BORDEAUX INP Ensc

PROJET XMAS TREATS

DEUX MILLE QUARANTREAT

Justifications techniques

Léa Jourdan
Maxime Etcheverry

Sommaire

DEUX MILLE QUARANTREAT 1
Sommaire **2**
Introduction **3**
Format des données **4**
Données des tableaux 4
Structuration du code **5**
Découpage 5
Dynamique d'exécution 5
LancerPartie() 5
AfficherLentement(string) 6
Paramètres: 6
Description: 6
Appel d'autres fonctions: 6
InitTableauZeros(int[][]) 7
Paramètres: 7
Description: 7
Appel d'autres fonctions: 7
AjouterBonbon(int [][]) 7
Paramètres: 7
Description: 7
Appel d'autres fonctions: 7
Lignes (int [][]) 7
Paramètres: 7
Description: 7
Appel d'autres fonctions: 7
CompterZeros(int[]) 8
Paramètres: 8
Description: 8
Appel d'autres fonctions: 8
AfficherTableau(int[][]) 8
Paramètres: 8
Description: 8

Appel d'autres fonctions: 8
AfficherItem(int) 8
Paramètres: 8
Description: 8
Appel d'autres fonctions: 8
CalculerScore (int [][]) 9
Paramètres: 9
Description: 9
Appel d'autres fonctions: 9
Swiper(int [][] , char) 9
Paramètres: 9
Description: 9
Appel d'autres fonctions: 9
Affichage du meilleur score 9
Fonctionnalités Bonus 10
Fonctionnalités d'affichage 10
Fonctionnalités de mode de jeu 10
Organisation de travail **11**
Planning 11
Répartition des tâches 11

Introduction

L'objectif de ce projet était de programmer un jeu ressemblant au jeu 2048 mais avec des sucreries à la place des chiffres, en mode Console et en utilisant le langage C#. La grille de jeu prend la forme d'un tableau de tableaux, c'est cette grille qui va évoluer au fur et à mesure de la partie. La grille va voir ses items se déplacer selon le choix du joueur et fusionner. La grille contient au départ deux bonbons placés aléatoirement. Le joueur va chercher à les fusionner, selon le mode de jeu pour atteindre le meilleur score ou pour atteindre un score prédéterminé mais avec le minimum de coups.

Les contraintes techniques :

La programmation a été faite en C#, en mode console, dans les conditions habituelles des TP. La programmation orientée objet est elle aussi exclue. Le code source respecte les règles

d'indentations, la norme camelCase. Des commentaires sont présents pour expliquer des parties de code complexes.

I. Format des données

Nous avons fait le choix d'utiliser des tableaux imbriqués de la forme `int [][]` et non des tableaux multidimensionnels de la forme `int [,]`. Ce choix a été notre première difficulté, sur les conseils de Mme UNREIN, nous avons opté pour les tableaux imbriqués en raison de leur flexibilité. En effet, ils offrent une plus grande flexibilité en termes de taille, car chaque sous-tableau peut avoir une taille différente. De plus, ils peuvent être plus faciles à utiliser dans certains cas, surtout lorsque la taille des dimensions est dynamique ou varie d'une ligne à l'autre, nous trouvions que la méthode d'accès à un caractère précis du tableau était plus abordable et coïncidait mieux avec notre façon de coder les sous programmes.

Données des tableaux

Nous avons décidé d'utiliser des tableaux d'entiers plutôt que des tableaux contenant directement les caractères. Ainsi le chiffre 1 correspond au bonbon, le 2 au réglisse, le 3 au cookie et le 4 au sucre d'orge. Cela permet de faciliter les fusions. En effet, après avoir vérifié qu'une fusion allait avoir lieu grâce à la variable $fusion \in \{0, 1\}$, on incrémente de 1 la valeur de la case concernée.

La transformation des chiffres en caractères représentant les bonbons se fait grâce à la fonction `AfficherItem( int numero)`.

.

II. Structuration du code

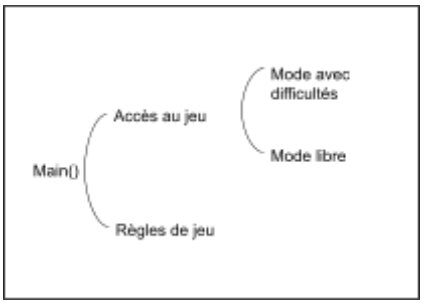
Découpage

Nous avons opté pour un découpage en sous programmes pour chaque action de la partie. Cela facilite la répartition des tâches à effectuer et l'identification des problèmes en cas de bug tout en rendant le programme principal très lisible.

Dynamique d'exécution

LancerPartie()

La fonction `LancerPartie()` est notre programme principal, cette procédure appelle chacun des sous-programmes. La fonction `LancerPartie()` est d'abord séparée en 2 parties : la partie appelant les fonctions et la partie affichant les règles. La première partie est elle-même séparée en plusieurs sous parties: le joueur a le choix entre plusieurs modes de jeux proposant plusieurs difficultés et un mode libre.



La partie des Règles de Jeu affiche les règles si le joueur a appuyé sur "r" lorsque ça lui a été proposé de le faire. Cette partie appelle seulement la procédure `AfficherLentement (string )`.

Le joueur accède à la partie d'accès au jeu (cf schéma) s'il a appuyé sur la touche "a". Il doit à présent choisir entre 4 options: un mode de jeu "facile" , un "moyen", un "difficile" et un

mode libre “record”. Le code sépare le mode libre des autres grâce à un `“if{}” “else{}”`. Les modes ont le même principe de fonctionnement, seules des conditions d’arrêts différentes permettent de les distinguer. Détaillons le principe de fonctionnement d’une partie. Un tableau est initialisé à la taille n (la valeur de n sera discutée plus tard). Ce tableau se remplit de zéros avec l’appel de la fonction `InitTableauZeros (int [][])`, la fonction `AjouterBonbon ( int [][] )` permet de placer aléatoirement un bonbon, elle est donc appelée deux fois en début de partie. Le tableau s’affiche avec la fonction `AfficherTableau(int[][])` puis le score est calculé avec la fonction `CalculerScore (int [][])`. Une fois le jeu mis en place, on rentre dans une boucle while représentant les coups successifs du joueur. Le score est affiché, le joueur peut effectuer un déplacement avec l’appel de la fonction `Swiper (int[][], char)`, un nouveau bonbon aléatoire est placé, le tableau s’affiche ensuite avec tous les changements effectués et le score est calculé.

Pour le mode libre la complexité supplémentaire était d’enregistrer le meilleur score parmi les parties précédentes pour pouvoir les comparer à celui de la partie en cours. Pour les autres modes il s’agit de faire tourner la partie tant que le joueur n’a pas atteint son nombre de coups maximum ou qu’il a un score inférieur à 50. Une fois son dernier coup effectué, s’il a un score égal ou supérieur à 50, il a gagné, si ce n’est pas le cas il a perdu.

`AfficherLentement(string)`

Paramètres:

String texte

Description:

Cette fonction utilise la commande `System.Threading.Thread.Sleep` pour afficher le texte au fur et à mesure et non d’un coup.

Appel d’autres fonctions:

Aucun

InitTableauZeros(int[][])

Paramètres:

`int [] [] tableauFinal`

Description:

Cette fonction remplit le tableau de zéros grâce à une boucle for.

Appel d'autres fonctions:

Aucun

AjouterBonbon(int [][])

Paramètres:

`int [] [] tableauJeu`

Description:

Cette fonction ajoute un bonbon dans une case aléatoire vide. Une case vide est une case avec la valeur zéro.

Appel d'autres fonctions:

`Lignes (tableauJeu)`

Lignes (int [][])

Paramètres:

`int [] [] tableauJeu`

Description:

Cette fonction trouve une ligne au hasard contenant au moins une case vide.

Appel d'autres fonctions:

`CompterZeros ( int [])`

CompterZeros(int[])

Paramètres:

`int [] tableau`

Description:

Cette fonction compte le nombre de zéros dans un tableau à une dimension.

Appel d'autres fonctions:

Aucun

AfficherTableau(int[][])

Paramètres:

`int [] tableau`

Description:

Cette fonction permet de passer du tableau à l'affichage dans la console et repose sur une boucle dont la taille dépend de celle rentrée par l'utilisateur

Appel d'autres fonctions:

`AfficherItem(int)`

AfficherItem(int)

Paramètres:

`int [] numero`

Description:

Cette fonction est ce qui nous permet de passer des entiers du tableau à leur correspondance dans le jeu. Sans cette fonction on aurait utilisé des tableaux de caractères

Appel d'autres fonctions:

Aucun

`CalculerScore (int [][])`

Paramètres:

```
int [] [] tableauJeu
```

Description:

Cette fonction calcule le score grâce à une boucle for. On parcourt les cases une à une en ajoutant à chaque fois au score la valeur des bonbons correspondante.

Appel d'autres fonctions:

Aucun

`Swiper(int [][] , char)`

Paramètres:

```
int [] [] tableauJeu
char direction
```

Description:

Cette fonction réalise le déplacement et la fusion des items grâce à deux boucles for permettant de parcourir les colonnes puis les lignes pour avoir accès à chaque case. Il doit y avoir une seule fusion par tour donc la variable int fusion qui agit comme un booléen permet de vérifier que c'est bien le cas. Admettons que le but est de "swiper" vers le haut, la case observée prend la valeur de celle d'en dessous et ainsi de suite pour tout remonter. Quant à la fusion, si les deux cases observées ont la même valeur alors on incrémente de 1 la case du haut et on donne la valeur zéro à celle du bas. Le principe est le même pour les trois autres directions, il faut juste changer les indices.

Appel d'autres fonctions:

Aucun

Affichage du meilleur score

L'affichage du meilleur score demande l'utilisation des bibliothèques System.IO et System.Text. Ces bibliothèques permettent de lier un document .txt au code du programme. Le document texte s'appelle Score.txt et contient le meilleur score des parties jouées précédemment. A chaque partie la commande `StreamReader` permet de lire la première ligne du document texte. Cette ligne représente le meilleur score précédent, si ce score est

inférieur au nouveau score de la partie actuelle alors il sera remplacé par ce dernier grâce à la commande `StreamWriter`.

Fonctionnalités Bonus

Nous allons procéder à lister les différentes fonctionnalités bonus effectuées et à expliquer les difficultés que nous avons rencontré pour chacune d'entre elles

Fonctionnalités d'affichage

Pour l'affichage nous avons ajouté deux fonctionnalités:

- Les couleurs des bonbons: cette fonctionnalité a été rendu possible par la commande système Console.ForegroundColor
- La vitesse d'affichage dans la console: grâce à la commande `System.Threading.Thread.Sleep` nous avons pu afficher les caractères un à un, à la vitesse souhaitée.

Fonctionnalités de mode de jeu

Nous avon décidé d'implémenter différents mode de jeu:

- Le mode libre correspond à la version originale du jeu avec un choix de taille du tableau. Pour se faire, il a suffit d'utiliser un `Console.ReadLine` dans lequel le joueur rentre la taille du tableau souhaité.
- La sauvegarde du meilleur score: cette fonctionnalité a été incluse dans le mode libre afin d'y ajouter de l'intérêt et un esprit "compétitif". La principale difficulté a résidé dans la maîtrise des commandes `StreamReader` et `StreamWriter`.
- Les modes à difficulté qui consistent à atteindre un nombre de points en un nombre de tours précis. Il nous a fallu inclure des variables de tour et de points et faire des comparaisons après chaque tour passés
- Affichage à chaque tour du score: nous avons utilisé la fonction `Score(int)` dans le programme principal

III. Organisation de travail

Planning

Les premières réunions de travail ont servi à mettre sur papier le découpage des différents sous programmes et le format des tableaux de données: nous avons visualisé le squelette du programme lors d'un premier jet (4h pleines). Dans un second temps, nous avons codé tous les sous programmes (environ 2 semaines). Enfin nous avons écrit le programme principal et corrigé les bugs (7h pleines).
Le temps restant nous a permis d'ajouter les éventuels bonus et d'améliorer la lisibilité du code et de l'affichage (environ 1 semaine).

Répartition des tâches

La majorité du travail a été réalisée à deux comme par exemple le squelette et les sous programmes les plus importants (`Lignes (int [][])`, `AjouterBonbon (int [][])` et une partie du `LancerPartie()`). Cependant certaines tâches ont quand même été divisées afin d'aller plus vite en se basant sur les envies et/ou compétences de chacun.