



FILIÈRE INFORMATIQUE, 1ÈRE ANNÉE

Rapport de projet S5

## "Tilings"

*AUTEURS*

Aymane LAMHAMDI

Mahmoud KASOUR

*ENCADRANTS*

M. David Renault

M. Denis Barthou

4 décembre 2020

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Le jeu . . . . .	2
1.2	Problématiques et difficultés . . . . .	2
1.3	Méthodologie . . . . .	2
<b>2</b>	<b>Outils utilisés</b>	<b>3</b>
2.1	Éditeur de texte . . . . .	3
2.2	L’outil Git . . . . .	3
2.3	Makefile . . . . .	3
2.4	L <sup>A</sup> T <sub>E</sub> X . . . . .	3
<b>3</b>	<b>Base Version</b>	<b>3</b>
3.1	Les tuiles . . . . .	4
3.2	La file . . . . .	5
3.2.1	Structure de file . . . . .	5
3.2.2	Les fonctions de la file . . . . .	5
3.3	Le plateau du jeu . . . . .	8
3.3.1	Structure du plateau . . . . .	9
3.3.2	Les fonctions du plateau . . . . .	9
3.4	Les joueurs . . . . .	11
3.4.1	Structure du joueur . . . . .	11
3.4.2	Les fonctions player . . . . .	12
3.5	L’aléatoire en C . . . . .	12
3.6	Le jeu de tuiles . . . . .	13
3.6.1	Mélanger et distribuer . . . . .	13
3.6.2	Remplir et sélectionner . . . . .	14
3.7	Fonction de choix de paramètres de l’entrée . . . . .	15
3.8	Conception et déroulement du jeu . . . . .	15
3.9	Organisation et dépendances des fichiers . . . . .	16
<b>4</b>	<b>Achievement 1</b>	<b>16</b>
4.1	Changements et ajouts effectués . . . . .	17
4.1.1	Structure du joueur . . . . .	17
4.1.2	Structure des motifs et fonctions associées . . . . .	17
4.2	Conception et déroulement du jeu . . . . .	19
4.3	Organisation et dépendances de fichiers . . . . .	20
<b>5</b>	<b>Tests et vérifications des fonctions</b>	<b>21</b>
<b>6</b>	<b>Améliorations possibles</b>	<b>22</b>
<b>7</b>	<b>Conclusion</b>	<b>22</b>

# 1 Introduction

Ce rapport a pour objectif de décrire l'ensemble du processus de réalisation du 1er projet de programmation des élèves en première année d'informatique à l'ENSEIRB-MATMECA. Le projet consiste en l'implémentation du jeu des tuiles "Tiling", en langage C. Ce sujet de projet est construit sur la notion d'achèvement. Nous devons, dans un premier temps, établir une première version selon des consignes précises (la base version), puis réaliser différents achievements dévoilés au cours de la réalisation du projet.

## 1.1 Le jeu

Tiling est un jeu qui se joue de 2 jusqu'aux 20 joueurs sur un plateau carré de taille  $DIM \times DIM$ . Le principe est de réaliser des pavages du plan à l'aide des tuiles (Tuiles de Wang). Les tuiles sont distribuées aléatoirement et équitablement aux joueurs et chacun fait une pile de l'ensemble de ses tuiles, et les joue une par une.

Les règles du jeu sont de respecter la contiguïté et la connexité des tuiles. Le joueur qui ne peut pas poser de tuile à son tour, il remet sa tuile en dessous de sa pile et passe son tour. Si le joueur pose sa tuile sans respecter les règles, il est éliminé, ses tuiles restent en jeu. Si aucun joueur ne peut jouer, ou si un des joueurs a joué sa dernière tuile, la partie se termine.

Le joueur ayant posé le plus de tuiles gagne la partie.

## 1.2 Problématiques et difficultés

L'objectif est donc la réalisation de ce jeu en C. Une décomposition du problème en sous-problèmes est nécessaire pour éviter la redondance de l'information et simplifier au maximum chaque problème. Il faut en premier temps trouver une ou des structures de données adaptées au jeu. Une structuration de données efficace est fondamentale pour réaliser le projet.

## 1.3 Méthodologie

Un tel programme nécessite de définir tout d'abord la structure d'une tuile, du tableau, des joueurs et aussi de créer une structure de file pour gérer les jeux des joueurs et enfin mettre en place la boucle de jeu.

Nous présenterons dans un premier temps nos choix de structure de données et la création d'une structure de file des tuiles. Puis dans un second temps, nous présenterons les différentes fonctions nécessaires que nous avons implémentées dans le programme, évaluer leur complexité en temps et en espace. Ensuite, nous exposerons les différents tests qui permettent de valider le bon fonctionnement de notre code. Enfin, nous présenterons dans une dernière partie la finalité de notre programme et de potentielles améliorations.

## 2 Outils utilisés

Nous avons choisi d'utiliser un certain nombre d'outils d'aide au développement pour ce projet. L'utilisation de certains a été plus que conseillée (notamment l'utilisation du Makefile).

### 2.1 Éditeur de texte

Au début, nous avons commencé avec l'éditeur Emacs pour nous familiariser un peu plus avec cet éditeur. Ensuite, nous sommes passés à l'éditeur Visual Studio Code (VSC) pour différentes raisons: Sa rapidité, l'intégration Git parfaite, la gestion de projet, trouver la déclaration d'une fonction, d'une variable et la définition des structures qu'on peut bien aussi trouver dans d'autres IDE.

### 2.2 L'outil Git

Git est un système de contrôle de version open source conçu pour gérer tout, des petits aux grands projets avec rapidité et efficacité. Il nous a été utile pour les suivis des modifications apportées par chaque personne d'une manière simple et ordonnée. Ainsi de déposer nos fichiers dans un dépôt en ligne afin d'éviter toute perte de données.

### 2.3 Makefile

Nous avons utilisé l'outil d'aide à la compilation "make" pour la compilation séparée réalisée grâce à un makefile. Notre dépôt contient plusieurs fichiers sources .c et fichiers headers .h .

Les fichiers headers contiennent les structures, les constantes et les prototypes des fonctions définies dans les fichiers.c . La compilation séparée nous permettra de mieux organiser notre projet.

### 2.4 L<sup>A</sup>T<sub>E</sub>X

Ce rapport a été rédigé en LaTeX. Il nous permet de nous concentrer sur le contenu du document sans se soucier de la mise en forme qui sera effectuée automatiquement. Un éditeur L<sup>A</sup>T<sub>E</sub>X en ligne, Overleaf, nous a facilité l'utilisation et nous a permis la collaboration en temps réel.

Le rapport et tous les fichiers relatifs à son écriture sont déposés dans le dossier "report" sur le dépôt.

## 3 Base Version

La version base du projet est d'écrire un programme qui met en place le système du jeu, pendant lesquels des joueurs vont pouvoir placer des tuiles les uns à la suite des autres. Le programme se termine si l'un des joueurs a réussi à placer toutes ses tuiles, ou si tous les joueurs ont passé leur tour.

Pour celà, on devrait d'abord créer des tuiles et des jeux de tuiles, ensuite créer une structure de file pour gérer les jeux des joueurs et enfin mettre en place la boucle du jeu tout en assurant le respect des règles.

### 3.1 Les tuiles

Les tuiles du jeu sont des tuiles carrées divisées en quatre triangles, chacun est coloré d'une couleur particulière.

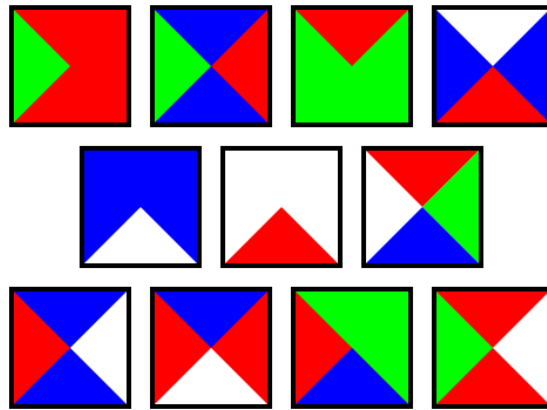


Figure 1: Tuiles de Wang (source: Wikipédia)

Nous définissons tout d'abord une première structure nommée *color* qui contient deux champs essentiels pour définir une couleur: le nom et l'ANSI-code<sup>1</sup>.

```
1 struct color{
2     const char *name;
3     const char *ANSI_code;
4 };
```

Puis nous définissons la structure d'une tuile nommée *tile* qui contient les 4 triangles de structure *color*, définis dans un tableau de taille 4:

```
1 struct tile{
2     struct color *color_direction[MAX_DIRECTION];
3 };
```

Nous avons choisi de définir ces 2 structures dans deux fichiers séparés : *struct\_tile.h* et *struct\_color.h*. Ce dernier n'était pas nécessaire puisqu'on pourrait accéder au pointeur de struct *color* via la fonction *color\_from\_name* seulement à partir du nom de la couleur.

Les tuiles sont arrangées dans des decks qui sont simplement la donnée d'un ensemble de paires (tuile  $\times$  quantité). Le deck est rempli via la fonction *deck\_init*.

<sup>1</sup>L'ANSI-code est un code qui est associé à chaque couleur et qui nous permettra ici d'afficher les caractères en couleur sur le terminal.

Le tableau complet des codes couleurs ANSI fonctionnant en C: <https://gist.github.com/RabaDabaDoba/145049536f815903c79944599c6f952a>

```

1 struct deck_pair {
2     const struct tile* t;
3     unsigned int n;
4 };
5
6 struct deck {
7     struct deck_pair cards[MAX_DECK_SIZE];
8     unsigned int size;
9 };

```

La structure `deck_pair` contient deux champs: la tuile et sa quantité.

La structure `deck` contient deux champs: une liste de structures de `deck_pair`, `cards`, de taille 100. Et `size`, le nombre de tuiles différentes.

## 3.2 La file

### 3.2.1 Structure de file

Cette structure va nous permettre de gérer les tuiles des joueurs. Une telle file devrait pouvoir accepter d'insérer des éléments à l'intérieur, de récupérer un pointeur vers l'élément en tête de la file ainsi que de retirer des éléments si la file n'est pas vide, tout cela en respectant l'ordre d'insertion.

Nous avons choisi de créer une telle structure en manipulant un tableau.

```

1 struct file{
2     int head;
3     int tail;
4     const struct tile *tiles [MAX_TILE+1];
5 };

```

Cette structure contient trois champs:

`head`: l'indice de la tête de la file.

`tail`: l'indice de la queue de la file.

Une liste `tiles` qui contient les adresses des tuiles.

`tiles[0]` est toujours égale à `NULL`, cette case nous permettra de savoir si la file est vide ou pas.

### 3.2.2 Les fonctions de la file

Afin de manipuler la file, on aura besoin d'un ensemble de fonctions :

```

1 void file_init(struct file *f);
2
3 int file_is_empty(struct file *f);
4
5 int file_size(struct file *f);
6
7 void file_push(struct file *f, const struct tile *t);
8
9 const struct tile* file_top(struct file *f);

```

```

10
11 void file_pop(struct file *f);

```

La fonction *file\_init* :

Cette fonction permet d'initialiser une file vide où l'indice de la tête et celle de la queue de la file est nulle. Cette fonction est de complexité, en espace et en temps, constante.

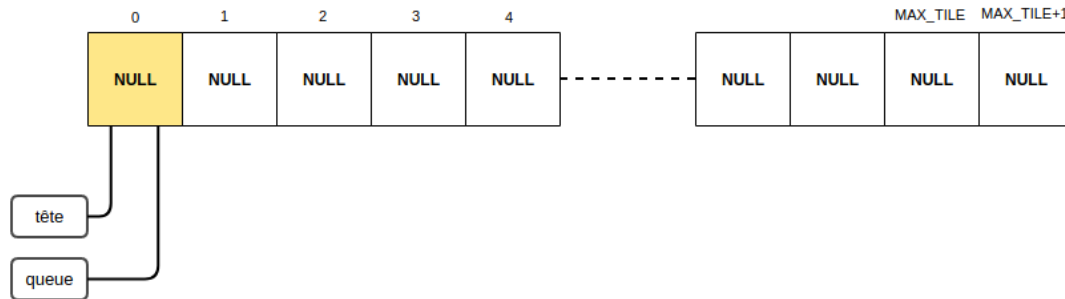


Figure 2: File vide

La fonction *file\_is\_empty* :

Cette fonction vérifie si la file est vide ou pas. C'est ici où apparaît l'intérêt de `tiles[0]` défini dans la structure de file. La file est vide que si l'indice de la tête de la file est nulle. Cette fonction est de complexité, en espace et en temps, constante.

La fonction *file\_size* :

Cette fonction retourne la taille de la file. Une file vide est de taille nulle.

Dans l'exemple suivant, la tête de la file est d'indice 1 et la queue est d'indice 3. La taille de la file est donc 3. Cette fonction est de complexité constante.

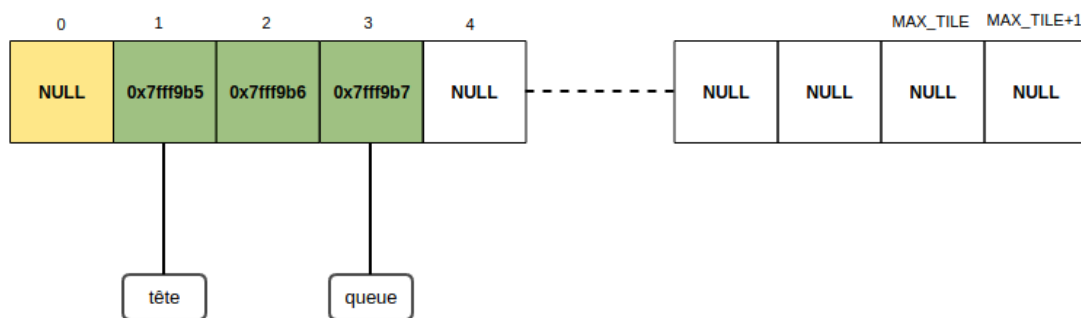


Figure 3: Une file de taille 3

Ici la file contient un seul élément donc elle est de taille 1.

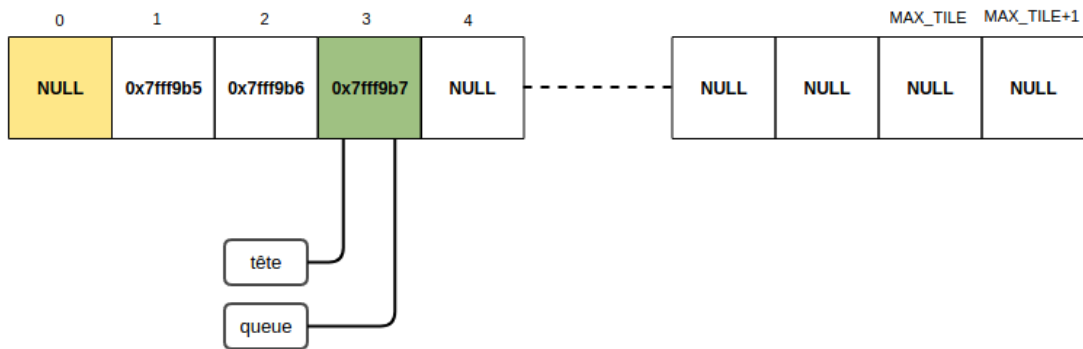


Figure 4: Une file de taille 1

Et ici, on représente une file de taille 5 dans le cas où l'indice de la tête est supérieur à celui de la queue.

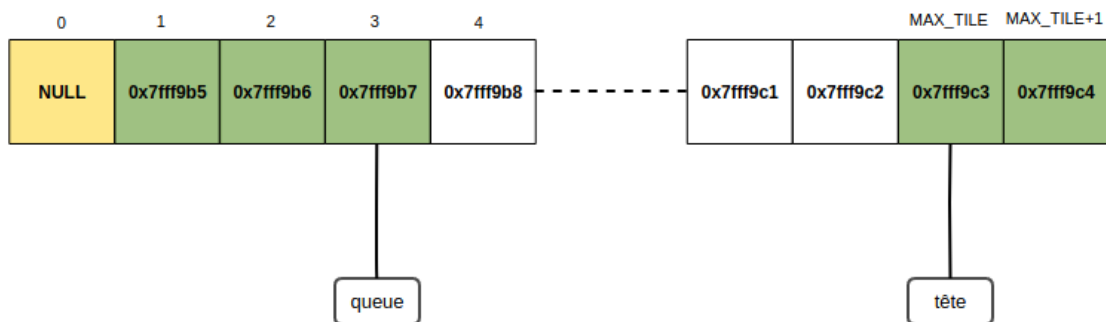


Figure 5: Une file de taille 5

La fonction ***file\_top*** :

Cette fonction retourne l'élément qui se trouve dans la tête de la file si cette dernière n'est pas vide. Cette fonction est de complexité constante.

La fonction ***file\_push*** :

Cette fonction permet d'insérer un élément, qui dans notre cas est un pointeur vers une structure de tuile, dans la file si elle n'est pas pleine. Cette fonction est de complexité constante.



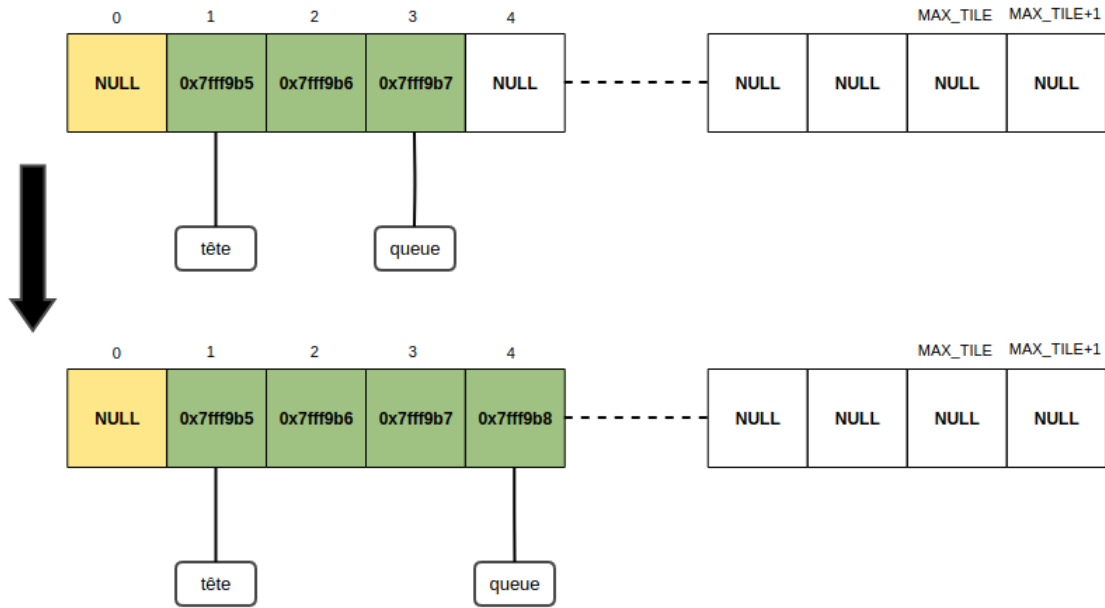


Figure 6: Une file avant et après l'insertion d'un élément

La fonction *file\_pop* :

Cette fonction permet de défiler un élément de la file si cette dernière n'est pas vide. Cette fonction est de complexité constante.

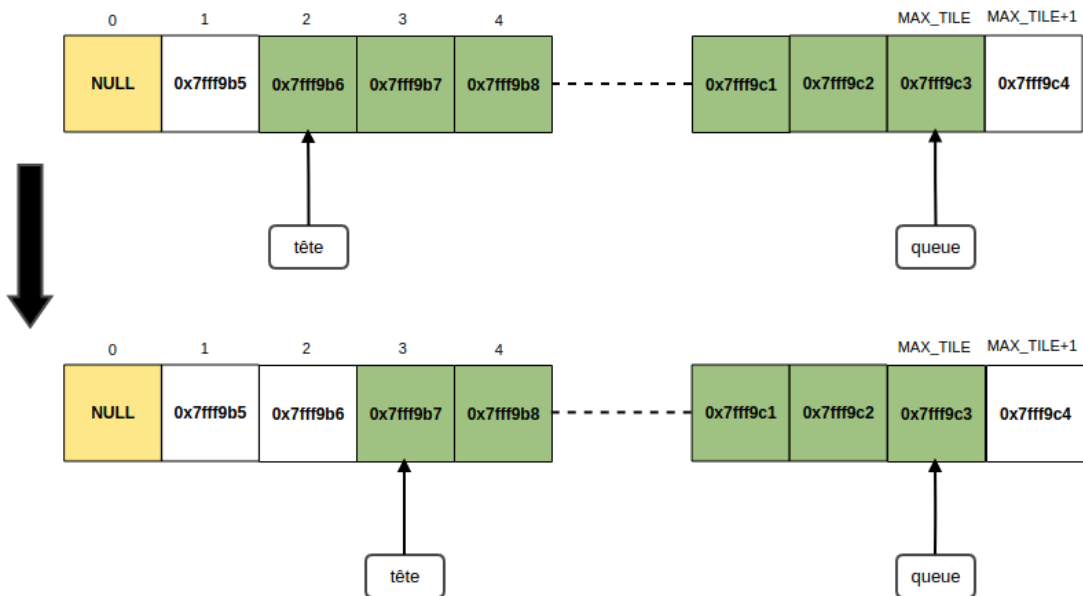


Figure 7: Une file avant et après l'enlèvement d'un élément

### 3.3 Le plateau du jeu

Le jeu se joue sur un plateau de taille maximale  $50 \times 50$ . Le joueur devrait pouvoir y déposer sa tuile tout en respectant les règles du jeu. L'implémentation d'une structure du plateau est donc nécessaire.

### 3.3.1 Structure du plateau

On a défini la structure du plateau, qu'on nomme *board*, comme suit:

```
1 struct board{
2     unsigned int dim;
3     const struct tile *tile_board[MAX_BOARD_SIZE][MAX_BOARD_SIZE];
4 };
```

Elle contient deux champs: la dimension du plateau et un tableau de pointeurs vers des structures de tuiles. Une implémentation d'une structure de coordonnées nous facilitera l'accès à une case de ce plateau. On la nomme *coord* et elle est défini comme suit:

```
1 struct coord{
2     unsigned int x;
3     unsigned int y;
4 };
```

Où x représente la ligne et y la colonne.

L'accès à la case (i,j) du plateau se fera par `tile_board[i][j]`

### 3.3.2 Les fonctions du plateau

Afin de manipuler le tableau du jeu, on aura besoin d'un ensemble de fonctions :

```
1 void board_init(struct board *b, int size);
2
3 int board_is_empty(struct board *b);
4
5 int can_place_tile(struct board *b, struct coord c);
6
7 int can_place_tile_rules(const struct tile *t, struct board *b, struct
    coord c);
8
9 void place_tile(const struct tile *t, struct board *b, struct coord c);
10
11 void show_board(struct board *b);
```

La fonction ***board\_init***:

Cette fonction permet de définir la dimension du plateau et de remplir ses cases par des tuiles vides. Donc de remplir le `tile_board` par l'adresse de la tuile vide `empty_tile`. Cette fonction a une complexité quadratique  $O(n^2)$  avec n la taille du plateau.

La fonction ***board\_is\_empty***:

Cette fonction nous permettra de savoir si le plateau est vide de tuiles ou pas. Cette fonction a une complexité quadratique  $O(n^2)$  avec n la taille du plateau.

La fonction ***place\_tile***:

Cette fonction permet de placer une tuile dans les coordonnées (i,j) du plateau. Et ce en chargeant `tile_board[i][j]` du tableau par l'adresse de la tuile. Cette opération

n'est valable que si la tuile de la case de l'emplacement souhaité est une tuile vide. On vérifie cela grâce à la fonction `can_place_tile`. Cette fonction a une complexité constante  $O(1)$ .

La fonction `can_place_tile`:

Cette fonction nous permet de vérifier si on peut placer une tuile dans une case du plateau ou pas. L'opération est valide que si la case est remplie d'une tuile vide. On n'évoque pas encore le respect des règles du jeu à ce stade, la fonction `can_place_tile_rules` s'en charge. Cette fonction est de complexité constante  $O(1)$ .

La fonction `can_place_tile_rules`:

Cette fonction nous permet de vérifier si une tuile donnée peut être placée dans une case du plateau tout en respectant les règles du jeu:

- 1- La case doit être vide (remplie d'une tuile vide)
- 2- L'ensemble des tuiles doit rester connexe
- 3- Les bords de deux tuiles attenantes doivent être de la même couleur.

Si les trois conditions sont respectées, la tuile pourra être posée dans la case du plateau et la fonction retourne 1, 0 sinon. Cette fonction est de complexité constante  $O(1)$ .

La fonction `show_board`:

Cette fonction permet d'afficher le plateau du jeu sur un terminal. Seuls les tuiles non vides sont représentées. La fonction parcourt tout le plateau affiche la tuile en représentant ses cotés en couleur.

Ceux-ci sont représentés par des caractères de triangles. Cette fonction a une complexité quadratique  $O(n^2)$  avec  $n$  la taille du plateau.

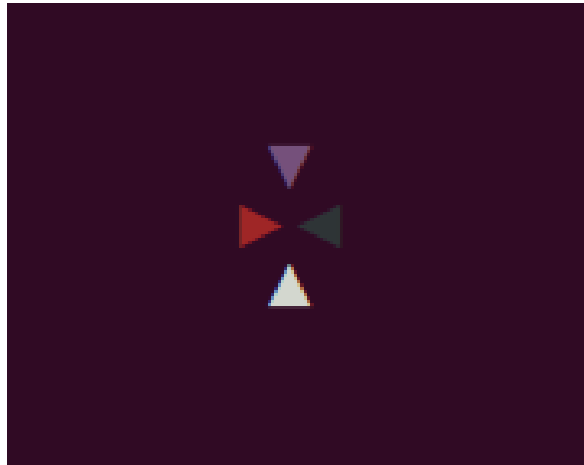


Figure 8: Affichage d'une tuile

L'affichage du plateau à la fin de chaque tour du jeu nous a été très utile pour vérifier si l'ensemble des joueurs respectent les règles du jeu et nous a permis de suivre l'avancement du jeu tour par tour.

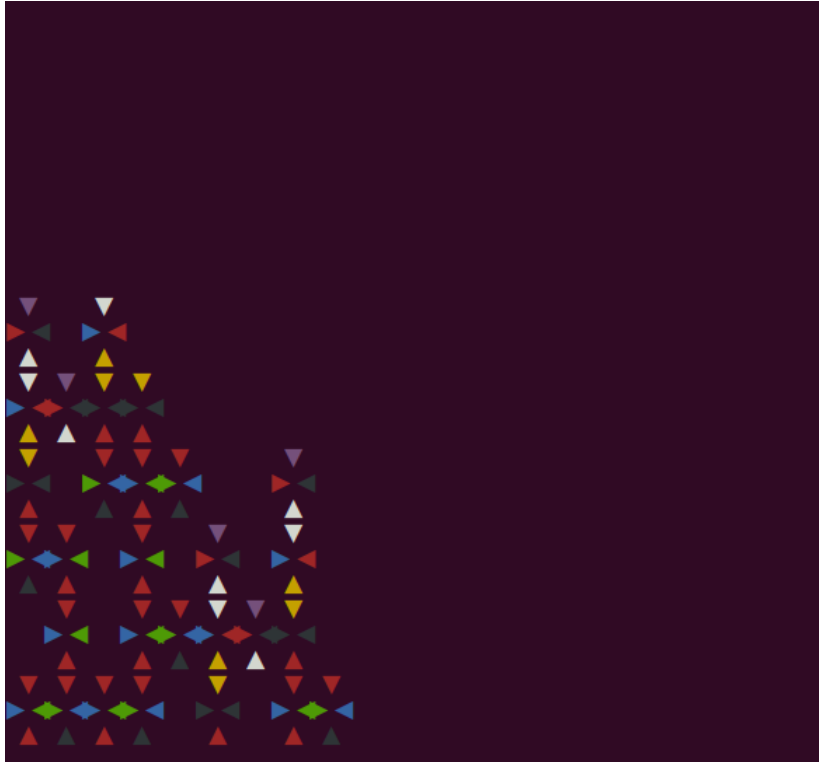


Figure 9: Affichage d'un plateau de tuiles de dimension 10

## 3.4 Les joueurs

### 3.4.1 Structure du joueur

Le nombre des joueurs est entre 2 et 20. Chaque joueur possède une pile de l'ensemble de ses tuiles, un score associé et un statut (actif/inactif). Le joueur est par défaut actif, il ne l'est plus que s'il enfreint l'une des règles du jeu. Dans ce cas, ses tuiles restent en jeu, mais son score en fin de sa partie est nul.

Ceci dit, nous définissons la structure du joueur, qu'on nomme *player* de la manière suivante :

```
1 struct player{
2     int active;
3     int score;
4     int hand_size;
5     struct file hand;
6 }
```

Nous définissons aussi une structure nommée *players* qui contient le nombre de joueurs au total, actifs et une liste de joueurs.

```
1 struct players{
2     int players_count;
3     int players_active;
4     struct player list_players[MAX_PLAYERS];
5 };
```

### 3.4.2 Les fonctions player

on a défini les fonctions associées suivantes:

```
1 void players_init(int size, struct players *p);
2
3 struct player *get_next_player(struct players *p, int turn);
```

La fonction ***players\_init***:

Cette fonction permet d'initialiser le nombre de joueurs, leurs scores et leurs decks de tuiles *hand* qu'ils possèdent. Cette fonction a une complexité linéaire  $O(n)$  avec  $n$  le nombre de joueurs.

La fonction ***get\_next\_player***:

Cette fonction retourne l'adresse du joueur actif à chaque tour du jeu en respectant l'ordre de la liste des joueurs. Cette fonction est de complexité constante  $O(1)$ .

## 3.5 L'aléatoire en C

Le jeu de tuiles devrait être mélangé aléatoirement, et distribué équitablement aux joueurs. On aura besoin donc de générer une suite de nombre pseudoaléatoires. Les fonctions, déclarées dans `stdlib.h`, qui le permettent sont:

```
1 int rand(void);
2
3 void srand(unsigned int seed);
```

La fonction ***rand***:

Cette fonction retourne un nombre aléatoire à chaque appel, compris entre 0 et `RAND_MAX`. On peut réduire l'intervalle des nombres retournés en utilisant l'opérateur modulo. Dans l'exemple suivant, `r` est compris entre 0 et `size-1`

```
1 int r = rand() % size;
```

La fonction ***srand***:

Cette fonction permet d'initialiser le générateur de nombres pseudoaléatoires (RNG<sup>2</sup>) qu'on appelle une graine ou *seed*. Elle est initialisée avec l'heure de l'exécution du programme par défaut sauf indication contraire par l'utilisateur lors de l'exécution du programme avec le paramètre optionnel `-s`. En effet, pour éviter de se retrouver avec la même suite de nombres à chaque exécution du programme, il faut modifier à chaque fois la graine, or l'heure est une valeur qui diffère à chaque appel du programme. En initialisant le RNG avec l'heure actuelle, on devrait obtenir une suite de nombres différentes à chaque exécution.

```
1 seed = time(NULL);
```

---

<sup>2</sup>random number generator (RNG) est un dispositif capable de produire une séquence de nombres pour lesquels il n'existe aucun lien déterministe entre un nombre et ses prédécesseurs, de façon que cette séquence puisse être appelée « suite de nombres aléatoires ». Source: Wikipedia

## 3.6 Le jeu de tuiles

### 3.6.1 Mélanger et distribuer

Après avoir initialisé le deck avec les tuiles du jeu, nous devons pourvoir mélanger ces tuiles et les distribuer aléatoirement et équitablement à l'ensemble des joueurs.

La structure de deck contient l'ensemble de couples (tile x quantité). Nous avons choisi d'abord de distribuer ces tuiles dans une liste de tuiles, puis mélanger aléatoirement les tuiles de cette liste, pour enfin les distribuer à l'ensemble des joueurs.

Le schéma suivant illustre les trois opérations:

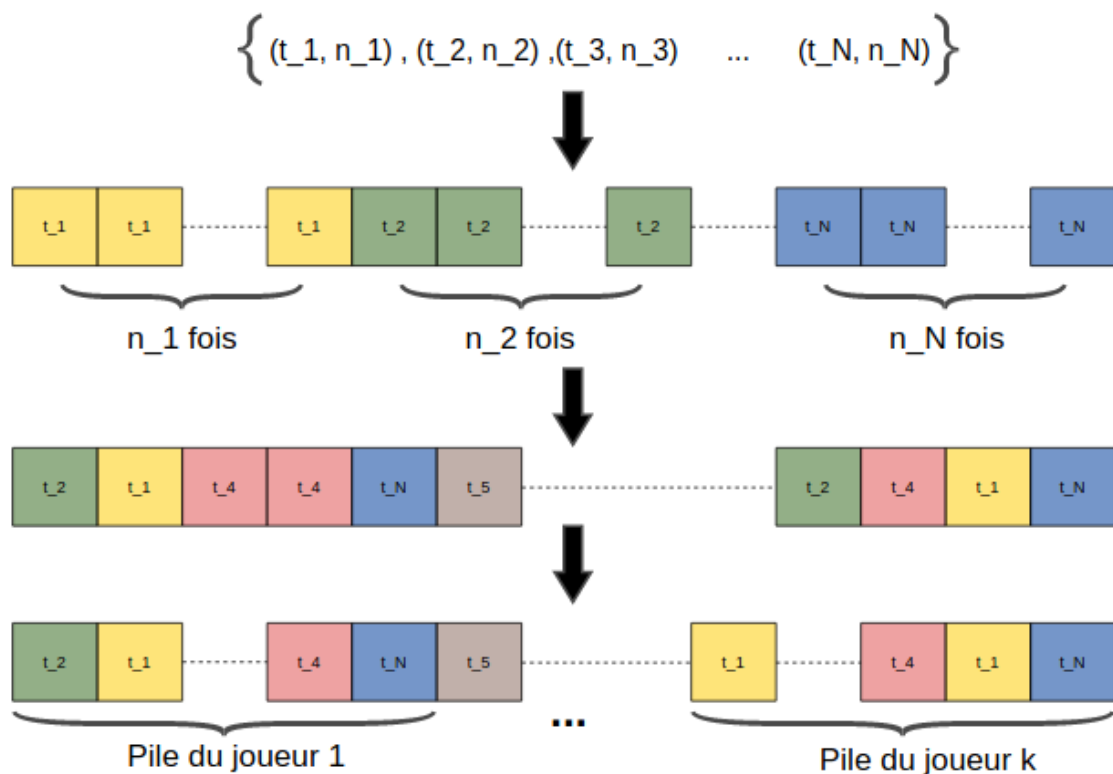


Figure 10: Schéma illustrant les 3 opérations

le couple (tile, quantité) est représenté par  $(t_i, n_i)$ , avec  $i$  allant de 1 jusqu'à  $N$ .  $N$  désigne le nombre de tuiles différentes qu'on possède dans le deck initial.

$k$  désigne le nombre de joueur total.

Les fonctions qui gèrent ces opérations sont définies comme suit:

```
1 int tiles_count(struct deck *d);
2
3 int hand_size(int t_c, int p_c);
4
5 void split_tiles_tolist(struct deck *d, const struct tile *list[]);
6
7 void mix_tiles(int tiles_count, const struct tile *list[]);
8
```

```

9 void split_tiles(int tiles_count, const struct tile *list[], struct
    players *p);
10
11 void split_deck(struct deck *d, struct players *p);

```

La fonction ***tiles\_count***:

Cette fonction retourne le nombre total de tuiles initialisées dans le deck. Cette fonction est de complexité linéaire  $O(n)$  avec  $n$  le nombre de tuiles différentes initialisées dans le deck.

La fonction ***hand\_size***:

Cette fonction retourne le nombre de tuiles que doit posséder chaque joueur après une distribution équitable des tuiles. Tous les joueurs devront posséder le même nombre de tuiles. Cette fonction est de complexité constante  $O(1)$ .

La fonction ***split\_tiles\_tolist***:

Cette fonction permet de remplir une liste par les adresses des tuiles qui se trouvent dans le deck initial.

La fonction ***mix\_tiles***:

Cette fonction permet de mélanger aléatoirement les éléments d'une liste. Chaque élément de la liste est échangé avec un autre élément, d'indice plus grand strictement, et choisi au hasard.

Cette fonction a une complexité linéaire  $O(n)$  avec  $n$  le nombre de tuiles dans la liste.

La fonction ***split\_tiles***:

Cette fonction permet de remplir équitablement la pile de chaque joueur à partir de la liste des tuiles mélangées définie précédemment.

La fonction ***split\_deck***:

Toutes les fonctions définies ci-dessus sont regroupées dans cette fonction.

```

1 void split_deck(struct deck *d, struct players *p){
2     int t_c = tiles_count(d);
3     split_tiles_tolist(d, list_tiles);
4     mix_tiles(t_c, list_tiles);
5     split_tiles(t_c, list_tiles, p);
6 }

```

### 3.6.2 Remplir et sélectionner

À chaque tour du jeu, une liste des positions jouables est remplie puis le programme sélectionne aléatoirement une position de cette liste. Pour cela, on a défini les fonctions suivantes:

```

1 int fill_authorized_places(const struct tile *t, struct board *b, struct
    coord list[], int turn);
2

```

```
3 struct coord select_position(const struct tile *t, struct board *b, struct
    coord list [], int turn);
```

### 3.7 Fonction de choix de paramètres de l'entrée

Notre exécutable exécutera une partie entière de jeu en prenant comme paramètres optionnels:

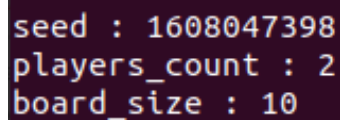
- n : Pour indiquer le nombre de joueur.
- b : Pour indiquer la taille du plateau.
- s : Pour initialiser le générateur aléatoire.

La fonction *parse\_opts* se charge de modifier les valeurs des variables globales: *seed*, *players\_count*, et *board\_size*.

```
1 void parse_opts(int argc, char* argv[]);
```

### 3.8 Conception et déroulement du jeu

Par défaut, le RNG est initialisé aléatoirement, le nombre de joueurs et la taille du plateau sont initialisés respectivement à 2 et  $10 \times 10$ , sauf indications contraires par l'utilisateur lors de l'exécution du programme.



```
seed : 1608047398
players_count : 2
board_size : 10
```

Figure 11: Exemple d'affichage de paramètres pour un jeu standard

Tout d'abord, on initialise le plateau du jeu, le deck des tuiles et les joueurs, on distribue les tuiles aux joueurs. On initialise également le nombre de tour *turn* et le nombre de tours passés *skipped\_turn*. On utilise une boucle while qui a pour condition: *skipped\_turns* != *p.players\_active*. Et donc tant que tous les joueurs n'ont pas passé leurs tours, on sélectionne un joueur par ordre sur la liste des joueurs *list\_players*, on sélectionne la tuile que possède le joueur en tête de sa pile, pour ensuite déterminer la liste des places autorisées et jouables. *list\_authorized\_places*. Si la liste est vide, le joueur ne peut pas poser de tuiles et son tour est passé. Sinon le programme en sélectionne au hasard des coordonnées et le joueur y place sa tuile et gagne 1 point en score. On peut suivre le déroulement du jeu en affichant à la fin de chaque boucle le plateau du jeu.

Le jeu se termine en affichant "GAME OVER" et les scores des joueurs par l'appel de la fonction *display\_results*:

```
1 void display_results(struct players *p);
```

Celui qui possède le score le plus élevé remporte la partie.



-----RESULTS-----	
PLAYER	SCORE
Player 1	91
Player 2	110
Player 3	50
Player 4	70
Player 5	30
Player 2 wins !	

Figure 12: Affichage des scores des joueurs à la fin d'une partie

### 3.9 Organisation et dépendances des fichiers

Notre code est organisé selon plusieurs fichiers .c et .h.

Les fichiers .h contiennent des structures et les prototypes des fonctions définies dans les fichiers .c . On a choisi de définir les structures color et tile dans deux fichiers : struct\_color.h et struct\_tile.h pour les inclure dans notre fichier test: test\_main.c .

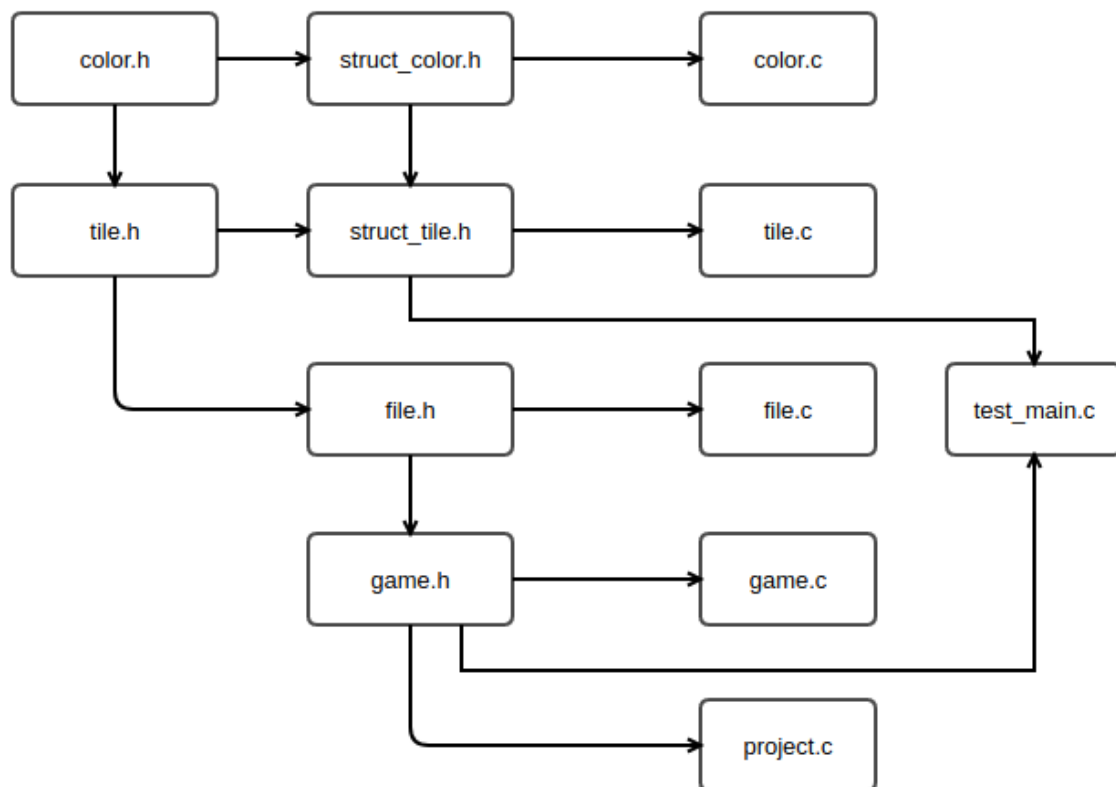


Figure 13: Organisation initiale des fichiers

## 4 Achievement 1

Dans cette version du projet, les tuiles ont désormais un propriétaire, qui correspond au joueur ayant posé la tuile. Les règles du jeu définissent aussi un ensemble de motifs,

rapportant chacun un certain nombre de points.

## 4.1 Changements et ajouts effectués

Pour définir le propriétaire d’une tuile et associer chaque tuile à son possesseur, deux choix s’offrent à nous. Ou bien définir une nouvelle structure qui se compose de deux champs, la tuile en question et la structure du joueur propriétaire. Ou bien ajouter à la structure de joueur deux champs: une liste des coordonnées des tuiles centrales posées par le joueur, et un entier qui décrit le nombre de tuiles centrales possédées par le joueur. Nous avons choisi la seconde option. La structure du joueur est donc modifiée et on a défini de nouvelles structures et fonctions associées aux motifs.

### 4.1.1 Structure du joueur

La nouvelle structure du joueur est définie comme suit :

```
1 struct player{
2     int active;
3     int score;
4     int hand_size;
5     struct file hand;
6     int central_tiles_owns;
7     struct coord *coord_central_tiles[MAX_TILE];
8 };
```

On a ajouté à la définition précédente de struct player deux champs supplémentaires:  
central\_tiles\_owns: Le nombre de tuiles centrales que le joueur possède.  
coord\_central\_tiles: Une liste de coordonnées des tuiles centrales que le joueur possède.

### 4.1.2 Structure des motifs et fonctions associées

Nous définissons la structure des motifs, qu’on nomme *pattern*, de la manière suivante :

```
1 struct pattern{
2     int size;
3     int score;
4     const struct tile *pattern_tiles[5][5];
5 };
```

Elle contient trois champs: la taille du motif, le score associé, et une matrice de taille  $5 \times 5$  d’adresses vers les tuiles qui définissent le motif. pattern\_tiles[2][2] contient l’adresse de la tuile centrale.

Nous définissons également une structure, qu’on nomme *patterns*, qui contient le nombre des motifs définis et une liste de motifs.

```
1 struct patterns{
2     int number_patterns;
3     struct pattern list_patterns[MAX_PATTERN];
4 };
```

Les fonctions associées aux motifs sont les suivantes:

```

1 void patterns_nil(struct patterns *pat);
2
3 void patterns_init(struct patterns *p);
4
5 int is_central_tile(const struct tile *t, struct patterns *pats);
6
7 int num_pattern_board(struct board *b, struct patterns *pats );
8
9 void update_tile_owner(struct player *p, struct coord c, int size);
10
11 int is_pattern(const struct tile *t, struct coord c, struct board *b,
12               struct pattern *pat);
13
14 int coord_equal(struct coord c1, struct coord c2);
15
16 int index_owner(struct coord c, struct players *p);
17
18 void attribute_score_pattern(struct board *b, struct players *p, struct
19                             patterns *pats);

```

La fonction ***patterns\_nil***:

Cette fonction permet d'initialiser les tuiles associées à chaque motif à des tuiles vides. Elle a une complexité quadratique  $O(n^2)$  avec  $n = 2 \times \text{taille\_motif} + 1$ .

La fonction ***patterns\_init***:

Cette fonction permet d'initialiser et de définir les motifs mis en jeu.

On a défini 4 motifs du jeu de la façon suivante:

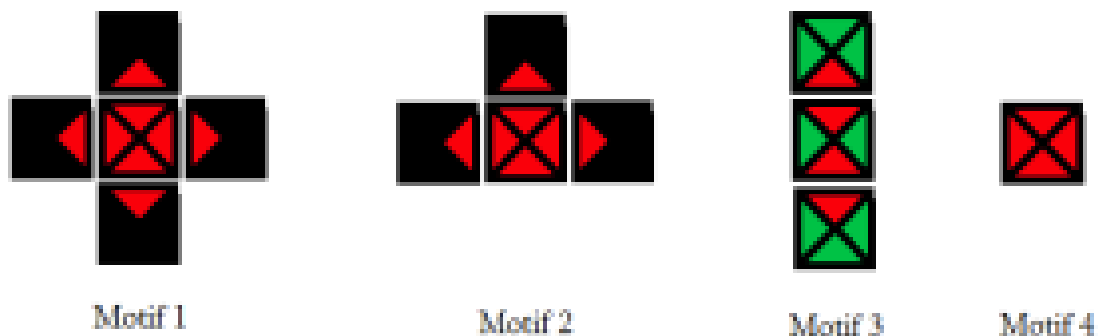


Figure 14: Exemple de motifs

La fonction ***is\_central\_tile***:

Cette fonction prend en paramètre une tuile et la structure *patterns* et permet de vérifier si la tuile est une tuile centrale d'un motif. Elle retourne 1 si c'est le cas, 0 sinon. Elle a une complexité linéaire  $O(n)$  avec  $n$  le nombre des motifs définis.

La fonction ***num\_pattern\_board***:

Cette fonction prend en paramètre le plateau du jeu et les motifs, et elle renvoie le nombre de motifs présents dans le plateau. Elle a une complexité quadratique  $O(n^2)$  avec  $n$  la taille du plateau.

La fonction ***update\_tile\_owner***:

Cette fonction prend en paramètres un joueur, les coordonnées de la tuile posée, et le nombre de tuiles centrales que le joueur possède. Elle met à jour la structure du joueur en ajoutant les coordonnées passées en paramètre à la liste *coord\_central\_tiles* du joueur.

La fonction ***is\_pattern***:

Cette fonction prend en paramètres une tuile, ses coordonnées, le plateau du jeu, et un motif défini. Elle vérifie si la tuile complète le motif. Pour cela, on vérifie si la tuile est une tuile centrale d'un motif, et si c'est le cas on compare les tuiles posées à ses alentours dans le tableau aux tuiles qui définissent le motif associé. Si ceci est vérifié la fonction renvoie 1, et 0 sinon. La fonction a une complexité quadratique  $O(n^2)$  avec  $n = 2 \times \text{taille\_motif} + 1$ .

La fonction ***coord\_equal***:

Cette fonction permet de vérifier si les deux coordonnées sont égales. Cette fonction est de complexité constante  $O(1)$ .

La fonction ***index\_owner***:

Cette fonction prend en paramètres les coordonnées d'une tuile centrale, et la structure des joueurs. Elle renvoie l'indice du propriétaire de la tuile. Si la tuile n'est pas centrale, la fonction renvoie -1.

La fonction ***attribute\_score\_pattern***:

Cette fonction est appelée à la fin du jeu. Elle prend en paramètres le plateau du jeu, les joueurs, et les motifs. La fonction parcourt tout le tableau et pour chaque motif trouvé, son score associé est attribué au joueur propriétaire de la tuile centrale associée à la motif.

## 4.2 Conception et déroulement du jeu

Nous ajoutons aux initialisations précédentes, l'initialisation des motifs.

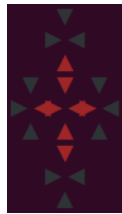


Figure 15: Exemple d'un motif du jeu

Le jeu suit le même principe défini précédemment jusqu'à l'instant où un joueur pose une tuile. À ce stade, on vérifie si la tuile est une tuile centrale par la fonction

*is\_central\_tile*. Si c'est le cas, ses coordonnées sont ajoutées à la liste *coord\_central\_tiles* via la fonction *update\_tile\_owner*. Une fois le jeu terminé, on fait appel à la fonction *attribute\_score\_pattern* qui affiche le nombre de motifs trouvés sur le plateau et ajoute le score associé à chaque motif au score du joueur propriétaire. On affiche ensuite les scores des joueurs par l'appel de la fonction *display\_results*.

```

-----GAME OVER-----
16 PATTERNS FOUND !
PATTERN[3] FOUND : Player 1 wins 20 points
PATTERN[3] FOUND : Player 2 wins 20 points
PATTERN[3] FOUND : Player 1 wins 20 points
PATTERN[3] FOUND : Player 2 wins 20 points
PATTERN[3] FOUND : Player 2 wins 20 points
PATTERN[3] FOUND : Player 1 wins 20 points
PATTERN[3] FOUND : Player 1 wins 20 points
PATTERN[3] FOUND : Player 2 wins 20 points
PATTERN[3] FOUND : Player 1 wins 20 points
PATTERN[3] FOUND : Player 2 wins 20 points
PATTERN[3] FOUND : Player 1 wins 20 points
PATTERN[3] FOUND : Player 1 wins 20 points
PATTERN[3] FOUND : Player 1 wins 20 points
PATTERN[3] FOUND : Player 1 wins 20 points
PATTERN[3] FOUND : Player 2 wins 20 points
PATTERN[0] FOUND : Player 2 wins 34 points
PATTERN[1] FOUND : Player 2 wins 70 points
PATTERN[3] FOUND : Player 2 wins 20 points

-----RESULTS-----

      PLAYER | SCORE
-----
      Player 1      208
      Player 2      271
Player 2 wins !

```

Figure 16: Affichage des motifs trouvés et les scores à la fin d'une partie

### 4.3 Organisation et dépendances de fichiers

Nous avons choisi d'ajouter deux fichiers `pattern.[ch]` sur lesquels nous avons défini les structures des motifs et les fonctions associées.

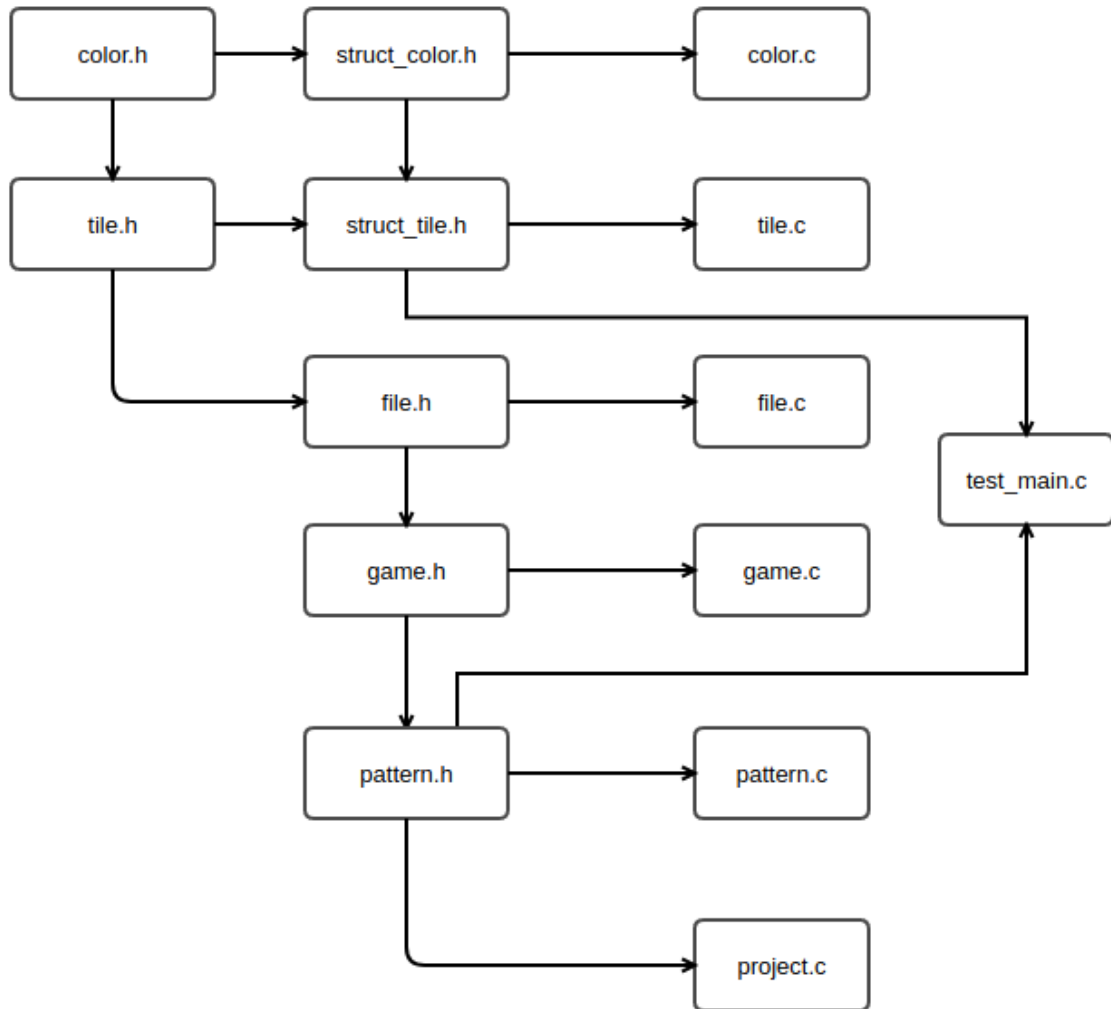


Figure 17: Organisation des fichiers

## 5 Tests et vérifications des fonctions

Afin de nous assurer du bon fonctionnement des fonctions implémentées, nous avons créé un fichier test *test\_main.c* qui contient les différents tests des fonctions définies. Pour la fonction *tile\_equals* par exemple, nous y avons effectué deux tests.

```

1 if (tile_equals(&t1,&t1) == 1)
2     printf("tile_equals: PASSED (1/2)\n");
3 else
4     printf("tile_equals: FAILED in line %d\n",__LINE__);
5 if (tile_equals(&t1,&t2) == 0)
6     printf("tile_equals: PASSED (2/2)\n");
7 else
8     printf("tile_equals: FAILED in line %d\n",__LINE__);
  
```

Si le test est passé, le nom de la fonction suivi d'un "PASSED" est affiché sur l'écran, sinon on affiche "FAILED" suivi de la ligne d'erreur.

Pour vérifier si les tuiles sont distribuées aléatoirement aux joueurs, nous avons décidé

d'afficher la pile des différents joueurs avant et après cette opération. Et pour les tests des motifs, nous avons placé 3 différents motifs dans un plateau de jeu pour ensuite vérifier le bon fonctionnement des fonctions associées.

## 6 Améliorations possibles

Dans notre version du projet, les joueurs respectent tout le temps les règles du jeu et ne pose leur tuile que si elle vérifie les deux conditions définies initialement. Nous avons pensé à ce que ça ne serait pas toujours le cas, en ajoutant un paramètre optionnel `-p` : la probabilité d'erreur. Sa valeur est comprise entre 0 et 10. Si sa valeur est `i`, alors les joueurs ont une probabilité de  $i/10$  pour ne pas respecter les règles. On peut donc initialiser cette valeur à 0, sauf indication contraire de l'utilisateur.

La fonction `fill_authorized_places` remplit la liste des places autorisées et jouables en utilisant soit la fonction `can_place_tile` soit la fonction `can_place_tile_rules` selon la valeur de `r`. Avec `r = 0` si les joueurs devront respecter les règles, et `r = 1` sinon.

```
1 int r = rand() % (p+1)
```

## 7 Conclusion

Dans le cadre des projets de programmation, nous avons donc réalisé un programme du jeu Tilings. Ce projet nous a permis de mettre en oeuvre les connaissances acquises dans le cours de programmation impérative et de les améliorer. Nous avons été confrontés à de nombreux problèmes mais nous avons pu trouver des solutions alternatives pour les résoudre. Aussi, nos professeurs encadrants nous ont donné de précieux conseils et nous les en remercions.