

Projet Infor S5

Sandwich

**KHIARI Skander
MEDGHALI Majid**



Departement Informatique
ENSEIRB-MATMECA
2021/2022

Table des matières

Introduction	2
1 Implémentation de l'image	3
1.1 création et affichage du monde	3
1.2 voisinage d'une cellule	3
2 implémentation des règles	3
2.1 définition des strcutures	3
2.2 initialisation des règles	5
2.3 test des règles	5
3 mise en place de l'algorithme de calcul d'itération	6
3.1 implementation de la file	6
3.2 architecture du projet	7
3.3 Description des étapes d'une itération	7
conclusion	9

Introduction

fezvdfc

1 Implémentation de l'image

1.1 création et affichage du monde

Pour commencer, la première étape était d'initialiser une image qu'on fera évoluer par la suite. Cette image est représentée par la structure monde (struct world) qui est préalablement définie dans fichier world.h. Pour l'initialiser on a eu recourt à la fonction world-init dans le fichier principal qui :

- Pour les achievement 0 et 1 : initialise aleatoirement une struct world avec seulement du blanc et du noir grâce à la fonction rand
- Pour l'achievement2 : implemente l'image initial du sable

l'affichage de l'image se fait grâce à la fonction afficher-image en respectant les règles de données de sortie

1.2 voisinage d'une cellule

Une fois le monde initialisée, l'étape suivante est d'arriver à recuperer les voisins d'une case donnée.

Pour faire cela, on a décidé de lister les voisins à partir d'une représentation à deux dimensions du monde. Cette représentation nécessite donc de créer une matrice de hauteur HEIGHT et de largeur WIDTH et de la remplir par la suite à partir des éléments de la structure world. Ceci est rendu possible grâce à la fonction remplir-image1.

Une fois la matrice créée, on peut récupérer facilement la voisins d'une cellule donnée en jouant sur les indices de cette matrice. Par exemple, le voisin à droite d'une case de coordonnées (i,j) dans une matrice "image" est $\text{image}[i\% \text{HEIGHT}][(j+1)\% \text{WIDTH}]$ (Les modulus permettent de prendre en compte les cases sur le bord de l'image).

Voici un test qui affiche les voisins de la case centrale pour une image 3*3 :

```
1 2 3
4 5 6
7 8 9
voisin (0,0): 1
voisin (0,1): 2
voisin (0,2): 3
voisin (1,0): 6
voisin (1,1): 9
voisin (1,2): 8
voisin (2,0): 7
voisin (2,1): 4
voisin (2,2): 1
```

2 implémentation des règles

L'image étant initialisée et que pour une case donnée on a tous ses voisins, la question maintenant est de savoir si cette case vérifie ou non une règle donnée. Pour cela on a commencé par définir la structure "struct rule" .

2.1 définition des structures

la définition de la structure "struct rule" a changé pendant chaque achievement :

- Achiev0 :

```
struct rule
{
    int couleur;
    int couleur-voisin;
    int couleur-new;
    int regle[8];
};
```

Pour savoir si une règle est vérifiée ou non on a codé la fonction rule-match qui retourne :

- 2 si la couleur de la case passée en paramètre est différente de l'entier "couleur" de la règle
- sinon : on détermine pour une case donnée le tableau des voisins et on compte le nombre de voisins ayant la couleur "couleur-voisin" et on retourne :
 - 1 si ce nombre est dans le tableau d'entiers "regle" c-a-d la règle est vérifiée
 - 0

- Achiev1 :

```
struct rule
{
    int couleur[3];
    int couleur-voisin[3];
    int couleur-new[3];
    int regle[8];
};
```

cette structure est équivalente à la précédente mais autorise plusieurs transformations possibles pour une règle donnée. Par exemple, dans notre projet : une cellule morte correspond à la couleur noir et une cellule vivante correspond à une couleur du triplet "blanc, rouge, bleu".

- Achiev2 :

Dans l'achiev2, on a décidé de changer complètement struct rule :

```
struct rule
{
    int dx;
    int dy;
    int couleur;
    int voisin[4];
};
```

Dans la fonction rule-match on a distingué deux cas de figures pour les règles :

- les règles pour lesquelles dy=0 (y correspond à l'axe horizontale) pour représenter les règles correspondant aux cases qui descendent verticalement (le sable qui tombe). Ainsi si la couleur de la case est égale à la couleur de la règle est que la case juste en dessous est noire alors rule-match retourne 1.
- les autres règles pour lesquelles dy=-1 ou dy=1 pour représenter les règles correspondant aux cases qui descendent le long d'une pente. Ainsi, on détermine les 4 voisins qu'on veut examiner et on les compare au tableau "voisin". S'ils sont égaux alors on retourne 1

si aucune de ces conditions ne retourne 1 alors on retourne 0

2.2 initialisation des règles

achiev0 et achiev1

les règles du jeu de la vie ont été initialisées dans le fichier rule.c de la manière suivante en utilisant un tableau de structures du type struct rule : une pour les cellules vivantes et l'autre pour les cellules mortes.

achiev0 :

```
struct rule rule[2]={{ {NOIR, BLANC, BLANC, {3,3,3,3,3,3,3}}, {BLANC, BLANC, NOIR, {0,1,4,5,6,7,8}}}};
```

achiev1 :

```
struct rule rule[2]={{ {NOIR, NOIR, NOIR}, {ROUGE, BLEU, BLANC}, {ROUGE, BLEU, BLANC}}, {{3,3,3,3,3,3,3}, { {ROUGE, BLEU, BLANC}, {ROUGE, BLEU, BLANC}, {NOIR, NOIR, NOIR}, {0,1,4,5,6,7,8}}}};
```

achiev2

De même, les règles du sable ont été initialisées grâce à un tableau de structures :

```
struct rule rule[3]={{ {1,0,JAUNE, {0,0,0,0}}, {1,-1,JAUNE, {JAUNE, JAUNE, JAUNE, NOIR}}, {1,1,JAUNE, {NOIR, JAUNE, JAUNE, JAUNE}}}};
```

- règle1 : si la case est jaune et que celle juste en dessous est noire alors elle descend
- règle2 : si la case est jaune et que son voisin droit est jaune et que ses voisins d'en dessous de droite à gauche sont respectivement jaune jaune et noir alors elle descend et se décale à gauche.
- règle3 : si la case est jaune et que son voisin gauche est jaune et que ses voisins d'en dessous de gauche à droite sont respectivement jaune jaune et noir alors elle descend et se décale à droite.

Remarque : pour les 3 premières règles le tableau voisins ne sert pas pour la vérification de la règle et donc a été initialisé à 0.

2.3 test des règles

Achiev 0 et 1

```
-----test_des_règles-----
Pour l'achievement 0 et 1 on prend les règles du jeu de la vie
3 3
#
0 16777215 0
16777215 0 0
0 0 16777215

on prend la case (1,1), qui est noire, et on applique la règle qui correspond aux cases NOIRES
rule_match retourne 1
c-a-d la règle est vérifiée car la case noire est entourée de 3 cases blanches donc elle devient blanche:
-----

on s'intéresse ici à la deuxième règle qui correspond aux cases BLANCHES:
-on prend la case(0,1), qui est blanche
rule_match retourne 0
c-a-d la couleur de la case correspond à la couleur de la règle mais la règle n'est pas vérifiée, car cette case ne compte que 2 voisins blancs, et
donc on ne fait rien

-on prend la case(1,1), qui est noire, de la première image
rule_match retourne 2
c-a-d la couleur de la case ne correspond pas à la couleur de la règle et donc la règle ne s'applique pas et donc on ne fait rien
```

Achiev 2

```
--test_des_règles-----
Pour l'achievement 2 on prend les règles du sable
3 3
#
0 0 65535
65535 65535 0
65535 65535 0

on choisit la case (0,2), qui est jaune, et on applique la règle qui correspond aux cases jaunes avec une case noire en dessous
rule_match retourne 1
c-a-d la règle est vérifiée car la case en dessous de cette case est noire donc cette case doit avancer
-----

on s'intéresse ici aux cases noires:
on choisit la case(1,2) et on applique pour n'importe quelle règle
rule_match retourne 0
c-a-d cette case n'avance pas

-----

on s'intéresse ici aux cases jaunes avec 3 voisins jaunes: 2 en dessous et un à gauche
on choisit la case(1,1) et la règle qui correspond à cette situation
rule_match retourne 1
c-a-d la règle correspondant à cette situation est vérifiée est donc cette case va se déplacer en bas et à droite
```

3 mise en place de l'algorithme de calcul d'itération

3.1 implementation de la file

Une fois qu'on a implémenté les règles, il faut maintenant stocker les changements quelque part. Pour les achievements 0 et 1, on a utilisé une file pour représenter l'image après transformation. Pour cela on a opté pour une file chaînée avec le dernier élément qui pointe vers NULL. Cette file est représentée grâce à ces deux structures :

```
struct element
{
    int val;
    struct element* suivant;
};

struct file
{
    struct element* premier;
}
```

la première structure représente un élément de cette file avec "val" la valeur de cet élément et "suivant" un pointeur vers l'élément suivant

la deuxième structure contient un pointeur vers le premier élément de cette file

Nous avons implémenté deux fonctions de base permettant de manipuler cette file à savoir : "enfiler" et "defiler".

Remarque : dans la fonction enfileur on a utilisé malloc pour construire le nouveau élément qu'on va enfileur

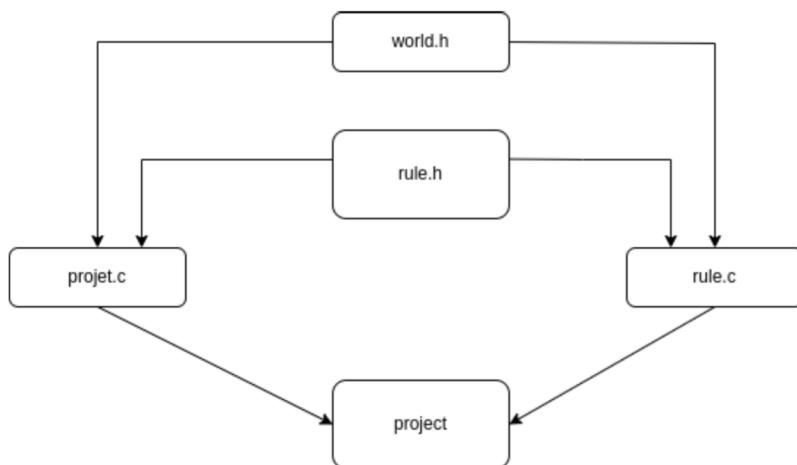
test de la file

```
-----test_de_la_file--  
on enfile 2 et 3 et 5 on obtient.  
etat de la file: 2 3 5  
si on defile  
etat de la file: 3 5  
si on enfile 8  
etat de la file: 3 5 8
```

3.2 architecture du projet

Dans ce projet nous avons utilisé deux fichiers .c :

- le premier est rule.c qui nous servait pour coder les fonctions de rule.h.
- le deuxième est projet.c qui contient le main et d'autres fonctions et structures comme les structures de la file et ses fonctions et les fonctions comme queue-append ou world-apply.
Neanmoins nous aurions pu mettre ces fonctions dans dans fchiers tel que file.c et file.h pour ne garder que le main dans projet.c .



3.3 Description des étapes d'une itération

achiev0 et 1

1. Après initialisation du monde, on commence par le parcourir ligne par ligne de haut en bas, colonne par colonne de gauche à droite.
2. Pour une case donnée, on teste pour chaque règle si elle est vérifiée ou non : chaque règle est récupérée grâce à la fonction rule-get qui nous retourne un pointeur vers celle-ci.

Dans la fonction "queue-append" l'entier "a" nous retourne :

- 1 si cette case vérifie la règle "rule" passée en paramètre, et si c'est le cas on enfile dans la file f la nouvelle couleur "couleur-new" obtenue grâce à la fonction rule-change-to.
- 0 si cette case la même couleur que l'entier "couleur" de la règle mais ne vérifie pas la règle, et dans ce cas on enfile dans la file f la la couleur de la case sans transformation.
- 2 si cette case n'a pas la même couleur de l'entier "couleur" de la règle et dans ce cas on fait rien.

3. Après passage par toutes les cases on obtient une file qui contient les changements à effectuer sur le monde. On applique alors ce changement grâce à la fonction world-apply.

achiev2

Pour l'achievement 2, on a rencontré quelques problèmes à travailler avec une file. En effet, pour une case Jaune entourée que par des cases noires rule-match va retourner 1 pour la première règle et 0 pour les 2 autres, et donc on va enfiler plusieurs fois des couleurs pour la même case. On a décidé alors d'effectuer quelques changements :

1. On a décidé de ne plus procéder avec une file pour stocker les changements mais avec une structure world nommée world1
2. pour une case donnée on ne stocke pas sa nouvelle couleur mais sa nouvelle position si elle va se déplacer ou sinon sa position si elle ne bouge pas.
3. enfin, on a défini un compteur pour que si $a=1$ ($c-a-d$ une règle est vérifiée) on ne test plus pour les autres règles.

conclusion