère un de ses pions. Le pion tombe jusqu'à ce qu'il soit coincé par un autre pion ou par le bas de la colonne (Figure 2).

La partie se termine lorsqu'un joueur a gagné (il a aligné quatre pions de sa couleur, Figure 3), ou alors lorsque le plateau est plein (match nul).

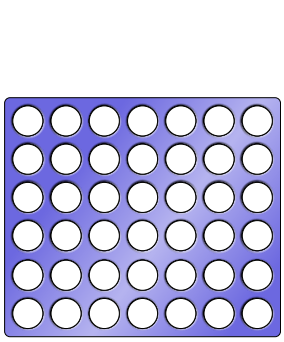

FIGURE 1 – Plateau vide

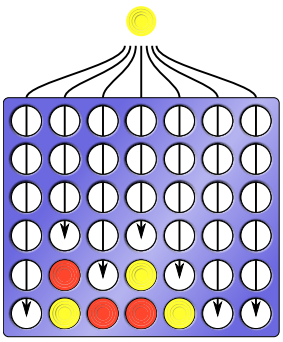

FIGURE 2 – Coups possibles

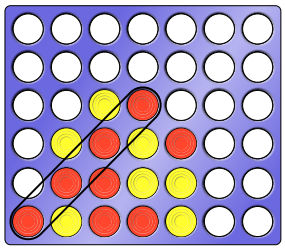

FIGURE 3 – Fin de partie

Le fichier `noyau_p4.ml`

Dans cette première section, vous allez créer un fichier `noyau_p4.ml` contenant certains éléments de base pour la suite du TP. Vous aurez besoin de types suivants (insérez donc les lignes suivantes dans votre fichier) :

```
type player = Red | Yellow;;
```

1. Le jeu *Puissance 4* est un jeu pour les enfants à partir de 7 ans, mais heureusement pour le programmer, il n'y a pas d'âge minimal.

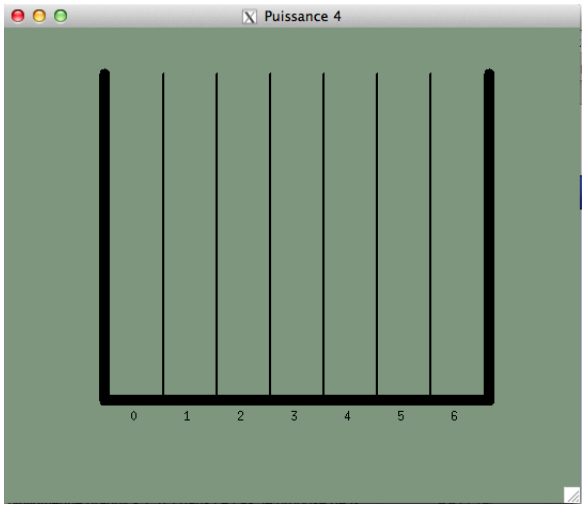

FIGURE 4 – exemple du plateau (Exercice 3).

```
type box  = Empty | P of player;;
type outcome = Draw | Open | W of player ;;
type board = box array array;;
```

Le plateau de jeu est représenté par un tableau de sept éléments : les colonnes. Chaque colonne est un tableau de six éléments de type `box`. Le plateau initial s'obtient ainsi :

`let b = Array.make_matrix 7 6 Empty`

Par exemple : `b.(0).(0)` désigne la première case de la première colonne du plateu. On peut choisir de considérer cette case comme étant celle en haut à gauche, ou en bas à gauche du plateau.

Exercice 1. Écrivez une fonction `val play: board -> player -> int -> int` qui permet de jouer un coup. Dans un appel `play b p i`, `i` désigne la colonne où l'on veut jouer. L'appel modifie `b` et renvoie l'indice de la première case libre de la colonne `i`.

Exercice 2. Écrivez une fonction `val eval: board -> outcome` qui permet de connaître l'état du plateau jeu.

Interface graphique

La description du module `Graphics` de la librairie d'OCaml est disponible à l'adresse suivante : `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Graphics.html`. Pour utiliser ce module en mode interactif, il est nécessaire de charger `graphics.cma` dans le toplevel OCaml. On utilise pour cela :

```
#load "graphics.cma";; (* # initial nécessaire *)
open Graphics;;
```

La seconde instruction est facultative, mais elle évite d'avoir à écrire le préfixe `Graphics` avant chaque nom de fonction du module : par exemple on peut écrire `move_to` au lieu de `Graphics.move_to`, etc. On peut ensuite ouvrir une fenêtre graphique, par exemple de 800 lignes et 500 colonnes, en écrivant :

```
open_graph(" 500x800");; (* espace initial nécessaire *)
```

Après ouverture, le point courant est 0,0 (le coin en bas à gauche).

Les fonctions suivantes mettent en oeuvre une interface graphique pour Puissance 4.

Exercice 3. Écrivez une fonction `draw_board: unit -> unit` qui ouvre une fenêtre graphique, dont vous choisirez les dimensions, et y dessine le plateau de jeu, dans son état initial (voir un exemple en Figure 4) :

- ouvrir une fenêtre graphique, définir le titre de la fenêtre (`set_window_title`), colorier la totalité de la fenêtre avec une couleur de son goût (`rgb`, `set_color`, `fill_rect`) ;
- définir une épaisseur des lignes (`set_line_width`) et dessiner le cadre du plateau (`draw_poly_line`)
- changer l'épaisseur des lignes et dessiner les colonnes (`moveto`, `lineto`) et leur numéro (`draw_string`) ; utiliser des boucles pour éviter de la répétition inutile de code.

Exercice 4. Écrivez une fonction `draw_symbol: int -> int -> player -> unit` qui prend l'indice d'une colonne, l'indice d'une case sur cette colonne, un joueur, et qui dessine dans le plateau un cercle de la couleur du joueur, à la bonne place (utilisez `fill_circle`).

Exercice 5. Écrivez une fonction `wait_next_move: player -> int`, qui :

(a) Invite `player` à jouer (utilisez `draw_string`)

(b) Attend qu'un clic de la souris ait lieu sur une des colonnes du plateau.

(c) Renvoie l'indice de la colonne en question.

La fonctionnalité décrite au point (b) s'obtient ainsi :

```
let etat = wait_next_event[Button_down] in
    let x = etat.mouse_x and y = etat.mouse_y in ...
```

En effet, `wait_next_event: event list -> status` est une fonction du module `Graphics` qui prend en argument une liste de valeurs de type `event` (voyez dans `Graphics` la déclaration de ce type), attend que l'un des évenements en question se produise, et quand il se produit renvoie dans le résultat de type `status` (voyez dans `Graphics`), un certain nombre d'informations, dont les coordonnées du point de la fenêtre graphique où le clic de la souris s'est produit (cette information est significative si `Button_down` est un élément de la liste d'événements passée en argument).

Il s'agit de calculer la colonne du plateau dans laquelle ce point se trouve, et de renvoyer son indice.

Exercice 6. Écrivez une fonction `end_of_game : outcome -> unit` qui gère la fin de partie en annonçant le vainqueur s'il y en a un, ou alors match nul.

Boucle principale

Exercice 7. Écrire une fonction `main : unit -> unit` qui permet à deux joueurs de jouer l'un contre l'autre. La fonction `main` déclare le tableau de jeu, fait appel aux fonctions qui gèrent l'interface graphique et implémente la boucle principale, à exécuter tant que le jeu n'est pas terminé, c'est à dire tant que `eval(board) = Open`.

Exercice 8. Modifiez votre fonction `end_of_game : outcome -> unit` de l'Exercice 6 pour en plus proposer de commencer une nouvelle partie ou de quitter le jeu. Pour cela, les fonctions `main` et `end_of_game` peuvent être rendues mutuellement récursives, de façon à implémenter l'option "Nouvelle partie" dans `end_of_game` par un simple appel de `main`.

Jouer contre le programme

Stratégies

On appelle *stratégie* une fonction qui prend en paramètres un plateau de jeu et le joueur dont c'est le tour et qui modifie le plateau en retournant la position (ligne et colonne) modifiée.

Une autre stratégie très simple consiste à toujours jouer dans la première (en partant de la gauche par exemple) colonne jouable.

Exercice 9. Écrivez une fonction `strategie_simple: board -> player -> int * int` qui implémente la stratégie décrite ci-dessus. Vous pourrez utiliser la fonction `play`.

Une autre stratégie très simple (mais un peu moins) est la *stratégie aléatoire* — elle choisit au hasard une colonne jouable.

Exercice 10. Écrivez une fonction `strategie_aleas: board -> player -> int * int` qui implémente la stratégie décrite ci-dessus. Vous pourrez vous baser sur votre implémentation de `strategie_simple` et utiliser `Random.int(6)`, pour choisir une colonne aléatoirement [2].

Exercice 11. Modifier le programme pour donner au début du jeu le choix entre les options `UN JOUEUR` (contre l'ordinateur) et `DEUX JOUEURS`. Si cette dernière option est choisie, le jeu se déroule comme précédemment, sinon le joueur rouge, humain, joue contre le programme, qui joue selon la stratégie aléatoire. Le choix de l'option peut être proposé dans `draw_bord`, qui dans ce cas va renvoyer un entier (1 ou 2), ou dans une nouvelle fonction spécialisée de l'interface graphique. La fonction `main` teste cet entier et implémente les deux modalités de jeux dans un `if-then-else`.

Quelques heuristiques simples

Il y a des situations où jouer aléatoirement n'est clairement pas le meilleur choix. Par exemple s'il y a un coup immédiatement gagnant, il vaut mieux le jouer tout de suite. Si le joueur courant n'a pas de coup gagnant, mais qu'il y aurait un coup gagnant pour son adversaire, il vaut mieux qu'il prenne cette position avec un de ses propres pions.

Exercice 12. Écrivez une fonction `strategie_semi_aleas: board -> player -> int * int` qui se comporte aléatoirement sauf dans les cas décrits ci-dessus.

En fait, il existe une stratégie toujours gagnante pour le premier joueur [3]. Il existe aussi un algorithme pour déterminer (dans un état quelconque du jeu) s'il existe une stratégie qui va gagner (et qui dans ce cas trouve cette stratégie). Toutefois dans cette section nous allons nous contenter d'implémenter quelques règles (heuristiques) qui permettent de jouer assez bien.

Exercice 13. Si le joueur courant repère une ligne où son adversaire a déjà deux pions adjacents avec deux cases libres voisines, il est en danger. Il peut alors anticiper et jouer sur l'une de ses cases pour empêcher son adversaire de gagner. Écrivez une fonction `rule_base_inverse: board -> player -> int * int` qui, si cela est possible, joue un coup selon cette règle.

Exercice 14. Si le joueur courant repère une colonne où son adversaire a déjà deux pions l'un au dessus de l'autre avec le haut de la colonne libre, il peut insérer un de ses pions sur cette colonne pour empêcher son adversaire de gagner. Écrivez une fonction `rule_vertical: board -> player -> int * int` qui, si cela est possible, joue un coup selon cette règle.

Vous pouvez refléchir à des stratégies plus raffinées, et les implémenter si vous en avez le temps.

2. Attention aux bornes sur l'entier retourné par `Random.int`.
3. Voir par exemple la thèse de master de V. Allis (1988). Note : il avait plus de 7 ans en 1988.