TP n°7

Boucles, Graphismes et Événements

Le but de ce TP est d'implémenter le jeu de plateau *Puissance 4*[1]. Vous pouvez y utiliser les traits impératifs de OCaml (références, tableaux, boucles). Votre programme devra permettre à deux joueurs humains de jouer entre eux, à l'aide d'une interface graphique.

1 Règles du jeu

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

Rouge joue en premier. Les joueurs jouent à tour de rôle. Le joueur dont c'est le tour choisit une colonne qui n'est pas encore pleine, et y insère un de ses pions. Le pion tombe jusqu'à ce qu'il se retrouve bloqué par un autre pion ou par le bas de la colonne (Figure 2).

La partie se termine dès que l'un des deux joueurs a gagné (il a aligné quatre pions de sa couleur, Figure 3), ou lorsque le plateau est plein – dans ce cas, la partie est nulle.

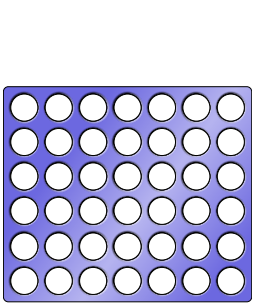

Figure 1 – Plateau vide

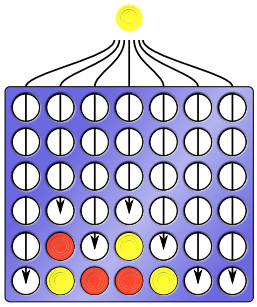

Figure 2 – Coups possibles

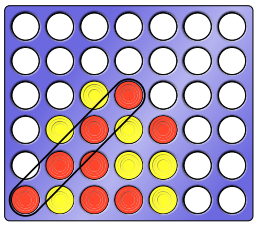

Figure 3 – Fin de partie

2 Fonctions de base pour la gestion du plateau

Vous aurez besoin des types suivants :

```
type player  = Red | Yellow;;
type cell    = Empty | Pawn of player;;
type outcome = Draw | Open | Win of player;;
type board   = cell array array;;
```

Le plateau de jeu sera représenté à l'aide du type `board`, comme une matrice d'éléments de type `cell` de taille 6×7. Le plateau initial pourra être obtenu à l'aide de la fonction `Array.make_matrix`.

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.

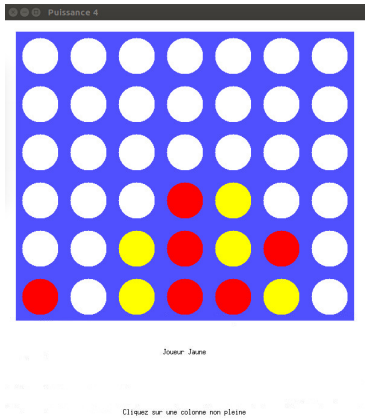

Figure 4 – Exemple d'affichage du plateau en cours de jeu

Exercice 1. Écrire une fonction `val play : board -> player -> int -> int`. Si `b` est la matrice représentant le plateau de jeu, `p` une valeur représentant l'un des joueurs et `c` le numéro d'une colonne non encore pleine choisie par ce joueur, l'appel `play b p c` doit : modifier `b` en jouant le coup correspondant ; renvoyer le numéro de la ligne modifiée dans `b`.

Noter que cette fonction suppose que `c` est toujours non pleine - cette condition sera vérifiée avant l'appel dans une autre partie du programme.

Exercice 2. Écrire une fonction `val eval : board -> outcome` examinant le contenu d'un plateau, et renvoyant l'état du jeu (`Draw` pour un match nul, `Win p` si `p` est gagnant, `Open` sinon).

Dans un premier temps, vous limiterez le traitement effectué par cette fonction à la vérification du fait que le plateau est non encore plein (`Open`/`Draw`), puis compléterez son écriture une fois l'interface graphique terminée.

3 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, en écrivant :

```
#load "graphics.cma";; (* diese initial necessaire *)
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(" 600x700");; (* espace initial dans la chaine necessaire *)
```

Après ouverture, le point courant est (0,0), en bas à gauche. L'approche présentée ic est celle suivie dans la solution mise en oeuvre dans l'exécutable `puissance4`, disponible sur Moodle. Vous pouvez bien sûr utiliser une approche différente.

3.1 Fonctions d'affichage

Exercice 3. Écrire une fonction `draw_empty_board: unit -> unit`, permettant d'afficher le plateau vide dans la fenêtre graphique, en suposant celle-ci déjà ouverte.

Exercice 4. Écrire une fonction `draw_symbol: int -> int -> player -> unit`. Un appel de la forme `draw_symbol i j p` doit dessiner dans le plateau un cercle de la couleur du joueur, à la case (`i`,`j`) – utilisez `fill_circle`.

3.2 Attente du coup d'un joueur

Exercice 5. Écrire une fonction `next_move: board -> player -> int`. A chaque appel de la forme `next_move b p`, cette fonction doit :

(a) inviter dans la fenêtre graphique le joueur `p` à jouer (utilisez `draw_string`),

(b) attendre qu'un clic de la souris ait lieu sur une des colonnes non encore pleines de `b`,

(c) renvoyer l'indice de la colonne choisie.

Le point (b) peut s'implémenter à l'aide de la fonction `wait_next_event` du module `Graphics`. Il suffit d'écrire une définition locale de la forme

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

Lorsqu'une expression de cette forme est évaluée, le programme suspend son exécution, et attend qu'un événement de la forme spécifiée (`Button_down`) se produise dans la fenêtre. Dans l'expression qui suit ce `let ... in`, les noms `x` et `y` désigneront les coordonnées du clic. Il est facile de déduire de `x` la colonne choisie, et de vérifier qu'elle n'est pas déjà pleine dans `b`.

Cette vérification peut être faite dans une boucle `while` qui durera tant que la colonne choisie n'est pas une colonne non encore pleine (pour mémoriser son numéro dans la boucle, servez-vous d'une référence déclarée avant celle-ci, et initialisée par exemple à `-1`).

3.3 Boucle principale

Ajouter à votre programme la déclaration de type suivante :

```
type state   = Play | Choose | Quit;;
```

Exercice 6. Écrire une fonction `end_of_game : outcome -> state`. Cette fonction suppose que son argument n'est pas `Open`, donc qu'il correspond à une fin de partie. Elle doit afficher le résultat de la partie, puis demander aux joueurs ce qu'ils souhaitent faire : rejouer, ou terminer le jeu. Dans le premier cas, elle renverra `Play`, et dans le second, `Quit`.

Exercice 7. Écrire enfin la fonction principale du programme, `main : unit -> unit`. Cette fonction utilisera deux références :

— La première sera appelée *l'état courant du programme* ; son contenu sera mis à jour après chaque appel de `end_of_game`, avec la valeur renvoyée par cette fonction.

— La seconde sera appelée *l'état courant du plateau* ; son contenu sera mis à jour après chaque coup joué, avec la valeur renvoyée par `eval`.

La fonction `main` devra implémenter le traitement suivant :

1. Ouvrir la fenêtre graphique.
2. Initialiser à `Play` l'état courant du programme.
3. Tant que l'état courant du programme n'est pas `Quit`, faire :
   (a) Déclarer un plateau vide, afficher un plateau vide.
   (b) Initialiser à `Open` l'état courant du plateau.
   (c) Tant que l'état courant du plateau est `Open`, jouer.
   (d) Appeler `end_of_game`.
4. Fermer la fenêtre graphique.

4 Extension : jouer contre le programme

Si vous avez été suffisamment rapide, vous pouvez tenter de traiter cette dernière partie – elle vous serait demandée si ce TD était un projet. Modifier la déclaration de `state`, en lui ajoutant deux constructeurs correspondant à deux nouveaux états du programme : le jeu à deux joueurs, et le jeu contre l'ordinateur :

```
type state =  Play | Two_players | One_player | Choose | Quit;;
```

Il faudra modifier votre fonction `main` pour demander à l'utilisateur, avant l'étape (3.a) s'il souhaite jouer à deux ou seul. On peut convenir du fait que dans ce dernier cas, l'ordinateur joue avec les pions jaunes.

Dans la partie (3.c) de la boucle principale, lorsque c'est à Jaune de jouer, son coup sera choisi non pas à l'aide de `next_move`, mais en appelant une fonction de *stratégie* – nous appelerons ainsi toute fonction prenant en arguments un plateau de jeu non plein, le joueur dont c'est le tour, et renvoyant un numéro de colonne jouable, *ie.* toute fonction de type `board -> player -> int`.

4.1 Stratégies élémentaires

Exercice 8. Une stratégie très simple consiste par exemple à toujours jouer dans la première colonne non encore vide en partant de la gauche. Écrire une fonction `strategie_simple` implémentant cette stratégie.

Exercice 9. Une autre stratégie très simple est la *stratégie aléatoire* — elle choisit au hasard une colonne jouable. Écrire une fonction `strategie_aleas` implémentant cette statégie.

Exercice 10. 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 s'il y a un coup gagnant pour son adversaire, il vaut mieux contrer ce coup si cela est possible. Écrire une fonction `strategie_semi_aleas` implémentant cette stratégie.

4.2 Stratégies améliorées

En fait, on sait qu'il existe une stratégie toujours gagnante pour le premier joueur[2]. Il existe aussi un algorithme pour déterminer (dans un état quelconque du jeu) s'il existe une stratégie gangante (et qui dans ce cas la calcule). Toutefois dans cette section nous allons nous contenter d'implémenter quelques règles (heuristiques) qui permettent au programme de jouer assez bien.

Exercice 11. 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. Écrire une fonction `rule_base_inverse` qui, si cela est possible, joue un coup selon cette règle – et sinon joue semi-aéatoirement.

Exercice 12. 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. Écrire une fonction `rule_vertical` qui, si cela est possible, joue un coup selon cette règle, et sinon applique la règle précédente.

2. Voir par exemple la thèse de master de V. Allis (1988). Note : il avait plus de 7 ans en 1988.