Projet IF2B A2023 (2048 Plus)

Ganne Charles Romain Moulin Quentin Ricci

24 octobre 2025

Table des matières

1 Introduction

Ce projet, réalisé dans le cadre de l'UE IF2B, est une implémentation du jeu 2048 Plus en C. Ce rapport présente le travail réalisé par le groupe. Le partage du code afin de travailler de manière colaborative a été fait avec GitHub (lien complet : https ://github.com/charlesganne/jeux-et-petits-projets-pour-apprendre/tree/main/2048). Nous nous intéresseront dans un premier temps à la structure générale du code, pour ensuite regarder plus en détail les choix faits et libertés prises lors du développement du jeu. Finalement nous présenterons le resultat final, feront un bilan de ce qui a été réussi et des points d'amélioration éventuels.

2 Structure générale

Le code s'articule en trois fichiers de code principaux, ainsi que le fichier de la grille pour le mode puzzle.

Le fichier 2048.c contient le menu principale du jeu. Il demande à l'utilisateur le mode de jeu dans lequel il veut jouer, puis selon sa réponse appellera une des trois fonctions suivantes : normal, duo et puzzle. Lors de l'exécution de ce code, l'utilisateur est sensé appuyer sur 4 touches différentes : n, d, p et la touche entrer. De manière générale, et dans toute la suite, le programme admet que l'utilisateur respecte à la lettre les instructions affichées à l'écran (que l'entrée est valide).

Le fichier fonctions_generales.c contient le corps même du jeu, toutes les fonctions importantes (initialiser la grille, bouger les pieces, ...). Leur fonctionnement sera étudié plus en détail plus loin dans ce rapport.

Le fichier fonctions_generales.h contient les déclarations des prototypes des fonctions qui auront besoin d'être utilisées dans le programme, et seulement celles ci, certaines fonctions sont seulement utilies au fichier fonctions_generales.c, et ne sont pas déclarées dans le fichier .h. Elles ne sont pas nécessaires en dehors de celui-ci.

Le fichier test.txt contient la grille à importer pour le mode puzzle.

3 Détails d'implémentation

3.1 Le fichier 2048.c

Ce fichier est essentiellement constitué de "printf" et de lectures d'entrée utilisateur. Il importe dans un premier temps les librairies nécessaires dont la librairie fonctions_generales.h qui contient les fonctions appelées dans la boucle principale.

Une variable booléenne jouer est initialisée à "vrai" afin d'indiquer à la boucle qu'une partie est lancée. La boucle principale se répète indéfiniment tant que jouer vaut true (vrai). Cette variable sera mise à jour à la fin de chaque partie selon le choix de l'utilisateur de recommencer une partie ou non.

3.2 Les entrées utilisateur

De manière générale, toutes les entrées de l'utilisateur sont gérées de la manière suivante :

Listing 1 – code pour les entrées utilisateur

```
1 char buffer[10];
2 printf("Entrez␣la␣donnee␣demandee\n");
3 fgets(buffer, sizeof(buffer), stdin);
4 // traitement de la ligne selon les modalites
```

* Déclaration d'un buffer sous forme d'une chaîne de caractères, et ce même si le joueur entre un chiffre, ce dernier sera converti en int selon les besoins du programme.

* Affichage d'un message indiquant à l'utilisateur précisément ce qu'il doit entrer pour que la suite du programme s'exécute correctement.

* La fonction "fgets" lit la prochaine ligne qui sera sur l'entrée standard, donc celle que tapera l'utilisateur dans le terminal dans des conditions normales d'exécution.

* Le traitement de l'entrée de l'utilisateur est différent selon les cas, mais en général : on remplace le dernier caracter qui est le \n de la touche entrer par un \0, puis soit on converti en entier avec la fonction "atoi", soit on regarde un caracter à une place précise qui est généralement une instruction de modification de la grille, soit on regarde toute la ligne, comme par exemple quand on a besoin d'ouvrir le fichier.

3.3 L'instruction NETTOIE_TERMINAL

Une instruction préprocesseur au debut du fichier fonctions_generales.c permet de nettoyer le terminal sur linux, mac et windows en ne modifiant qu'une seule ligne du code. Lors de la compilation du programme, le compilateur lit d'abord ces instructions (précédées d'un #), puis dans le cas de define, remplace toutes les occurences de "NETTOIE_TERMINAL" dans le code par la valeur qui suit lors de la déclaration de l'instruction.
Il est important d'adapter cette ligne au système d'exploitation sur lequel tournera le programme avant de l'exécuter. Sa compilation ne posera aucun probleme, mais lors de l'exécution, la commande tapée pourrait être invalide.

3.4 La déclaration et la libération de la mémoire

Listing 2 – code pour allouer une grille

```
1 int** grille = (int**)malloc(dim * sizeof(int*));
2 for(int i = 0; i < dim; i++)
3 {
4     grille[i] = (int*)calloc(dim, sizeof(int));
5 }
```

Dans un premier temps on alloue un pointeur qui pointe vers "dim" pointeurs vers des entiers avec la fonction malloc. Puis pour chaque pointeur ainsi aloué, on fournit une adresse vers un tableau de dim entiers, dim étant la taille de la grille. Cette mémoire a été allouée dynamiquement et ne sera donc pas détruite à la fin de la fonction.

Cependant, comme cette mémoire a été allouée dynamiquement, il faudra penser à la libérer avant de quitter le programme. On fait cela avec le code suivant :

Listing 3 – code pour libérer une grille

```
1 for(int i = 0; i < dim; i++)
2 {
3     free(grille[i]);
4 }
5 free(grille);
```

On libère chaque tableau, puis on libère le grand tableau.

3.5 L'affichage de la grille

Les conventions sont les suivantes : un 0 encode une case vide, un entier naturel qui est une puissance de 2 est une case occupée par une valeur valide, un -1 encode un obstacle ayant une position fixe. Les seules valeurs qui seront vraiment regardées par l'algorithme seront les 0 et les -1, car les autres seront nécessairement des puissance de 2 (en exécution nominale du code).

Listing 4 – code pour afficher une grille

```
1  if(grille[i][j] == 0)
2  {
3      printf("␣␣␣␣␣");
4  }
5  else if(grille[i][j] == -1)
6  {
7      printf("X␣␣␣␣");
8  }
9  else
10 {
11     printf("%-5d", grille[i][j]);
12 }
```

Pour un 0 on affiche simplement 5 espaces (une case vide). Pour un -1, on affiche un X et 4 espaces (un obstacle), et sinon, on affiche la valeur (%d) et on complète à gauche avec des espaces pour avoir exactement 5 caractères (%-5d).
Pour délimiter la zone où se situe la grille, on affiche une ligne de "+- - -+" au début, à la fin ; et à chaque début et fin de ligne on affiche une "|".

La fonction qui affiche les deux grilles côte à côte est très similaire. La seule différence est qu'elle affiche chaque ligne en même temps. Pour ce faire elle retourne à la ligne seulement apres avoir affiché la ligne des deux grilles, et pas seulement après avoir affiché une ligne comme dans la fonction précédente.

3.6 Le mouvement des pièces

Les 4 fonctions (haut bas gauche droite) étant sensiblement similaires, seule la fonction déplacant les pièces vers le haut sera expliquée vraiment en détail dans ce rapport.

Listing 5 – bouger les pièces vers le haut

```
1  void haut (int** grille, int dim) {
2      for (int ord = 0; ord < dim; ord++) {
3          for (int abs = 0; abs < dim; abs++) {
4              int case_courante = grille[ord][abs];
5              if (case_courante != 0) {
6                  int indice_case_suivante = ord+1;
7                  while (indice_case_suivante < dim) {
8                      int case_suivante = grille[
                           indice_case_suivante][abs];
9                      if (case_suivante == -1) {
10                         break;
11                     }
12                     if (case_suivante != 0) {
13                         if (case_courante ==
                               case_suivante) {
14                             grille[ord][abs] +=
                                   grille[
                                   indice_case_suivante
                                   ][abs];
15                             grille[
                                   indice_case_suivante
                                   ][abs] = 0;
16                         }
17                         break;
18                     }
19                     indice_case_suivante++;
20                 }
21             }
22         }
23     }
24
```

```
25      for (int ord = 0; ord < dim-1; ord++) {
26          for (int abs = 0; abs < dim; abs++) {
27          int case_courante = grille[ord][abs];
28              if (case_courante == 0) {
29                  int indice_case_suivante = ord+1;
30                  while (indice_case_suivante < dim &&
                        grille[indice_case_suivante][abs
                       ] != -1) {
31                      int case_suivante = grille[
                           indice_case_suivante][abs];
32                      if (case_suivante != 0) {
33                          grille[ord][abs] =
                               case_suivante;
34                          grille[indice_case_suivante
                               ][abs] = 0;
35                          break;
36                      }
37                      indice_case_suivante++;
38                  }
39              }
40          }
41      }
42  }
```

Cette fonction est spécialement conçue pour s'adapter aux deux modes de jeu (puzzle et non puzzle)
L'idée de l'algorithme est de sommer toutes les cases sommables selon les règles sans se soucier de les écraser en haut, puis de les projetter seulement ensuite. Elle se découpe donc en deux blocs principaux.

Le premier bloc parcourt la grille du haut vers le bas pour traiter en colonnes (ord = ordonnée et abs = abscisse), et de gauche à droite afin que chaque colonne, même si elle est traitée indépendament soit traitée en même temps.
La variable case_courante contient la case considérée dans la grille (chaque case se trouvera à un moment dans cette variable étant donné le parcourt vu précédement de la grille). Si cette case est vide, on assigne à la variable indice_case_suivante l'indice en ordonnée de la case suivante.
On parcourt ensuite le reste de la colonne tant que l'on ne tombe pas sur

un obstacle (case_suivante != -1) ou tant que l'on ne tombe pas sur la fin de la grille (indice_case_suivante < dim), on continue de passer à la case suivante.
Si la case n'est pas vide, on sait déjà qu'elle n'est ni un obstacle ni un mur, on regarde alors si elle est égale à la case courante. Si oui on l'ajoutte et si non on sort de la boucle et passe à la case suivante (on ne peut plus chercher de paire avec la case courante car il y a une case différente entre).
Sinon on passe à la casse suivante.

Une fois que toutes les cases sommables ont été sommées, il faut les projetter. On fait exactement le même parcourt de la grille, et à chaque fois qu'on tombe sur une case vide et on regarde toutes les cases suivantes jusqu'à tomber sur un obstacle ou un mur, auquel cas on passe à la case suivante. Si on tombe sur une case non vide, on permute la case vide et la case non vide.

3.7 Vérifier que l'entrée utilisateur est valide dans le contexte du jeu

La fonction est_valide prend en entrée la grille, ses dimensions et l'entrée de l'utilisateur. La grille est copiée intégralement dans une nouvelle grille, la modification souhaitée par le joueur est appliquée à la nouvelle grille. On teste ensuite que les deux grilles sont différentes. Si elles ne le sont pas, le mouvement n'a bougé aucune pièce, et le mouvement n'est pas valide, on retourne alors false. Sinon il est valide, on retourne true.
Cette fonction admet que l'utilisateur entre une des lettres zqsd. Le comportement reste indéterminé si l'entrée n'est pas valide dans le contexte du jeu (il faut toujours que le joueur entre z, q, s ou d).

3.8 L'insertion aléatoire dans la grille

La fonction qui insert un nombre aléatoire à une position valide prend en entrée la grille et sa taille. Elle compte le nombre de cases libres dans la grille. Elle génère alors un nombre aléatoire entre 0 et ce nombre de case libres. Deux boucles imbriquées parcourent ensuite seulement les cases libres de la grille en décrémentant progressivement la variable qui contient la position aléatoire de la case seulement si elles tombe sur une case libre. Quand la vriable index vaut 0, c'est qu'on est sur la casse séléctionnée aléatoirement, on choisit à nouveau aléatoirement une valeur : 2 ou 4, et elle est attribuée à la case.

3.9 La fin de partie

La partie se termine si l'une des trois conditions suivantes est invoquée :

— Le joueur n'a plus aucun mouvement valide, il a alors perdu.
— Le joueur gagne la partie, lorsqu'il obtient une case indiquant la valeur 2048.
— Le programme s'arrête de manière inattendue à cause d'une erreure.

Dans les deux premiers cas, un message est alors affiché dans la console pour indiquer au joueur l'état de la partie, son score, et lui proposer une nouvelle partie.
La fonction partie_gagnee parcourt toute la grille à la recherche de la valeur 2048, et la fonction situation_bloquee vérifie qu'il existe au moins un mouvement valide parmis les 4 possibles.

3.10 La lecture du fichier pour le mode puzzle

La fonction read_file est une fonction qui ne sera utilisée que pour initialiser le mode puzzle. Elle prend en entrée une chaîne de caractères ainsi qu'un pointeur vers un entier. Ce pointeur est une adresse vers un entier (passage par référence donc) qui aura été déclaré préalablement. À cette adresse sera placé lors de l'exécution de la fonction la taille du fichier. Cette methode est employée car une fonction ne peut pas retourner deux valeurs.
La fonction commence par ouvrir le fichier en lecture. Si l'ouverture échoue, elle affiche un message d'erreur sur la sortie standard et retourne le pointeur NULL. L'exécution de cette fonction présuppose que le fichier est de la forme suivante :

Listing 6 – fichier de la grille

```
1 <debut du fichier><dimension de la grille>
2 v v ... v X v
3 X v X ... X v
4 .   .         .
5 .     .       .
6 .       .     .
7 v X v v ... v
8 <fin du fichier avec retour a la ligne>
```

Si le fichier s'est ouvert correctement, la fonction lit la première ligne du fichier qui contient les dimensions du fichier, convertit la chaîne de caractères en un entier, et assigne à l'adresse taille_grille la taille de la grille. Le curseur est ensuite replacé au tout début du fichier car le buffer utilisé n'est plus le même. Une boucle for qui bouclera autant de fois qu'il y a de lignes dans le fichier (les dimensions de la grille). Pour chaque passage dans la boucle, une ligne du fichier est lue, puis une deuxième boucle for parcourt dim fois la ligne pour "découper" chaque valeur qui est intéressante avec la fonction strtok. La fonction strtok est ici paramétrée pour éviter les espaces " " et les fins de ligne "\n". Chaque valeur isolée avec strtok est convertie selon si elle est un obstacle ou un nombre, puis ajoutée dans la grille allouée préalablement. La grille est ensuite retournée sous forme de pointeur vers un pointeur vers un entier.

3.11 Les modes de jeu

Chaque mode de jeu fonctionne de manière très similaire. Soit il demande à l'utilisateur la taille de la grille, l'initialise et commence la partie, soit il demande à l'utilisateur le chemin vers le fichier qui contient la grille, convertit le fichier en grille et commence la partie.
Un tour de jeu dans une partie se déroule de la manière suivante :

- Demander à l'utilisateur quel mouvement il veut appliquer à la grille
- Si le mouvement est valide, il est appliqué à la grille, et un nouveau tour de jeu peut commencer
- Si le mouvement est invalide (ne fait bouger aucune pièce), un message d'erreur est affiché, et l'utilisateur est incité à fournir une entrée valide lors du prochain tour de jeu.

Le code qui effectue un tour de jeu est logé dans une boucle while qui tournera indéfiniment tant que la partie de sera pas terminée. La partie se termine si le joueur n'a plus aucun mouvement valide ou si le joueur gagne en faisant apparaitre la valeur 2048 quelquepart dans la grille.
Lorsque la partie se termine, si le joueur a gagné, un message de victoire est affiché, et si le joueur a perdu, cela lui est simplement indiqué. Le score est ensuite affiché. On lui propose ensuite de recommencer une nouvelle partie. Si il veut retenter sa chance, la variable true est retournée par la fonction, sinon, elle retourne false. Ce retour est ensuite traité par la fonction de menu principale pour initialiser une nouvelle partie ou dire "au revoir".

4 Compilation

Pour compiler le projet, il faut placer tous les fichiers du code dans le même dossier, s'y placer depuis un terminal avec la commande cd, puis taper la commande "gcc *.c" qui compile tous les fichiers et crée un fichier a.out dans le dossier courant.

5 Conclusion

5.1 Le rendu final

Le code est fonctionel, le joueur peut jouer dans les trois modes de jeu demandé si il fournit des entrées correctes au programme. Le rendu graphique est minimaliste, il s'agit simplement de printf dans la console. La mémoire est bien libérée à la fin de l'exécution du code afin d'éviter de mettre en danger le système.

5.2 Les faiblesses du code et les améliorations possibles

— L'affichage pourrait être en couleur et/ou dans une interface graphique
— Les entrées de l'utilisateur sont supposées valides sans en rien le contraindre. Il pourrait être une bonne idée d'ajouter des sécurités et des messages d'erreure si l'entrée est totalement invalide (pas si il fournit un z alors que la grille ne peut être projettée vers le haut, mais si il fournit la valeur 12 ou la chaine dshbvfgvniedgnfhegnfcgefezrfgnergfnizegnf). La gestoin des erreures ne pourrait jamais être parfaite, mais elle pourrait être améliorée.