Rapport du Projet 24°
Jeux : Prairie

Heddadji Dianna
Ngoie-Mbayi Jonathan

Decembre 2018

Introduction

Dans le cadre de notre projet, nous avons choisi de réaliser le projet numéro 24, Prairie. Ce projet consiste à programmer en C un jeux sur une table de dimension 11 X 7 cases. L'objectif du jeux est d'amener au moins l'un des buffles de l'autre côté du plateau pour marquer la victoire du camp de buffle,de capturer tous les buffles ou de les bloquer avec les chiens pour marquer la victoire pour l'autre camp qui est celui du chef indien.

Methode de travail et code

Nous avons choisi ce projet à cause de la complexité du problème notamment, la structure des données et la mémorisation des coups suivants. Pour commencer , nous avions décidé de repartir et bien organiser nos fichiers de façon à avoir des header (.h) et des fichiers(.C)dans lesquels nous coderons des fonctions liées au déplacement, à la fin de la partie et aux différents cas de victoire ou défaite. Après des longues heure d'écriture et tests, nous avons pu mettre sur pied un programme viable et stable répartie comme suite :

main.c : C'est le fichier principale, dans celui-ci nous trouverons plusieurs inclusions, la déclaration du tableau de char contenant les chiens etc.

Une fonction void bufferEmpty() qui servira à vider le buffer lorsqu'on souhaite rejouer une partie.

Dans le main, on commence par déclarer toutes nos variables, tableaux, pointeur..

Le coeur du jeu sera enveloppé dans une boucle do-while :

Dans celle-ci, une autre boucle do-while sera déclarée afin d'afficher la présentation et demande à l'utilisateur de saisir le mode de jeu auquel il souhaite jouer.

En récupérant la valeur de retour de scanf, on peut ensuite faire des tests dessus afin de quitter le programme.

Une fois que l'utilisateur a bien choisi son mode de jeu, on initialise la grille et les variables, le tout dans le do-while de façon à ce que le jeu se remette au départ lorsque l'utilisateur souhaite rejouer.

On a choisi de créer deux header afin de séparer l'affichage, de la mécanique du jeu.

display.h : Dans ce fichier header, nous trouverons tous les prototypes des fonctions d'affichage et d'initialisation ainsi que les différentes constantes utilisées dans le jeu.

On commence par définir une condition avec #ifndef DISPLAY_H_ pour éviter les inclusions infinies de fichiers puis on définie la constante DISPLAY_H_ en question. Le principe sera le même pour les futurs fichiers header. On inclue ensuite la bibliothèque standard, puis on définie les constantes et les prototypes.

Pour une question d'optimisation, on a préféré placer dans des constantes, le nombre de lignes et de colonnes que comportera le plateau afin de les modifier facilement à l'avenir si besoin.

Enfin, on termine par #endif pour montrer que la condition est finie.

display.c : Il s'agit du fichier complémentaire au précédent, celui-ci inclura ce dernier et pourra ainsi utiliser librement les fonctions et constantes qui en sont définies.

On y trouvera deux procédures :

```
void initialization(char grid[COLUMN\_NB][LINE\_NB], char * dogs);
void display(char grid[COLUMN\_NB][LINE\_NB]);
```

initialization servira à initialiser presque toutes les cases du tableau à la constante VOID '_' (déclarée dans display.h).

Pour cela, on prend la grille du plateau en paramètre char grid[COLUMN_NB][LINE_NB] ainsi qu'un autre tableau de char, (dogs).

Contrairement au buffles ou à l'indien, on a préféré placer les chiens dans un tableau de char au lieu de les mettre chacun dans une constante.

On crée ensuite deux variables entières : column et line qui serviront à parcourir la grille.

Dans la première boucle, on place les "vides" avec VOID en commençant par la ligne à 1 et la colonne à 0, (grid[0][1]), car la ligne 0 sera réservée au placement des buffles qui sera d'ailleur la deuxième boucle qui suit.

Dans la ligne 0 on place donc tous les buffles :

Pour placer l'indien, on divise le nombre total de colonne (COLUMN_NB) par 2 et on soustrait le nombre total de lignes (LINE_NB) par 2 pour qu'il se retrouve au milieu de l ?avant dernière ligne :

Pour les chiens, on en place deux de chaque côté de l'indien avec :

void display(char grid[COLUMN_NB][LINE_NB]) ;

Cette procédure sert à afficher l'état du jeu.

Elle prend qu'un seul paramètre, la grille grid[COLUMN_NB][LINE_NB], à l'aide de boucles for on affiche le contenu du tableau à deux dimensions, le numéro de lignes et de colonnes.

dogsMovement.c : Dans ce fichier, nous trouverons une seule fonction, dogsMove dont le prototype est le suivant :

```
int dogsMove(char grid[COLUMN\_NB][LINE\_NB],char *dogs,IndianDogPositionOldpoin
```

dogsMove sert à gérer les déplacements des chiens.

Sa valeur de retour est 1, si son déplacement respecte ceux énoncés dans les règles du jeu, 0 sinon.

Elle prend en paramètre la grille (grid[COLUMN_NB][LINE_NB]), le tableau de char contenant les symboles des chiens (dogs), et deux données : idopptr et idpc.

Idopptr : est le pointeur de structure ayant accès aux coordonnées des chiens idpc : contient les nouvelles coordonnées qu ?on souhaite attribuer au chien choisi.

Dans un switch case, on met à jour les coordonnées c et l en fonction du choix.

Puis on les copies dans des variables temporaires afin de ne pas les modifier directement.

On créer un autre switch case qui sera composé de huit cas parmi lesquels on on parcours une trajectoire en partant du chien qu'on a pas encore déplacé et on vérifie si on tombe sur les coordonnées qu'on souhaite lui attribuer. Si c'est le cas, alors on vérifie si entre les coordonnées qu'on souhaite lui attribuer et notre position actuelle, il n'y a pas d'obstacles, c'est-à-dire que si le chien ne saute pas au-dessus d'un buffle, d'un autre chien ou de l'indien pendant son déplacement.

Gameover.c : Dans ce fichier, nous trouverons une seule fonction, il s ?agit de testGameOver qui teste si le jeu est terminé ou bien s ?il continu :

Voici son prototype :

```
int testGameOver(char grid[COLUMN\_NB][LINE\_NB],int buffles)
```

testGameOver prend la grille du plateau (grid[COLUMN_NB][LINE_NB]) ainsi que le nombre de buffles restants (buffles).

Elle renvoie 3 valeurs de retours différentes selon le cas :

— 0 : si personne ne gagne donc le jeu continue,
— 1 : si le joueur buffle gagne,
— 2 : si le joueur indien gagne,

On commence par déclarer 3 variables locales : column, line et block.

A l'aide d'une boucle for et de column tu line, on parcourt la dernière ligne du plateau et on test à chaque fois si la case actuelle contient un buffle, si c'est le cas on renvoie immédiatement 1 pour indiquer que le joueur buffle a gagné.

Sinon on passe au teste suivant.

Si le nombre de buffles restants est plus petit ou égale à cinq alors on entre dans un cas particulier où l'on teste si les buffles restants sont bloqués par les chiens et l'indien c'est-à-direct qu'il ne peuvent plus avancer d'une case alors dans ce cas le joueur indien gagne.

Pour cela, on met notre variable locale à 0 directement après la première condition if(buffles $<= 5$) puis on parcourt la grille pour trouver nos buffles restants et on teste pour chaque buffles si la ligne suivante est différente de VOID dans ce cas on incrémenté block.

Si à la fin block est égale au nombre de buffles restants alors on en déduit que tous les buffles restants sont bloqués et on renvoie 2.

Pour savoir si le jeu continue il suffit de tester si le nombre de buffle est vrai dans se cas on retourne 0.

Enfin, par défaut on renvoie 2 pour indiquer que l'indien a gagné.

Gameplay.c : Dans ce fichier, on commence par inclure l'entête « gameplay.h »,

Puis on y trouveras deux fonctions :

```
int searchingBuffaloLine(char grid[COLUMN\_NB][LINE\_NB], buffaloPosition bp)
```

'searchingBuffaloLine' cherche la ligne correspondante à la colonne dans lequel se trouve le buffle.

Si on parvient à trouver un buffle avec la colonne qu'on lui envoie par l'intermédiaire de la donnée bp, sa valeur de retour est le numéro de la ligne correspondante, sinon -1.

Elle prend en paramètre, la grille du plateau ainsi qu'une donnée bp dans laquelle on y stockera la ligne et la colonne du buffle qu ?on souhaite déplacer.

La fonction tourne ainsi, on crée une variable i qui nous aidera à parcourir le nombre de ligne à l'aide d'une boucle for, juste avant, on décrémente la colonne afin d'obtenir le vrai nombre de colonne (car à l'affichage on commence à 1 qui correspond à 0 dans notre programme) et éviter un débordement lors de la condition if où l ?on vérifie si la case actuelle contient le buffle.

Si c'est le cas, on reincrémente la colonne avant de retourner la ligne trouvée +2.

Le +2 ici sert à indiquer qu'on envoie la ligne suivante côté utilisateur c ?est-à-dire que si on souhaite indiquer la vraie ligne suivante dans notre programme on aurait juste écrit +1 au lieu de +2, encore une fois c ?est une question d'affichage.

Sinon, si on n'a pas trouvé de buffle on renvoie bp.line = -1 de façon à avertir le programme qu'il y'a une erreur dans la saisie du buffle à déplacer celui-ci ne fait plus partie du plateau car il a été capturé.

```
int searchingBuffaloLine(char grid[COLUMN\_NB][LINE\_NB], buffaloPosition bp){
int i;
--bp.column;
  for(i=0; i
  if(grid[bp.column][i] == BUFFALO){
++bp.column;
return bp.line = i+2;
}
return -1;
}
```

Set permet de placer les pièces : Voici son prototype

```
int set(char grid[COLUMN_NB][LINE_NB],char * dogs, int * buffles,
int player,IndianDogPositionOldpointer,idopptr,
IndianDogPositionCurr,idpc, buffaloPosition bp)
```

Elle prend en paramètres la grille du jeu (grid[COLUMN_NB][LINE_NB]), le tableau de char contenant les chiens (dogs), le nombre de buffles restant (buffles), le numéro du joueur et trois données la première est un pointeur ayant accès aux coordonnées des chiens et de l'indien (idopptr), la deuxième contient les nouvelles coordonnées qu ?on souhaite attribuer ou aux chiens ou à l'indien, cela dépend du choix, enfin la dernière contient les coordonnées du buffle qu'on souhaite déplacer si c'est le joueur buffle qui joue.

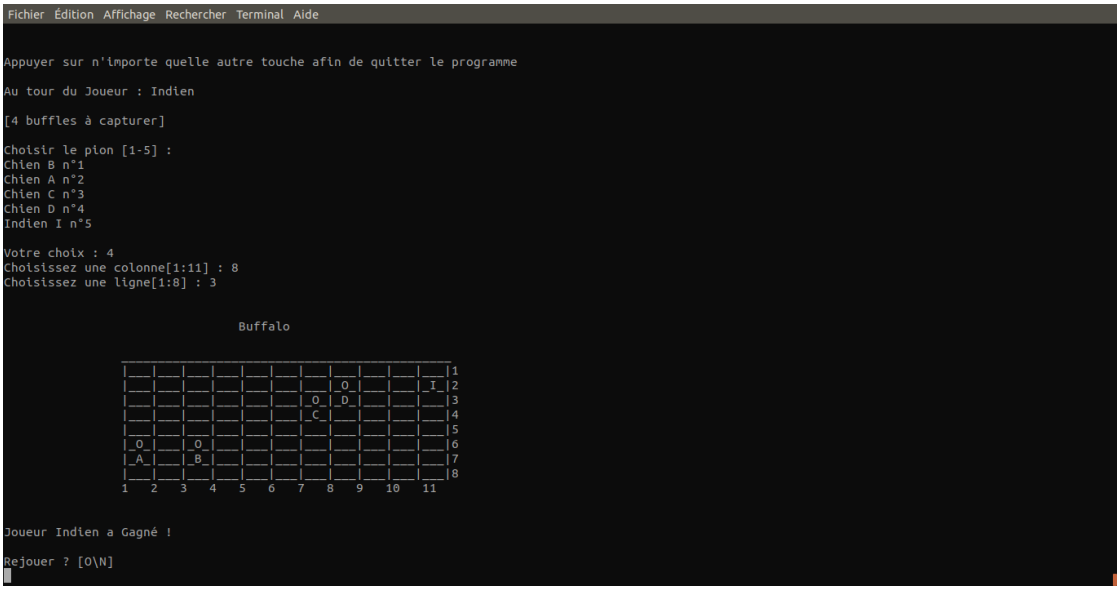
La fonction set renvoie 1 si elle est parvenue à déplacer la pièce sélectionnée, 0 sinon.

Pour des questions d'affichage, on décrémente la ligne et la colonne qu'on lui envoie et on vérifie dans des condition if, si la valeur de Player est positive alors c'est le joueur buffle qui joue, sinon c'est le joueur indien.

Dans le cas du joueur buffle on vérifie d'abord si la colonne et la ligne qu'on lui envoie sont bien dans l'intervalle, sinon on retourne 0.

Si elles étaient bien dans l'intervalle, alors on vérifie si la ligne suivant le buffle est vide dans ce cas on met la valeur qui contient le buffle à VOID et on met les nouvelles coordonnées à BUFFALO puis on retourne 1.

Le fonctionnement est le même pour le joueur indien si ce n'est qu'on l'empêche de jouer sur la ligne 0 et la ligne 7, de plus l'indien a le droit de se déplacer sur un buffle. On définit un switch case pour déterminer quelle pièce déplacée en fonction du choix.



Conclusion

Le projet fut un challenge qui nous a ainsi permis d'évoluer et de voir la complexité du language et certaines problematiques liées à celui-ci par le processus de debuggage.

References

http ://www.ai.univ-paris8.fr/ jj/Cours/LI2/PI$2_2$018.*html*
https ://www.edu.upmc.fr/c2i/ressources/latex/aide-memoire.pdf
https ://jeuxstrategieter.free.fr/Prairie_complet.php
https ://openclassrooms.com/fr/dashboard