- Introduction à l'algorithmique et au langage C -

Programmation ludique

TP n°6

1re année ESIEA - Semestre 1

L. Beaudoin & R. Erra & A. Gademer & L. Avanthey

2016 - 2017

Avant propos

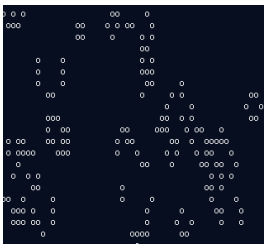
Pour clore ce premier semestre et mettre en pratique tout ce que nous avons vu, rien de mieux que de quitter les exercices pour coder un « vrai programme » de a à z. Pour cela nous vous proposons un TP ludique : votre travail consistera à coder un petit jeu dans le terminal. C'est l'occasion d'apprendre en vous amusant et de consolider vos acquis !

Quatre pour le prix d'un

Ce TP est constitué d'un tronc commun et de quatre « parfums » différents : Jeu de la Vie, Puissance 4, Proximity et Tron. Pour avoir la note maximale vous devrez réaliser l'ensemble du tronc commun et UN des quatre jeux au choix. Le sujet décrit l'ensemble des étapes à réaliser vous ne devez donc lire que les sections qui concernent votre jeux (avec bordure) et les sections qui concernent tous les jeux (sans bordure).

1 Les différents sujets

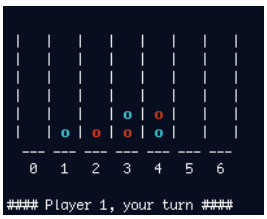
1.1 Jeu de la Vie



Le Jeu de la Vie, inventé par John H. Conway, n'est pas un jeu à proprement parler. C'est un automate cellulaire qui permet de simuler la vie et la mort d'une colonie de cellules selon un ensemble de règles. Ces dernières, bien que très simples, permettent de voir apparaître des schémas assez complexes, comme des oscillateurs, des planeurs, des canons et beaucoup d'autres formes dignes d'intérêt.

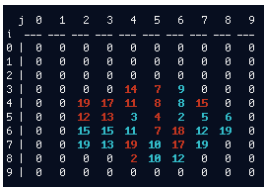
Pour aller plus loin : www2.lifl.fr/~delahaye/pls/2009/180.pdf.

1.2 Puissance 4



Vous connaissez tous le puissance 4, cette petite grille dans laquelle deux joueurs s'affrontent pour aligner au moins quatre de leurs pions à l'horizontal, à la verticale ou sur les diagonales et ainsi remporter la partie. Les joueurs jouent chacun à leur tour, jusqu'à ce que l'un deux gagne ou que la grille soit totalement remplie (match nul).

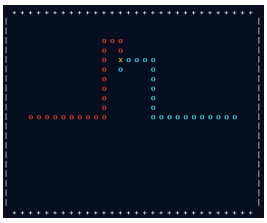
1.3 Proximity



Proximity est l'une des nombreuses variantes des jeux de conquête. Les deux joueurs piochent tour à tour des unités qu'ils placent sur le terrain. Chaque unité possède un numéro qui correspond à sa force. Lorsque le joueur la pose, les forces alliées à proximité s'en trouvent renforcées, et dans le même rayon, les forces ennemies plus faibles sont vaincues et deviennent alliées. Le but est de conquérir et de contrôler le plus de terrain possible.

Pour se faire une idée : http://www.koreus.com/jeu/proximity.html.

1.4 Tron



Ce jeu reprend le principe de la célèbre séquence de l'arène du film « Tron » de Walt Disney Pictures. Les motos-lumières s'affrontent en tournant à angle-droit et laissent derrière elles des traînées lumineuses persistantes et mortelles. Dans ce jeu, deux joueurs contrôlent chacun une moto-lumière. Le but est de survivre le plus longtemps possible sans percuter un rayon, les murs de l'arène ou la moto adverse.

Pour comprendre la dynamique de jeu : http://fltron.com/.

Pourquoi ne puis-je pas choisir un autre jeu ?

Tout simplement parce que nous avons calibré ces quatre jeux spécialement pour le cours. Ils ne font appel qu'aux notions que nous avons vues, ils peuvent être validés par le correcteur en ligne, ils sont de difficulté équivalente et ils sont très proches les uns des autres. Rien ne vous empêche par la suite, quand vous aurez terminé ce TP, de vous lancer sur votre temps libre pour coder d'autres jeux (ceux présentés dans ce sujet ou d'autres encore), c'est très formateur !

2 Les modalités

Ce TP sera noté. Vous pouvez le réaliser soit tout seul, soit en binôme (aucune autre configuration n'est possible), sachant que ce choix ne peut être changé au cours du TP. Voici les critères qui seront pris en compte pour la notation du TP :

- Exercices soumis sur le correcteur automatique. Notez que les exercices soumis mais non validés (IR, TE, MV, etc.) seront tout de même pris en compte (pas de notation binaire, mais graduée).
- Les commentaires dans le code.
- Une éventuelle évaluation orale avec démonstration.

CE & W

Nous vous rappelons qu'un code qui ne compile pas vaudra 0/20 et qu'un code contenant des avertissements (warnings) impliquera une note divisée par deux (moitié des points).

Avertissement plagiat

Nous vous rappelons que le travail doit être strictement personnel, le TP étant noté, les règles anti-plagiat s'appliquent. Tout travail rendu par un groupe (ou monôme) ressemblant trop à celui d'un autre groupe (ou monôme) sera sanctionné.

3 S'attaquer au problème

N'oubliez pas que l'art de l'algorithmique consiste à décomposer un problème compliqué en sous-problèmes plus simples. C'est le moment d'appliquer tout cela. Notre énoncé de base est de coder l'un des quatre jeux, commençons donc par découper ce problème complexe en trois parties principales :

— Déclaration des outils nécessaires au jeu
— Le jeu
— La fin du jeu

4 Première partie : déclaration des outils de base

Nous allons commencer par créer le fichier, déclarer notre plateau de jeu et les outils de bases associés.

EXERCICE 1

 Créez un nouveau fichier et écrivez-y le code C minimal.

EXERCICE 2

 Déclarez deux constantes en `#define`[1], NBLIN et NBCOL (majuscules **obligatoires**), qui détermineront la taille de notre plateau de jeu. Pour le moment, vous les mettrez à 10 toutes les deux (nous ajusterons la taille du terrain par la suite). Attention, nous n'utiliserons ces constantes que dans le `main`, jamais à l'intérieur des autres fonctions.

EXERCICE 3

 Dans le `main`, déclarez un tableau 2D d'entiers appelé `board`. Vous utiliserez les deux constantes précédentes pour fixer les dimensions du tableau. C'est notre plateau de jeu.

EXERCICE 4

 Écrivez la fonction `initArray` qui ne retourne rien et prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `iArray`, ainsi qu'une valeur entière `value`. Vous vous inspirerez de celle que vous avez écrite dans le TD « *Tableaux multidimensionnels* ». Elle doit initialiser tous les éléments du tableau avec la valeur passée en paramètre. Complétez le `main` avec l'appel à cette fonction pour laquelle vous demanderez une initialisation à 0.

EXERCICE 5

 Copiez-collez la fonction `void showArray(int nbLin, int nbCol, int iArray[nbLin][nbCol])` que vous avez écrite dans le TD « *Tableaux multidimensionnels* ». Cette dernière nous servira à afficher le contenu réel de notre tableau le temps de programmer le jeu (fonction qui nous aidera à trouver les erreurs). Complétez le `main` avec l'appel à cette fonction. Servez-vous en pour vérifier que votre première fonction a bien marché. Vous retirerez cet appel le cas échéant.

EXERCICE 6

 Soumettez votre code au correcteur automatique dans la partie correspondante. Si tout est validé, vous pourrez retirer pour la suite l'appel à la fonction `showArray` dans le `main`.

1. Reportez-vous au TD « *Hello World* » si vous ne savez plus comment faire.

5 Deuxième partie : le jeu

À nouveau nous allons devoir faire appel à notre savoir algorithmique. Le problème est complexe, nous devons le séparer en sous-problèmes plus simples. Nous allons donc décomposer cette partie en trois phases : la phase préparatoire, le déroulement de la partie et la fin de la partie.

EXERCICE 7

 Commencez par déclarer la fonction qui va lancer le jeu : `runGame` qui retourne un type entier et prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board`. C'est dans cette dernière que nous allons coder les trois phases énoncées ci-dessus. Nous nous contenterons de lui faire retourner 0 pour le moment. Appelons-la dans le `main`, juste après l'initialisation du plateau de jeu.

5.1 Phase 1 : Phase préparatoire

Certains jeux nécessitent de placer des éléments de base sur le plateau avant que les joueurs ne commencent la partie. C'est le cas de deux de nos jeux : le Jeu de la Vie et Tron.

Jeu de la Vie

Nous définirons une cellule morte comme étant une case du tableau égale à 0, et une cellule vivante une case égale à 1. Ayant initialisé notre tableau à 0 dans le `main`, nos cellules sont pour le moment toutes mortes. Nous avons donc besoin de placer quelques cellules vivantes sur le plateau avant de commencer le déroulement du jeu (ce que nous appellerons la génération initiale). Nous choisirons pour ce TP de les placer de manière aléatoire.

EXERCICE 8 (*Jeu de la vie*)

 Écrivez la fonction `seedRandomCells` qui ne retourne rien et qui prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board`. Cette fonction doit parcourir chaque case du tableau 2D pour y placer aléatoirement une cellule vivante (case égale à 1). Nous emploierons pour cela la fonction `rand()` qui génère une série pseudo-aléatoire [a]. La création d'une nouvelle cellule se fera selon une probabilité de 1/5, soit quand le nombre aléatoire modulo 5 (`rand()%5`) est égal à 0. Écrivez l'appel de cette fonction au début de la fonction `runGame`.

a. Nous avons déjà eu l'occasion de la rencontrer lors du TP3 « *Boucles & Embranchements* », elle est décrite dans la bibliothèque `stdlib.h`.

Tron

Nous avons besoin de sauvegarder trois informations par joueur : la dernière position du joueur (indices colonne-ligne) et sa direction (est-ce qu'il allait vers la droite, vers la gauche, vers le haut ou vers le bas). Au début de la partie, nous placerons les motos au milieu de l'arène, face à face.

Comment sauvegarder la position et la direction

Nous utiliserons pour cela un tableau unidimensionnel dont les deux premières cases correspondront respectivement à l'indice de la ligne et à l'indice de la colonne de la position, et dont la troisième case correspondra à la direction (identifiée par un numéro : 1 (gauche), 2 (haut), 3 (droite) et 4 (bas)). Nous l'appellerons le vecteur du joueur.

Tron

EXERCICE 9 (*Tron*)

 Ajoutez la fonction `initVectPlayer` qui ne retourne rien et qui prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board`, ainsi que le vecteur d'un joueur `vectPlayer` et le numéro du joueur associé `numPlayer`.

EXERCICE 10 (*Tron*)

 À l'intérieur de la fonction `initVectPlayer` :

- Vérifiez si le terrain est suffisamment large (au moins 10 colonnes), sinon quittez le programme (avec `exit(-1)`).
- Calculez l'indice de la ligne du milieu (nombre de ligne divisé par deux) et l'affecterez au vecteur du joueur.
- Calculez l'indice de la colonne : troisième colonne en partant du bord gauche pour le joueur 1 et en partant du bord droit pour le joueur 2. Vous l'affecterez au vecteur du joueur.
- Initialisez la direction du vecteur du joueur en fonction du numéro du joueur pour qu'il se dirige vers le centre de l'écran.
- Marquez la position sur le terrain (la trace de chaque joueur sera identifiée par leur numéro, 1 ou 2).

EXERCICE 11 (*Tron*)

 Écrivez l'appel de cette fonction au début de la fonction `runGame` pour chacun des deux joueurs. Vous déclarerez au préalable les deux tableaux unidimensionnels (de 3 cases) qui représentent les vecteurs des joueurs.

Toujours avant de commencer le déroulement du jeu, nous devons afficher l'état initial du plateau de jeu pour le montrer aux utilisateurs. Ceci est valable quel que soit le jeu.

EXERCICE 12

 Vous appellerez donc votre fonction `showArray` dans la fonction `runGame`. Par la suite nous la remplacerons par une fonction « customisée » qui nous affichera une représentation du plateau de jeu proche de l'aspect physique, plutôt que le contenu réel du tableau. Compilez et exécutez pour vérifier que votre état initial est correct.

Nous rajouterons par la suite quelques éléments à cette phase préparatoire, au fur et à mesure que nous en rencontrerons le besoin.

5.2 Phase 2 : Déroulement de la partie

Le déroulement de la partie est lui-même un problème complexe. Les jeux que nous vous proposons sont des jeux au tour par tour, c'est-à-dire qu'un certain nombre d'actions vont se répéter dans un même ordre tout au long de la partie. Par exemple au début d'un tour les joueurs (ou l'un d'eux) vont décider de déplacer ou poser quelque chose sur le plateau. L'état du plateau change en conséquence et le tour se termine. Le tour suivant commence et répète le même schéma : les joueurs décident d'une action, l'état du plateau change et le tour se termine, et ainsi de suite.

5.2.1 La boucle du jeu

Simuler le déroulement des tours

Pour simuler le déroulement de ces tours (et donc du jeu), nous utiliserons une boucle. Le type de boucle va changer en fonction du type de jeu. Certains jeux se terminent lorsque l'un des deux joueurs (ou les deux) perd ou gagne. Ces jeux sont événementiels et utiliseront donc une boucle de même type (`do while`). D'autres jeux se terminent au bout d'un nombre de tours donné (ie. connu à l'avance), ils utiliseront donc une boucle itérative (`for`).

Jeu de la Vie

Nous avons choisi de définir à l'avance le nombre de tours total à effectuer. Nous le stockerons dans une constante (`#define`) nommée NBSTEPS (majuscules obligatoires). Nous l'utiliserons uniquement dans la fonction `runGame` (et nul part ailleurs).

EXERCICE 13 (*Jeu de la vie*)

Ajoutez la constante NBSTEPS et fixez-la à 2 pour le moment (nous mettrons une autre valeur quand nous aurons terminé les tests).

EXERCICE 14 (*Jeu de la vie*)

Ajoutez la boucle itérative dans la fonction `runGame` en vous servant de la constante comme borne. Vous déclarerez votre compteur de tour au niveau des éléments de la phase préparatoire.

Proximity

Le nombre de tours total est limité par le nombre d'emplacements (cases) du terrain. Lorsque tous les emplacements sont remplis, le jeu se termine. Nous pouvons donc calculer le nombre de tours total en calculant le nombre de cases de notre plateau.

EXERCICE 15 (*Proximity*)

Dans la fonction `runGame`, au niveau des éléments de la phase préparatoire, calculez le nombre de tours total à l'aide des paramètres de la fonction et mémorisez-le dans une variable. Ajoutez ensuite la boucle itérative en vous servant de cette valeur comme borne. Vous déclarerez votre compteur de tour au niveau des éléments de la phase préparatoire.

Puissance 4

Le Puissance 4 est événementiel, c'est-à-dire qu'il se termine lorsque l'un des deux joueurs gagne. Il peut également y avoir un match nul lorsque la grille est entièrement pleine. C'est le deuxième cas de figure où le jeu est stoppé.

EXERCICE 16 (*Puissance 4*)

Dans la fonction `runGame`, au niveau des éléments de la phase préparatoire, déclarez une variable entière que vous utiliserez en variable booléenne (0 ou 1) et qui vous servira de drapeau (flag) pour savoir si le jeu doit s'arrêter ou non. Initialisez-la à vrai (1). Ajoutez la boucle événementielle (`do while`) et utilisez cette variable booléenne dans le test d'arrêt.

Puissance 4

EXERCICE 17 (*Puissance 4*)

Toujours dans la fonction `runGame`, au niveau des éléments de la phase préparatoire, déclarez une variable qui mémorise le nombre de coups possible (nombres de cases de la grille). À la fin de la boucle événementielle, décrémentez cette variable. Testez juste après s'il reste des coups à jouer, si ce n'est pas le cas, mettez le drapeau à faux (pour sortir de la boucle).

Tron

Tron est lui aussi événementiel, mais contrairement au Puissance 4, il se termine lorsque l'un des deux joueurs perds (ou les deux simultanément). Cela arrive en cas de collisions, soit avec les murs, soit avec les trainées lumineuses.

EXERCICE 18 (*Tron*)

Dans la fonction `runGame`, au niveau des éléments de la phase préparatoire, déclarez une variable booléenne (0 ou 1) qui vous servira de drapeau (flag) pour savoir si le jeu doit s'arrêter ou non. Initialisez-la à vrai (1). Ajoutez la boucle événementielle (`do while`) et utilisez cette variable booléenne dans le test d'arrêt.

5.2.2 Gestion des joueurs

Parmi les jeux qui se jouent à deux, nous distinguons deux catégories : la première lorsque les joueurs jouent chacun à leur tour (un tour = un joueur), la seconde lorsque les deux joueurs jouent « simultanément » lors d'un tour. Dans le cas de la première catégorie, il est important de connaître le numéro de joueur qui effectue les actions sur le tour en cours.

Proximity

EXERCICE 19 (*Proximity*)

Dans la fonction `runGame`, déclarez au niveau des éléments de la phase préparatoire une variable qui permettra de mémoriser le numéro du joueur en train de jouer. À l'intérieur de la boucle itérative, vous calculerez ce numéro. C'est le joueur 1 qui démarre la partie, le joueur 2 prend le tour suivant.

Puissance 4

EXERCICE 20 (*Puissance 4*)

Dans la fonction `runGame`, déclarez au niveau des éléments de la phase préparatoire une variable qui permettra de mémoriser le numéro du joueur en train de jouer. À l'intérieur de la boucle événementielle, vous calculerez ce numéro. C'est le joueur 1 qui démarre la partie, le joueur 2 prend le tour suivant.

5.2.3 Action d'un tour

À l'intérieur de la boucle, nous nous occupons d'effectuer les actions relatives au tour en cours, nous utiliserons notamment pour cela une fonction dédiée : `runAStep`.

Jeu de la Vie

L'objectif d'un tour est de calculer la nouvelle génération de cellules en se basant sur les règles de vie et de mort. Pour cela nous allons devoir parcourir tout le tableau pour évaluer chaque cellule une par une et décider si elle doit vivre ou mourir.

Génération actuelle, génération nouvelle

Attention, il ne faut surtout pas mélanger les deux générations, nous ne devons pas modifier l'état de la génération courante, car nous nous en servons pour connaître la suivante ! Il nous faudra donc utiliser un tableau temporaire tant que la nouvelle génération n'a pas été entièrement calculée.

Les règles de vie et de mort peuvent se résumer ainsi :

- Une cellule reste en vie si elle possède deux ou trois voisines vivantes, sinon elle meurt d'isolement ou d'étouffement.
- Une cellule nait si elle possède exactement trois voisines vivantes.

EXERCICE 21 (*Jeu de la vie*)

Commencez par déclarer la fonction `runAStep` qui ne retourne rien et prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board`.

EXERCICE 22 (*Jeu de la vie*)

En vous aidant du travail que vous avez effectué lors du TD « *Tableaux multidimensionnels* » sur le parcours des 8 voisins, écrivez la fonction `getNumberOfNeighboursAlive` qui retourne un type entier et qui prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board`, ainsi que les indices `cellLin` et `cellCol` d'une cellule du tableau. Cette fonction doit calculer et retourner le nombre de cellules vivantes autour de la cellule indiquée.

EXERCICE 23 (*Jeu de la vie*)

Écrivez la fonction `isCellDeadOrAlive` qui retourne un type entier et qui prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board`, ainsi que les indices `cellLin` et `cellCol` d'une cellule du tableau. La fonction doit retourner 1 si la cellule d'indices (cellLin, cellCol) est vivante à la génération suivante ou 0 si elle est morte à la génération suivante. Pour déterminer cela, vous utiliserez votre fonction qui calcule le nombre de voisines vivantes d'une cellule et les règles de vie et de mort.

EXERCICE 24 (*Jeu de la vie*)

Écrivez la fonction `copyArray` qui ne retourne rien et qui prend en paramètres (dans l'ordre) un nombre de lignes `nbLin`, un nombre de colonnes `nbCol` et deux tableaux d'entiers 2D de mêmes dimensions `array1` et `array2`. La fonction doit copier le contenu du premier tableau dans le second.

EXERCICE 25 (*Jeu de la vie*)

Dans votre fonction `runAStep`, écrivez les boucles de parcours qui vous permettent d'accéder à chaque cellule du plateau. À l'intérieur, en utilisant les fonctions que vous venez d'écrire, enregistrez le nouvel état de chacune d'entre-elles dans un tableau temporaire. À la fin de la fonction, n'oubliez pas d'écraser le tableau de l'ancienne génération par celui de la nouvelle grâce à votre fonction de copie pour ainsi mettre à jour l'état du plateau.

Proximity

Dans un tour, le joueur doit piocher une nouvelle unité dont la force lui sera révélée et choisir où la placer sur le terrain en indiquant les coordonnées. Si dans le voisinage proche de l'endroit où elle est placée se trouvent des forces ennemies plus faibles, elles se font convertir. Si dans ce même voisinage se trouvent des forces alliées, elles gagnent un point de force.

Distinguer les deux joueurs sur le plateau

Nous avons besoin de sauvegarder les valeurs de force des unités déjà positionnées, mais aussi de distinguer à qui elles appartiennent. Pour cela, nous les différencieront en les notants de manière positive pour un joueur et négative pour l'autre. Pour faire les calculs, nous ne tiendrons évidemment pas compte de ce signe.

EXERCICE 26 (*Proximity*)

Commencez par déclarer la fonction `runAStep` qui ne retourne rien et prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board` ainsi que le numéro du joueur.

EXERCICE 27 (*Proximity*)

Écrivez la fonction `getNextUnitForce` qui retourne un type entier et ne prend rien en paramètre. Cette fonction doit retourner une valeur de force aléatoire comprise entre 1 et 20. Nous emploierons pour cela la fonction `rand()` qui génère une série pseudo-aléatoire [a]. Nous repasserons les nombres générés entre 1 et 20 de cette manière : `rand() % 19 + 1`.

EXERCICE 28 (*Proximity*)

Ajoutez la fonction `placeUnitForce` qui ne retourne rien et qui prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board` ainsi que la valeur de force d'une unité `unitForce` et un tableau unidimensionnel pour sauvegarder les coordonnées de l'unité `ufCoord`.

EXERCICE 29 (*Proximity*)

À l'intérieur de la fonction `placeUnitForce` :

- Demandez à l'utilisateur les indices de la case sur laquelle placer l'unité : `"Enter the coordinates i and j of the case you want to put your UF on: "`.
- Récupérez ces coordonnées à l'aide de la fonction `scanf` en vérifiant bien que les nombres entrés ne sont pas des caractères (sinon `exit(-1)`).
- Reposez la question tant que les coordonnées ne sont pas dans le tableau (`"Wrong coordinates (outside of the board), please re-try.\n"`) ou que la case indiquée est déjà occupée par une unité (`"\nCell allready full, please re-try.\n"`).
- Une fois les coordonnées validées, affectez la valeur de la force de l'unité à la case indiquée et sauvegarde les unités dans le tableau unidimensionnel `ufCoord` (la ligne d'abord, puis la colonne).

a. Nous avons déjà eu l'occasion de la rencontrer lors du TP3 « *Boucles & Embranchements* », elle est décrite dans la bibliothèque `stdlib.h`.

Proximity

EXERCICE 30 (*Proximity*)

Ajoutez la fonction `affectNeighborhood` qui ne retourne rien et prend en paramètre le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board` ainsi que les indices d'une case du plateau `groundLin` et `groundCol`.

EXERCICE 31 (*Proximity*)

À l'intérieur de la fonction `affectNeighborhood` :

- Vous commencerez par récupérer le numéro du joueur grâce au signe de la valeur contenue dans la case d'indices données (positif = joueur 1, négatif = joueur 2).
- Puis en vous aidant du travail que vous avez effectué lors du TD « *Tableaux multidimensionnels* » sur le parcours des 8 voisins, vous affecterez le voisinage de la cellule donnée :
  → Si un voisin est un ennemi plus faible, il devient un allié (changement de signe).
  → Si un voisin est un allié, il gagne un point de force (en valeur absolue, soit +1 pour le joueur 1, -1 pour joueur 2).

EXERCICE 32 (*Proximity*)

Dans la fonction `runAStep`, en vous servant des fonctions que vous venez d'écrire :

- Récupérez la force d'une nouvelle unité.
- Affichez la valeur de la force de l'unité : `"\n#### Player X ####\nYour next Unit Force is YY\n"`, où X est le numéro du joueur (1 ou 2) et YY la force de l'unité piochée (entre 1 et 20).
- Modifiez le signe de la force de l'unité en fonction du numéro du joueur pour l'étiqueter.
- Placez l'unité sur le plateau d'après les coordonnées indiquées par le joueur.
- Modifiez le voisinage de l'unité.

Puissance 4

Dans un tour, le joueur doit choisir une colonne et y placer son pion. Ce dernier tombe jusqu'à trouver le premier emplacement vide de la colonne. Si ce nouveau pion permet la formation d'un alignement de 4 pions identiques sur la verticale, l'horizontale ou les deux diagonales, alors le joueur a gagné et le jeu s'arrête.

Distinguer les deux joueurs sur le plateau

Pour ce jeu, les pions n'ont pas de valeur, nous pouvons donc leur donner directement le numéro du joueur auxquels ils appartiennent.

EXERCICE 33 (*Puissance 4*)

Commencez par déclarer la fonction `runAStep` qui retourne un type entier et prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board` ainsi que le numéro du joueur `numPlayer`.

EXERCICE 34 (*Puissance 4*)

Ajoutez la fonction `getColumnForPawn` qui retourne un type entier et qui prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board`.

EXERCICE 35 (*Puissance 4*)

À l'intérieur de la fonction `getColumnForPawn` :

- Demandez à l'utilisateur l'indice de la colonne sur laquelle placer l'unité : `"Enter the coordinate of the column you want to put your pawn on: "`.
- Récupérez la coordonnée à l'aide de la fonction `scanf` en vérifiant bien que le nombre entré n'est pas un caractère (sinon `exit(-1)`).
- Reposez la question tant que la coordonnée n'est pas dans le tableau (`"Wrong number of column (outside of the board), please re-try.\n"`) ou bien que la colonne indiquée est déjà pleine (`"\nThis column is full so you can't add any more pawn, please re-try.\n"`).
- Une fois la coordonnée validée, vous la retournez.

EXERCICE 36 (*Puissance 4*)

Écrivez la fonction `placePawn` qui retourne un type entier et prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board` ainsi que le numéro de la colonne `pawnCol` dans laquelle placer le pion et la valeur du pion `pawn` (1 ou 2 selon le joueur). Cette fonction recherche le premier emplacement libre de la colonne donnée en partant du bas, y place le pion et retourne le numéro de ligne correspondant.

EXERCICE 37 (*Puissance 4*)

Ajoutez la fonction `checkFourInLine` qui retourne un type entier et prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board` ainsi que les coordonnées d'un pion `pawnLin` et `pawnCol`.

EXERCICE 38 (*Puissance 4*)

Complétez la fonction `checkFourInLine` en vous inspirant du travail que vous avez effectué lors du TD « *Tableaux multidimensionnels* » sur le parcours des voisins :

- Comptabilisez le nombre de pions de même type et consécutifs de part et d'autre du pion selon la verticale, l'horizontale et les deux diagonales.
- Pour chacun des quatre cas, si ce nombre est égal ou supérieur à 4, vous retournerez 1 (vrai).
- Si aucun des cas ne comptabilise un alignement de 4 pions, retournez 0 (faux).

Puissance 4

EXERCICE 39 (*Puissance 4*)

Dans la fonction `runAStep`, en vous servant des fonctions que vous venez d'écrire :

- Affichez le message suivant : `"\n#### Player X, your turn ####\n"`, où X est le numéro du joueur qui doit jouer.
- Récupérez l'indice de la colonne.
- Placez le pion et récupérez l'indice de la ligne.
- Vérifiez qu'il n'y a pas 4 pions alignés sur la grille.
- Retournez 1 (vrai) s'il y a 4 pions alignés, ou 0 (faux) sinon

Tron

Dans un tour, nous devons récupérer les événements claviers des deux joueurs (s'il y en a), mettre à jour la direction des motos, évaluer les collisions et déplacer les motos. En cas de collision, le jeu s'arrête.

EXERCICE 40 (*Tron*)

Commencez par déclarer la fonction `runAStep` qui retourne un type entier et prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board` ainsi que les vecteurs des deux joueurs `vectPlayer1` et `vectPlayer2`.

Pour récupérez les événements clavier, nous allons utiliser une fonction qui fait appel aux bibliothèques suivantes : `termios.h`, `fcntl.h` et `unistd.h` que vous rajouterez au début de votre programme.

EXERCICE 41 (*Tron*)

Ajoutez à votre code les trois bibliothèques citées ci-dessus, ainsi que la fonction `key_pressed` décrite ci-dessous. Cette dernière vous permettra de récupérer la valeur d'une lettre entrée au clavier sans bloquer le terminal.

```c
/* Function that get the keyboard event */
char key_pressed() {

   struct termios oldterm, newterm;
   int oldfd;
   char c, result = 0;

   tcgetattr (STDIN_FILENO, &oldterm);
   newterm = oldterm;
   newterm.c_lflag &= ~(ICANON | ECHO);
   tcsetattr (STDIN_FILENO, TCSANOW, &newterm);
   oldfd = fcntl(STDIN_FILENO, F_GETFL, 0);
   fcntl (STDIN_FILENO, F_SETFL, oldfd | O_NONBLOCK);
   c = getchar();
   tcsetattr (STDIN_FILENO, TCSANOW, &oldterm);
   fcntl (STDIN_FILENO, F_SETFL, oldfd);
   if (c != EOF) {
      ungetc(c, stdin);
      result = getchar();
   }

   return result;

}
```

Tron

EXERCICE 42 (*Tron*)

Écrivez la fonction `catchNextEvents` qui ne retourne rien et qui prend en paramètres (dans l'ordre) les vecteurs des deux joueurs `vectPlayer1` et `vectPlayer2`. À l'intérieur :

- Appelez 10 fois la fonction `key_pressed` et enregistrez le résultat dans un tableau de caractères (cela nous permet de récupérer les ordres des deux joueurs, s'il y en a).
- Mettez à jour la direction des vecteurs des joueurs à partir du dernier ordre donné pour chacun des joueurs en tenant compte de l'ancienne direction (nous tournons à droite ou à gauche). Les ordres valides sont `'q'` (gauche) et `'d'` (droite) pour le premier joueur, `'k'` (gauche) et `'m'` (droite) pour le second joueur. S'il n'y a pas d'ordre de nouvel ordre, la direction n'est pas modifiée (la consigne est inchangée).

EXERCICE 43 (*Tron*)

Écrivez la fonction `calculateNextPosition` qui ne retourne rien et qui prend en paramètres (dans l'ordre) le vecteur d'un joueur `vectPlayer` et un tableau 1D pour enregistrer la nouvelle position (2 cases) `nextPosPlayer`. Cette fonction doit enregistrer dans le tableau de position passé en paramètre les indices de la nouvelle position de la moto à partir de l'information de direction et de position actuelle enregistrées dans le vecteur du joueur.

EXERCICE 44 (*Tron*)

Écrivez la fonction `moveVehicle` qui ne retourne rien et qui prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board`, ainsi que le numéro du joueur `numPlayer`, le vecteur du joueur `vectPlayer` et le tableau qui contient la nouvelle position `nextPosPlayer`. A l'intérieur, marquez la nouvelle position sur le plateau de jeu à l'aide du numéro du joueur et mettez à jour le vecteur du joueur avec cette nouvelle position.

EXERCICE 45 (*Tron*)

Écrivez la fonction `checkForWallCollision` qui retourne un type entier et qui prend en paramètres (dans l'ordre) le nombre de lignes `nbLin` et le nombre de colonnes `nbCol` du terrain, ainsi que le tableau 1D qui contient la nouvelle position `nextPosPlayer`. La fonction retourne 1 (vrai) s'il y a une collision avec les murs (la moto essaye de sortir du plateau de jeu, donc du tableau 2D), et 0 (faux) sinon.

EXERCICE 46 (*Tron*)

Écrivez la fonction `checkForBeamCollision` qui retourne un type entier et qui prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board`, ainsi que le tableau 1D qui contient la nouvelle position `nextPosPlayer`. La fonction retourne 1 (vrai) s'il y a une collision avec les traces (valeurs non nulles) laissées sur le plateau, et 0 (faux) sinon.

Tron

EXERCICE 47 (*Tron*)

Dans la fonction `runAStep`, en vous servant des fonctions que vous venez d'écrire :

- Récupérez les ordres au clavier et la nouvelle direction que doivent prendre les motos.
- Calculez la prochaine position des motos.
- Vérifiez qu'il n'y a pas de collision avec les murs pour les deux joueurs. S'il y a collision, vous marquerez la position courante sur le plateau avec la valeur -1.
- Vérifiez qu'il n'y a pas de collision avec les traces lumineuses pour les deux joueurs. S'il y a collision, vous marquerez la prochaine position sur le plateau (celle qui engendre la collision) avec la valeur -1.
- Vérifiez qu'il n'y a pas de collision entre les deux motos elle-même (elles veulent toutes les deux se déplacer sur la même case). S'il y a collision, vous marquerez la prochaine position de chacune des motos sur le plateau (celle qui engendre la collision) avec la valeur -1.
- Si la moto du joueur 1 n'enregistre aucune collision, déplacez-la sur la prochaine position. Idem pour la moto du joueur 2.
- En cas de non collision, la fonction retourne -1. En cas de collision, la fonction retourne 0 si les deux joueurs ont généré des collisions (match nul), ou bien le numéro du joueur gagnant (celui qui n'a pas généré de collision) si un seul des deux joueurs a généré une collision.

EXERCICE 48

Ajoutez l'appel de la fonction `runAStep` dans la boucle de la fonction `runGame`. Puis à la suite affichez le nouvel état du plateau (avec la fonction `showArray`).

5.2.4 Contrôles à la fin d'un tour

Avant de passer au tour suivant, il va nous falloir effectuer un certain nombre de contrôles.

Jeu de la Vie

À la fin de chaque tour, nous allons contrôler le nombre de cellules vivantes.

EXERCICE 49 (*Jeu de la vie*)

Écrivez la fonction `getNumberOfLivingCells` qui retourne un type entier et prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board`. Cette dernière doit retourner le nombre de cellules vivantes sur le plateau.

EXERCICE 50 (*Jeu de la vie*)

Appelez votre fonction à la fin de la boucle itérative de la fonction `runGame`. Vous récupérerez le nombre de cellules vivantes dans un variable que vous aurez préalablement déclarée à la suite des éléments de la phase préparatoire. Vous afficherez juste après le calcul la phrase suivante : `"\n\nIl y a N cellules vivantes.\n"` où N est le nombre de cellules vivantes à chaque tour.

Proximity

À la fin de chaque tour, nous allons calculer le nombre de territoires appartenant à chacun des joueurs.

EXERCICE 51 (*Proximity*)

Écrivez la fonction `getNumberOfTerritoriesForPlayer` qui retourne un type entier et prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board` ainsi que le numéro du joueur concerné `numPlayer`. Elle doit retourner le nombre de territoires (nombre de cases) appartenant au joueur indiqué.

EXERCICE 52 (*Proximity*)

Appelez votre fonction à la fin de la boucle itérative de la fonction `runGame`. Vous récupérerez le nombre de territoires pour chacun des joueurs dans des variables que vous aurez préalablement déclarées à la suite des éléments de la phase préparatoire. Vous afficherez juste après le calcul la phrase suivante : `"\nTerritories Player 1: XX - Territories Player 2: YY\n"` où XX est le nombre de territoires du premier joueur et YY le nombre de territoires du second joueur.

Puissance 4

Nous avons déjà ajouté le test qui nous permet de savoir s'il reste des emplacements vides dans la grille. Il nous faut ajouter juste avant cela le test qui permet d'arrêter le jeu s'il y a un gagnant.

EXERCICE 53 (*Puissance 4*)

Dans la fonction `runGame`, à la fin de la boucle événementielle, juste avant le test du nombre d'emplacements, testez la valeur de retour de la fonction `runAStep` que vous aurez préalablement enregistrée dans une variable déclarée au niveau des éléments de la phase préparatoire.
Si ce retour vous indique un gagnant, vous devez passer le drapeau booléen d'arrêt de jeu à faux (0) et sauvegarder le numéro du joueur en cours (le gagnant).

Tron

Comme pour le Puissance 4, il nous faut ajouter le test qui nous permet d'arrêter le jeu s'il y a un gagnant.

EXERCICE 54 (*Puissance 4*)

Dans la fonction `runGame`, à la fin de la boucle événementielle, testez la valeur de retour de la fonction `runAStep` que vous aurez préalablement enregistrée dans une variable déclarée au niveau des éléments de la phase préparatoire.
Si ce retour vous indique un gagnant, vous devez passer le drapeau booléen d'arrêt de jeu à faux (0) et sauvegarder le numéro du joueur en cours (le gagnant).

5.3 Phase 3 : Fin de partie

Dans cette phase, nous allons terminer le jeu en renvoyant le numéro du joueur gagnant, ou 0 dans le cas d'un match nul.

Proximity

Dans Proximity, le gagnant est le joueur qui possèdent le plus de territoires (indépendamment de la valeur de ses forces).

EXERCICE 55 (*Proximity*)

 À la fin de la fonction `runGame`, déterminez qui est le gagnant. Vous sauvegarderez le numéro du joueur gagnant (1 ou 2) dans une variable que vous aurez préalablement déclarée dans la phase préparatoire. S'il y a match nul, vous mettrez 0. C'est cette variable que vous retournerez à la fin de la fonction.

Puissance 4

EXERCICE 56 (*Puissance 4*)

 À la fin de la fonction `runGame`, retournez le numéro du gagnant que vous avez sauvegardé.

Tron

EXERCICE 57 (*Tron*)

 À la fin de la fonction `runGame`, retournez le numéro du gagnant que vous avez sauvegardé.

6 Affichage du plateau de jeu

Pour l'instant, nous nous sommes contentés d'afficher le contenu réel du tableau afin de nous concentrer sur le moteur de jeu. Maintenant que ce dernier est bien avancé, nous allons un peu améliorer notre affichage pour qu'il soit plus proche de la réalité.

6.1 Simuler l'animation : effacer l'écran

EXERCICE 58

 Ajoutez à votre code la fonction `clearScreen` décrite ci-dessous. Elle vous permettra d'effacer l'écran du terminal et de replacer le curseur en haut. Cela donnera un effet d'animation à votre affichage.

```c
/* Function that clear the terminal's screen */
void clearScreen() {

   /* Animation - clear the previous display */
   printf("%c[2J", 0x1B);
   /* Animation - Move the cursor top-left */
   printf("%c[%d;%dH", 0x1B, 1, 1);

}
```

EXERCICE 59

 Ajoutez la fonction `showBoard` qui ne retourne rien et prend en paramètres (dans l'ordre) le nombre de lignes `nbLin`, le nombre de colonnes `nbCol` et le tableau d'entiers 2D associé `board`. Recopiez-y pour le moment le contenu de la fonction `showArray`.

EXERCICE 60


Ajoutez l'appel à la fonction qui efface l'écran au début de la fonction `showBoard`.

6.2 Ajuster la taille du plateau de jeu

Vous allez maintenant changer la valeur des constantes, pour pouvoir profiter pleinement du jeu et vérifier que tout marche bien à plus grande échelle. Les valeurs indiquées seront celles à fournir pour les soumissions au correcteur automatique. Vous pourrez les changer à votre gré pour vos tests personnels.

Jeu de la Vie

EXERCICE 61 (*Jeu de la vie*)


Changez la valeur des constantes comme suit :
- NBLIN à 15
- NBCOL à 50
- NBSTEPS à 5

Pour profiter du jeu, vous pourrez ajuster le nombre d'étapes pour vos propres tests à 50 par exemple.

Proximity

EXERCICE 62 (*Proximity*)


Changez la valeur des constantes comme suit :
- NBLIN à 3
- NBCOL à 3

Pour profiter du jeu, vous pourrez agrandir le terrain pour vos propres tests (à 10, 10 par exemple).

Puissance 4

EXERCICE 63 (*Puissance 4*)


Changez la valeur des constantes comme suit (taille classique de la grille) :
- NBLIN à 6
- NBCOL à 7

Tron

EXERCICE 64 (*Tron*)


Changez la valeur des constantes comme suit :
- NBLIN à 10
- NBCOL à 10

6.3 Améliorer le *gameplay*

Pour les deux jeux qui utilisent des séries pseudo-aléatoires (Jeu de la Vie et Proximity), il va falloir compléter votre code pour que le jeu soit plus réaliste et améliorer ainsi le *gameplay*. Pour que notre série pseudo-aléatoire ne soit pas la même à chaque exécution, nous allons lui donner une graine qui changera à chaque fois que le programme se lance.

Graine différente à chaque exécution

Le meilleur moyen est d'utiliser le temps comme graine : entre deux exécutions, le temps a évolué de quelques secondes et sa valeur lors de la seconde ne sera donc pas la même que dans la première.

EXERCICE 65 (*Jeu de la Vie & Proximity*)

Commencez par ajouter la bibliothèque `time.h`. Puis dans la fonction `main`, juste avant l'initialisation, ajoutez l'instruction suivante : `srand(time(NULL))`. Pour soumettre au correcteur automatique, vous mettrez en commentaire cette ligne, la fonction `srand` est interdite pour le rendu au correcteur automatique.

6.4 Afficher le plateau

Le contenu de la fonction d'affichage `showBoard` doit être traité au cas par cas, car il s'agit de récréer le plateau spécifique au jeu choisi.

Jeu de la Vie

Pour le Jeu de la Vie, nous n'avons pas besoin de délimitation de terrain et nous afficherons uniquement les cellules vivantes. Il nous faudra également ralentir l'affichage, sinon nous n'aurons pas le temps de voir ce que contiennent les générations.

EXERCICE 66 (*Jeu de la vie*)

Modifiez le code de la fonction `showBoard` pour ne plus afficher le contenu des cases mais un `'o'` quand il y a une cellule vivante et une espace `' '` sinon.

EXERCICE 67 (*Jeu de la vie*)

Vous ajouterez l'instruction `sleep(1)` ; après l'appel de la fonction qui efface l'écran pour ralentir le délai entre deux affichages. N'oubliez pas d'ajouter la bibliothèque associée à la fonction `usleep`, `unistd.h`.

Voici un exemple de l'affichage à obtenir :

Proximity

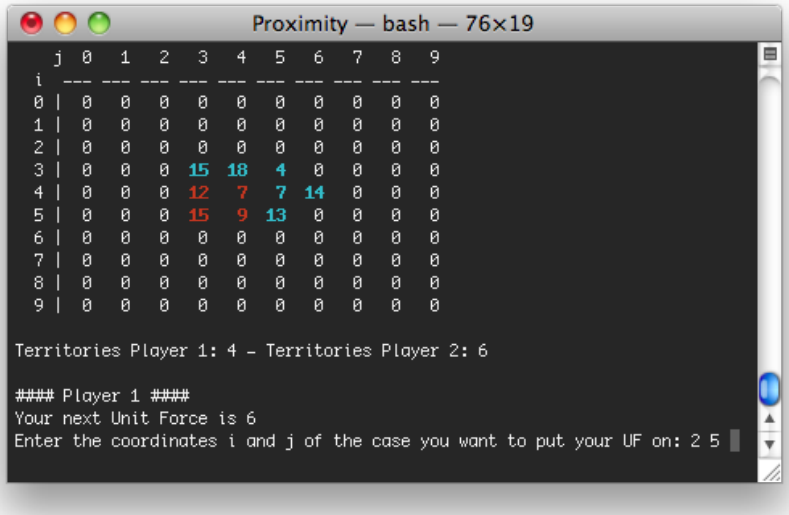
Pour Proximity, nous allons devoir indiquer les délimitations du plateau et les indices (lignes, colonnes) pour aider les joueurs à placer leurs pions. De plus, nous allons devoir différencier les unités de forces des deux joueurs visuellement. Nous ne pouvons pas utiliser le signe car ce ne serait pas très parlant pour l'utilisateur, nous préférerons donc utiliser la couleur.

EXERCICE 68 (*Proximity*)

Modifiez le code de la fonction `showBoard` pour :

- Aligner les colonnes. La valeur des unités de forces, des colonnes ou des lignes, peuvent être soit sur un chiffre, soit sur deux chiffres, or c'est très difficile à lire s'il n'y a pas d'alignement. Pour pallier à cela, vous allez modifiez le descripteur de type pour afficher ces valeurs systématiquement sur 3 caractères (la valeur est complétée par des espaces si nécessaire) : `"%3d "`.
- Distinguer les unités de forces selon les joueurs.
  - → Si la valeur de la case est nulle, alors il n'y a pas d'unité. L'affichage ne change pas.
  - → Si la valeur est strictement positive, alors il s'agit du premier joueur, attribuez lui la couleur rouge de cette manière : `"\033[31;01m%3d \033[00m"`.
  - → Si la valeur est strictement négative, alors il s'agit du second joueur, attribuez lui la couleur cyan de cette manière : `"\033[36;01m%3d \033[00m"`. N'oubliez pas d'afficher la valeur absolue et pas la valeur signée.
- Afficher la bordure du plateau et les indices (référez-vous à l'exemple ci-après comme modèle).

Voici un exemple de l'affichage à obtenir :



Puissance 4

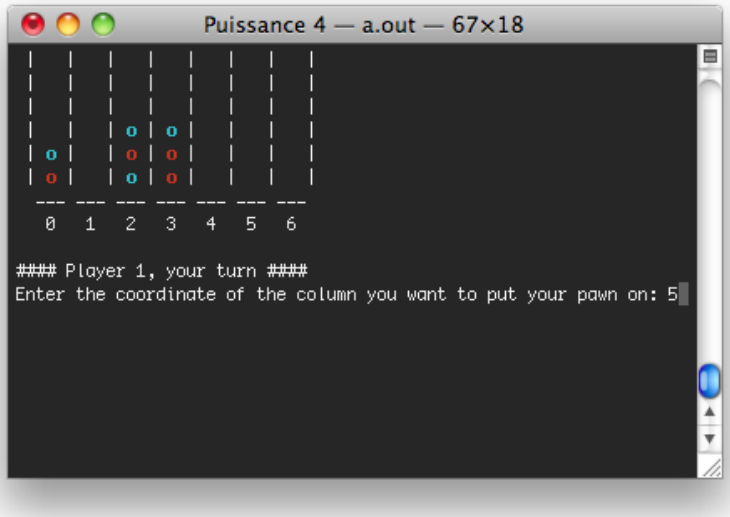
Pour le Puissance 4, nous allons devoir indiquer les délimitations de la grille ainsi que les indices des colonnes pour aider les joueurs à placer leurs pions. Comme pour Proximity, nous allons devoir différencier les pions des deux joueurs visuellement et de la même manière nous le ferons par la couleur.

EXERCICE 69 (*Puissance 4*)

Modifiez le code de la fonction `showBoard` pour :

- Distinguer les unités de forces selon les joueurs. Les pions du joueur 1 seront en rouge et ceux du joueur 2 en cyan.
  - → Si la valeur de la case est nulle, alors il n'y a pas de pion. Vous afficherez une espace.
  - → Si la valeur est égale à 1, alors il s'agit du premier joueur, affichez un rond rouge de cette manière : `"\033[31;01m o\033[00m""`.
  - → Si la valeur est égale à 2, alors il s'agit du second joueur, affichez un rond cyan de cette manière : `"\033[36;01m o\033[00m"`.
- Afficher la bordure du plateau et les indices des colonnes (référez-vous à l'exemple ci-après comme modèle).

Voici un exemple de l'affichage à obtenir :



Tron

Pour Tron, nous avons également besoin des délimitations de terrain (pour éviter de rentrer dans les murs) et nous devons afficher les traces lumineuses des motos que nous distinguerons comme les jeux précédents par la couleur. Comme pour le Jeu de la Vie, il nous faudra ralentir l'affichage, sous peine d'être déjà dans un mur le temps d'essayer de voir où se trouvent les motos.

EXERCICE 70 (*Tron*)

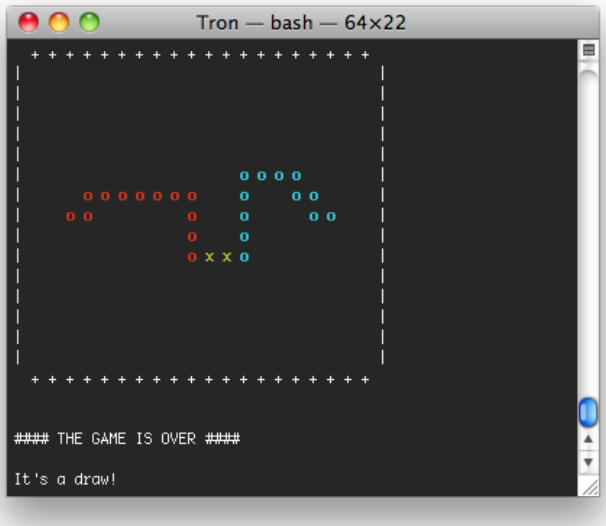
Vous ajouterez l'instruction `usleep(100000);` après l'appel de la fonction qui efface l'écran pour ralentir le délai entre deux affichages. Normalement, vous avez déjà ajouté la bibliothèque associée à la fonction `sleep`, `unistd.h`.

EXERCICE 71 (*Tron*)

Modifiez le code de la fonction `showBoard` pour :

- Distinguer les traces lumineuses selon les joueurs, ainsi que les collisions. Les traces du joueur 1 seront en rouge et celles du joueur 2 en cyan, quant aux collisions elles seront en jaune.
  - → Si la valeur de la case est nulle, alors il n'y a pas de trace ni de collision. Vous afficherez une espace.
  - → Si la valeur est égale à 1, alors il s'agit du premier joueur, affichez un rond rouge de cette manière : `"\033[31;01mo \033[00m""`.
  - → Si la valeur est égale à 2, alors il s'agit du second joueur, affichez un rond cyan de cette manière : `"\033[36;01mo \033[00m"`.
  - → Si la valeur est égale à -1 alors il s'agit d'une collision, affichez une croix orange de cette manière : `"\033[33;01mx \033[00m"`.
- Afficher la bordure du plateau (référez-vous à l'exemple ci-après comme modèle).

Voici un exemple de l'affichage à obtenir :



7 Troisième partie : la fin du jeu

La fin du jeu consiste à terminer le jeu en annonçant le résultat. Elle concerne tous les jeux ayant un mode à deux joueurs (tous, donc, sauf le Jeu de la Vie)

EXERCICE 72 (*Proximity - Puissance 4 - Tron*)

Écrivez la fonction `endOfGame` qui ne retourne rien et qui prend en paramètre le numéro du gagnant `numWinner`. La fonction doit afficher l'en-tête suivant : `"\n\n#### THE GAME IS OVER ####\n"` suivi du résultat du jeu : `"\nIt's a draw!\n"` si le match est nul (0) ou bien `"\nAnd the winner is Player X!\n"`, où X est le numéro du joueur gagnant (1 ou 2).

EXERCICE 73 (*Proximity - Puissance 4 - Tron*)

Appelez cette fonction à la fin du `main`. Vous aurez au préalable récupéré dans une variable l'information sur le résultat du match grâce au retour de la fonction `runGame`.