Rapport de projets Informatique S5 Sandwich

Rapport complet

Auteurs : Marine Vovard
Mathieu Dupoux

Responsable de projet : David Renault
Enseignant référent : Michaël Clément

7 janvier 2022

Table des matières

1 | Vue globale du projet

1.1 Énoncé et objectif du sujet

Ce rapport repose sur l'implémentation d'un automate cellulaire - une grille régulière de cellules contenant chacune un état choisi parmi un ensemble fini et qui peut évoluer au cours du temps [1] - en langage C. L'un des premiers objectifs a été d'implémenter le jeu de la vie [2] de John Cownway. La suite du projet nous a conduit a ajouter de nouvelles fonctionnalités ainsi qu'à optimiser les performances de notre programme initial. L'énoncé nous a guidé pour répartir ce projet en trois sections distinctes : l'implémentation des règles, celle du monde et de la file. Tous ces aspects ont alors été regroupés dans un fichier principal et des tests ont été effectué afin de vérifier le bon fonctionnement de notre algorithme. En ajoutant des fonctionnalités telles que le déplacement d'une cellule, le changement aléatoire d'état et la gestion des conflits, nous avons finalement généré deux exemples d'utilisation de notre algorithme : le jeu de la vie en couleur et la chute d'un tas de sable.

1.2 Stratégie globale d'implémentation

Le projet a débuté avec quelques règles simples :

1. Les deux fichiers d'en-tête concernant les règles et le monde donnés par le sujet ne peuvent pas être modifiés ;
2. Les données de sortie doivent être compatible avec la librairie `SDL` ;
3. Des options peuvent être ajoutées à l'exécutable pour laisser la possibilité à l'utilisateur de modifier la taille du monde et sa génération aléatoire.
4. L'utilisation de la compilation séparée et l'utilisation d'un `Makefile` est obligatoire.

Suite à ces exigences, nous nous sommes appuyés sur trois structures principales : le monde qui est une image 2D remplie de pixels, les règles qui déterminent le comportement de l'automate et la file qui stocke les changements à appliquer sur le monde une fois toutes les cellules étudiées.

Le programme principal fonctionne de la manière suivante :
— Dans un premier temps, nous créons un monde soit aléatoire, soit définit dans le code ;
— On ajoute les règles créées dans le fichier `rule.c` et stockées dans un tableau présent dans ce même fichier ;
— Pour chaque image à générer :
  — On parcourt toutes les cellules et on regarde si elles sont compatibles avec une règle. Si c'est le cas, on choisit quel changement effectuer (un changement d'état ou un déplacement) ;
  — On ajoute ces changements à une file ;
  — On défile tous les changements et on les applique sur le monde en résolvant les conflits possibles lors des déplacements d'une cellule. Le choix du gagnant est fait aléatoirement.

Les tests ont été effectués de manière à ce que chaque grande partie de notre algorithme soit vérifiée. Il existe donc des tests qui vérifient l'implémentation du monde et de ses fonctions, l'implémentation des règles et ses applications, et l'implémentation des files. Finalement, un test a été ajouté à la suite d'un ajout d'une fonctionnalité du programme : la gestion des conflits.

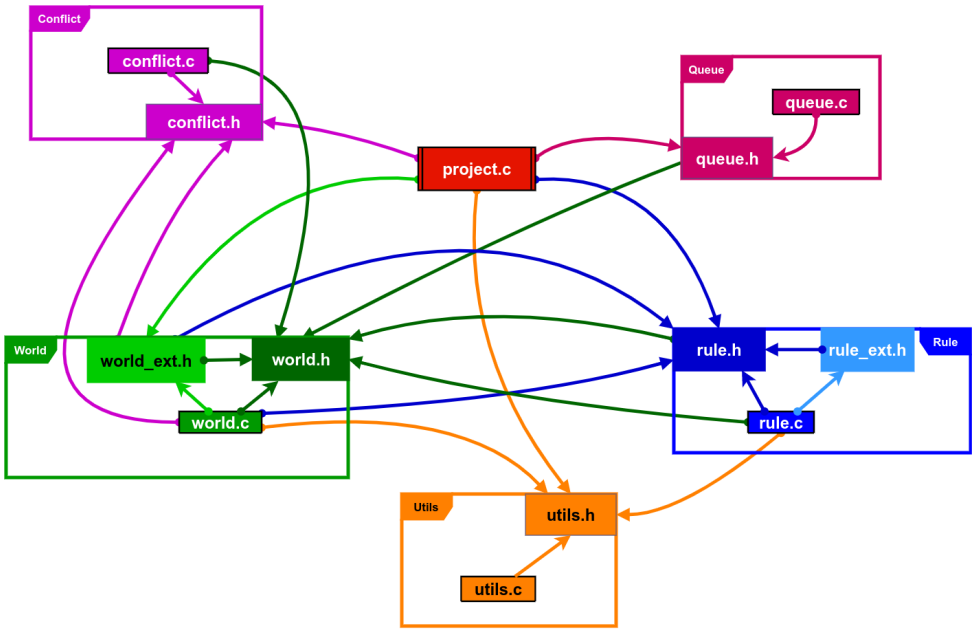
1.3 Organisation et dépendances du projet

Notre répertoire de travail est composé de trois dossiers :
— `src/` (source) : on y trouve tous les fichiers .c et .h nécessaires à la compilation du programme principal ; `project.c` (cf 1.1a)
— `tst/` (tests) : on y répertorie les fichiers nécessaires aux tests des fonctions créées dans le dossier `src/` (cf 1.1b) ;
— `doc/` (documents) : on y stocke tous les documents nécessaires à la compilation du rapport du projet ;

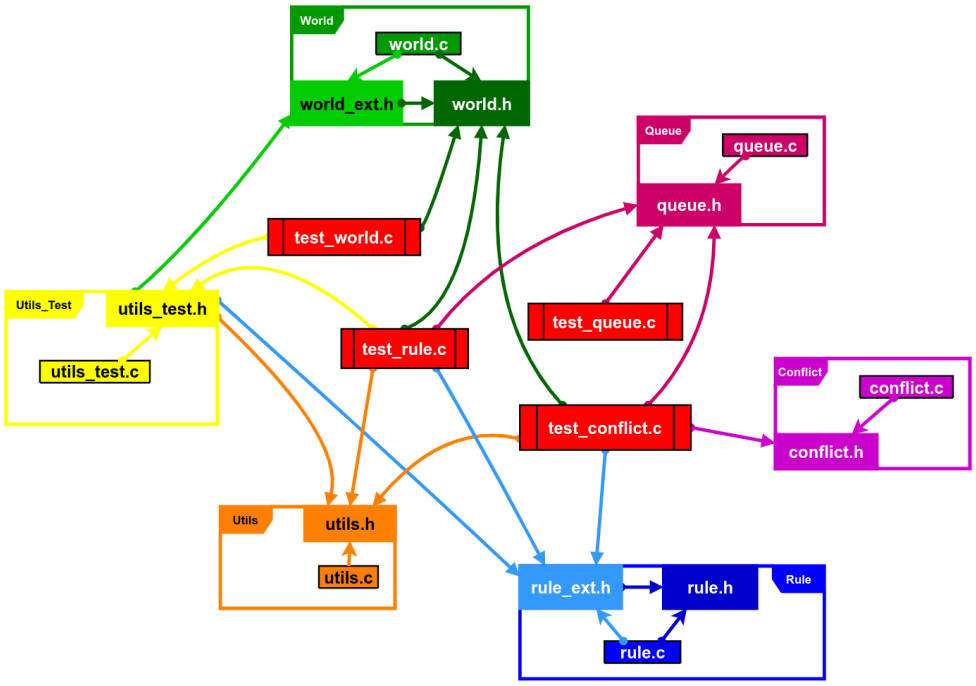
1. `https://fr.wikipedia.org/wiki/Automate_cellulaire`
2. `https://fr.wikipedia.org/wiki/Jeu_de_la_vie`

En racine du projet, seuls sont présents le fichier Makefile ainsi que le fichier `.gitignore` et le fichier `README.md` sont dans le répertoire principal.

Il a été choisi dans ce projet de séparer de manière consciencieuse les différents domaines du programme, le but étant de faciliter la modification partielle de l'algorithme tout en gardant un programme principal fonctionnel. Deux en-têtes `rule_ext.h` et `world_ext.h` ont été ajoutés (respectivement pour `rule.h` et `world.h`) afin d'étendre les accès aux fonctionnalités concernés, notamment pour les programmes de tests. L'ajout de la gestion des conflits et de la file à nécessité la création de fichiers dédiés à ces aspects particuliers du programme. Finalement, des fonctions ou des structures utiles dans plusieurs fichiers mais n'appartenant à aucun domaine particulier ont été regroupés dans les fichiers `utils.c` et `utils.h`.



(a) Dépendances des sources



(b) Dépendances des tests

FIGURE 1.1 – Graphes des dépendances du projet

2 | Monde et ses règles

2.1 Représentation et fonctionnement d'un monde

2.1.1 Définition d'un monde

La structure du monde est imposé par le sujet : un tableau de dimension `WIDTH` × `HEIGHT`. Il est constitué de nombres entiers positifs compris entre 0 et 4294967295 [1] représentant l'état d'une cellule. A chaque élément du tableau est associé une couleur, également appelé état. Il existe une bijection entre le tableau d'un monde et sa représentation 2D comme on peut le voir sur la figure 2.1. On a donc la relation suivante :

$$i * WIDTH + j = k,$$

k étant l'indice dans le tableau et (i, j) les indices dans le monde 2D.

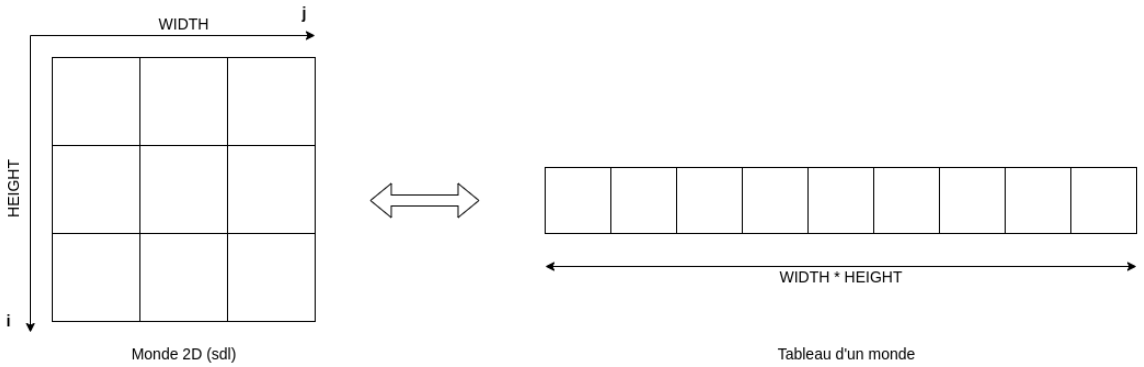


FIGURE 2.1 – Relation entre un tableau à une image de taille `WIDTH` × `HEIGHT`

Le monde est considéré comme un espace torique, c'est-à-dire que lorsqu'une cellule se déplace sur les bords de l'image, à l'image d'un planisphère [2], la cellule se retrouve sur le bord opposé. Pour ce faire il a fallu implémenter une fonction `int modulo(int x, int n)` complémentaire qui prenne en compte les nombres négatifs et qui retourne dans tous les cas un nombre positif.

```
1  int modulo(int x, int n)
2  {
3      if (x < 0) {
4          return n + (x % n);
5      } else
6          return x % n;
7  }
```

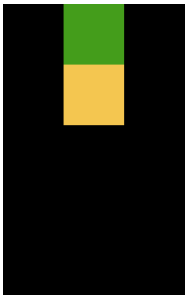
Pour faire le lien entre la représentation sous forme de tableau d'un monde et sa représentation 2D, nous avons implémenté la fonction `world_disp(struct world* w)`. Il a été nécessaire que le retour sur la sortie standard de la fonction soit compatible avec la librairie `sdl` afin de pouvoir afficher en couleur l'image (voir figure 2.2

1. Cet entier est la valeur maximale que peut prendre un entier positif (max unsigned_int) en C
2. Les auteurs de ce rapport se sont essayés à l'antanaclase, ils espèrent avoir correctement appréhender le concept derrière cette figure de style

```
3 5
#
0 1809732 0
5293812 0 0
0 0 0
0 0 0
0 0 0
#
0 1809732 0
0 5293812 0
0 0 0
0 0 0
0 0 0
#
```

(a) Sortie Standard

(b) Première image générée

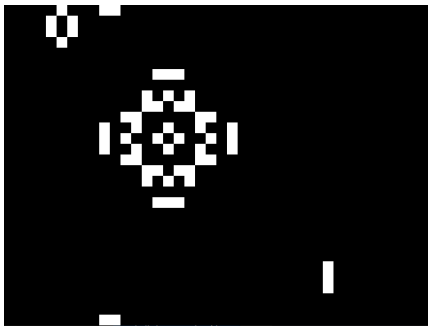


(c) Deuxième image générée

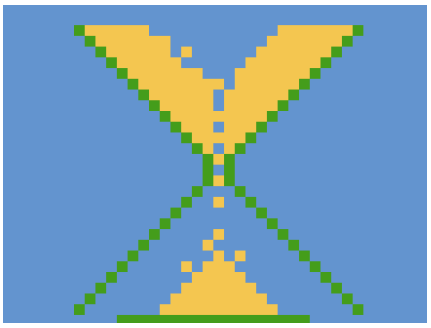
FIGURE 2.2 – Sortie de `world_display()` et résultat à l'affichage

2.1.2 Deux implémentations de monde

Nous avons utilisé deux grandes catégories de mondes lors de ce projet :
— le monde du jeu de la vie (d'abord en noir et blanc puis en couleur) qui est généré de manière aléatoire (voir la figure 2.3a) ;
— le monde du tas de sable représenté par la figure 2.3b, (qui peut être ou ne pas être aléatoire).



(a) Jeu de la vie



(b) Sablier

FIGURE 2.3 – Image obtenue avec SDL

L'implémentation de ces deux mondes a mis en évidence deux types de comportements pour une cellule : le changement d'état lié à son entourage (jeu de la vie) et le déplacement d'une particule (sablier).

Nous avons regroupé les quelques couleurs primaires utilisées dans ce projet dans une énumération enum state. Une couleur spéciale a été réservée sous le nom de `RANDOM_COLOR` et à l'image de la carte « 4 couleurs » dans un jeu de UNO, correspond de manière arbitraire à toutes les couleurs à la fois. Son utilisation est abordée plus en détaille dans la sous-section 2.1.3.

Afin de générer un monde initial, nous avons utilisé une fonction struct world world_init(char opt, int seed). Comme explicité plutôt, il doit est possible de générer un monde aléatoire. Pour ce faire, lors du lancement de l'exécutable principal `project`, il est nécessaire d'ajouter l'option -s (option programmable grâce à la libairie `optget`) et d'ajouter à la suite un entier qui sera une graine à la génération pseudo-aléatoire du monde.

Une dernière fonction est disponible concernant les mondes, la fonction void world_apply_rule() (manque ici les paramètres, pour des raisons visuelles). Elle permet d'appliquer les règles sur les cellules lorsque nécessaire. Cette fonction étant fortement liée à la façon dont nous résolvons les conflits, on en parle plus en détail dans la sous-section 3.2.2.

Finalement, lors de la génération du monde, il est important de définir ce qu'est une cellule vide. En effet, toujours lors de la gestion des conflits, nous avons fait le choix de ne déplacer une cellule que lorsque l'endroit où elle veut se déplacer est vide. Or tout état est en réalité un entier. Nous avons fait le choix de prendre la valeur 0 pour définir une cellule vide, ce qui correspond à la valeur `DEAD` dans l'énumération de nos états.

2.1.3 Tests concernant les mondes et complexité des fonctions

On teste la génération d'un monde aléatoire, c'est-à-dire qu'on regarde si deux mondes généré aléatoirement sont bien différents. Pour ce faire, comme il y a une petite chance que deux mondes aléatoires soient identiques, on laisse la possibilité de régénéré un monde dix fois avec la même graine. Si au cours des 1O itérations, le premier monde est identique aux 10 autres avec des graines différentes, alors on considère que la génération d'un monde aléatoire n'est pas respectée.

La fonction void world_apply_rule() étant intrinsèquement liée à la gestion des conflits et à de nombreux autres paramètres externes, elle est en réalité testée dans le fichier `test_conflict.c`.

En ce qui concerne la complexité, si on pose W la longueur et H la largeur de l'image, on a :
— world_display : on parcourt toutes les cases du monde et on les affiches donc la complexité est en $O(W \times H)$
— world_init : la complexité est de $O(W \times H)$ (car il faut associer à chaque cellule une valeur)

2.2 Implémentation et évolution des règles par motifs

Le principe d'un automate cellulaire est de définir des règles plus ou moins compliquées au début de l'algorithme et de les appliquer sur toutes les cellules du monde. Mais comment définir une règle ?

Une règle ne s'applique sur un pixel que dans des conditions spécifiques pré-établies. Nous nous sommes donc posez la question des cellules à observer, des conditions à remplir pour que la règle s'applique. Or dans le jeu de la vie, en plus de regarder l'état de la cellule à modifier, il faut également regarder ses cellules voisines. C'est pourquoi la première implémentation des règles repose sur les informations suivantes :
— un tableau de taille 9 (NB_NEIGHBOUR) contenant tous les voisins et la cellule a étudiée (située à l'indice 4 du tableau)
— un tableau qui contient les différentes modifications possibles (changements d'état ou déplacements) si la règle correspond à la situation du monde (la taille de ce tableau est déterminé par la valeur MAX_STATE) ;
— le nombre de changements possibles pour une règle

Par soucis de clarté, nous avons fait le choix d'implémenter les changements à appliquer par une structure qui comprend :
— la valeur de la cellule
— le déplacement selon `dx` et `dy`
Cette structure est la suivante dans notre code :

```
1    struct next_state {
2        unsigned int next_color;
3        int dx, dy;
4    };
```

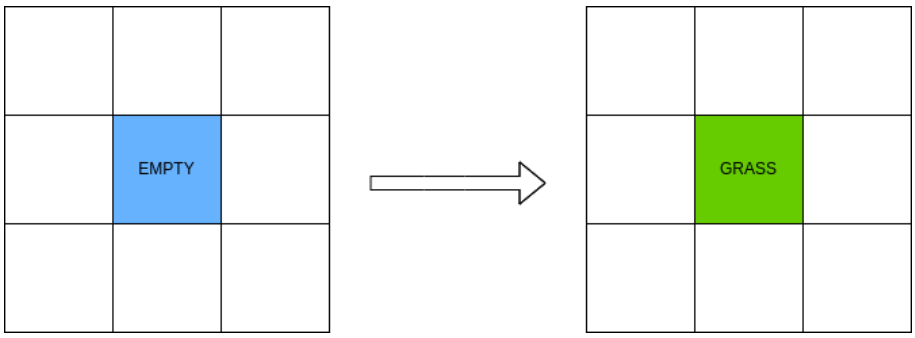
Nous avons choisi la convention suivante en ce qui concerne les changements d'une règle : soit on applique un déplacement, soit on applique un changement d'état. Le but de cette convention est de rester cohérent avec le principe de LAVOISIER[3] - *rien ne se perd, rien de ne crée, tout se transforme*. En effet, déplacer et modifier la couleur d'une cellule entre deux générations d'images revient à la faire disparaître.

Pour générer les règles de la vie, il a donc fallu générer tous les motifs possibles pour 9 voisins soit $2^9 = 512$ combinaisons. Nous nous sommes inspirés de l'écriture binaire pour générer tous les motifs en noir est blanc. L'inconvénient de cette méthode est qu'il correspond à la formule suivante :

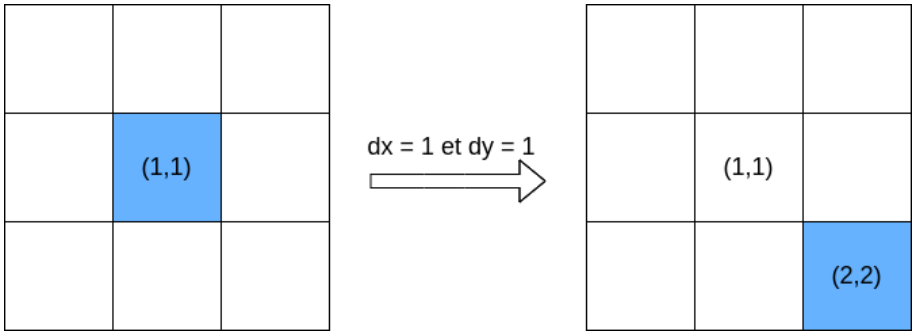
$$nombre\_couleurs^{nombre\_voisins}$$

En effet, plus le monde comporte d'états (ie plus il y a de couleurs), plus le nombre de motifs à générer est important. Il est également nécessaire de rappeler que la complexité de l'algorithme principal pour la génération d'une image dépend fortement du nombre de règles. En effet, pour chaque cellule l'algorithme cherche une règle compatible parmi toutes les règles existantes.

3. En réalité, dans le jeu de la vie, des cellules naissent ou meurent, c-à-d des cellules apparaissent et disparaissent. Le principe physique précédent n'est donc pas respecté. Pour être plus juste, le principe de LAVOISIER s'applique uniquement lorsqu'une cellule se déplace



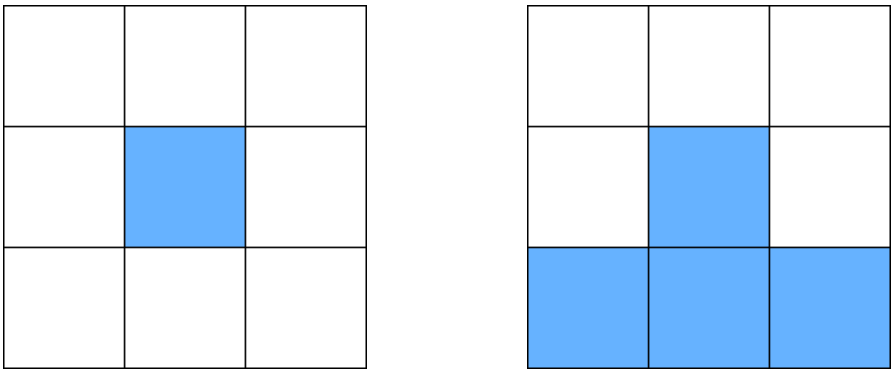
(a) Changement de couleurs



(b) Déplacement d'une particule

FIGURE 2.4 – Les deux types de modifications pour une règle

Une solution a été de faire des groupes de motifs en utilisant une couleur qui correspond à toutes les autres appelée `RANDOM_COLOR` (définit dans la sous-section 2.1.2). Cela à été fortement utile lors de l'implémentation de la chute de sable. Cependant, cette solution a un inconvénient. Lorsqu'on génère un motif on sait qu'il est unique. Cependant, plusieurs motifs avec des règles différentes peuvent être malencontreusement regroupés sous un même motif comportant `RANDOM_COLOR`. En effet, en regardant le graphe figure 2.5, si les cases blanches représentent cette couleur bonus, alors le motif 2 est un cas particulier du motif 1. En gagnant en complexité en diminuant le nombre de règle, on perd la bijection entre règle et motif.



(a) Motif 1 (b) Motif 2

FIGURE 2.5 – Deux motifs : le motif (b) est inclus dans le motif (a)

Il faut donc être prudent lors de l'implémentation des règles et de leur placement dans le tableau. En effet, c'est la première règle qui correspond qui est appliquée, les autres sont ignorées. Les règles les plus spécifiques doivent donc être en début du tableau et les plus génériques à la fin.

Cette solution n'est donc pas optimale de part sa forte complexité temporelle et sa fragilité. On doit ainsi trouver une autre manière de générer des règles, plus optimale, plus stable et surtout plus simple.

2.3 Refonte des règles : des motifs aux fonctions

2.3.1 Limites des motifs

On vient de voir que la forte complexité temporelle de la création des règles avec les motifs était un inconvénient majeur pour une bonne optimisation du projet. Mais encore, il ne semble pas intuitif d'utiliser des motifs avec certaines règles, par exemple du jeu de la vie. En effet, ce dernier ne comporte que deux règles :

— Si une cellule morte est entourée par 3 cases vivantes, alors elle devient vivante

— Si une cellule vivante est entourée par moins d'une case vivante (isolement) ou par plus de quatre cases vivantes (surpopulation), alors elle meurt.

Il paraît donc plus naturel de chercher un moyen d'implémenter uniquement deux règles pour créer un jeu de la vie.

2.3.2 Généricité des règles

Une manière simple de représenter une règle est de le faire avec une fonction. En retournant un booléen, elle indique ainsi si elle doit être appliquée ou non. Dans l'exemple du jeu de la vie, on doit créer deux règles dans `rule.c` :

— `int born(const struct world* w, unsigned int i, unsigned int j)` pour vérifier si une cellule morte doit vivre ou pas ;

— `int dead(const struct world* w, unsigned int i, unsigned int j)` pour vérifier si une cellule vivante doit mourir ou pas.

On doit alors modifier la structure d'une règle, en remplaçant le motif de la règle par un pointeur de fonction `int (*match)(const struct world*, unsigned int, unsigned int)`, contenant dans le cadre du jeu de la vie une de ces fonctions.

La structure des règles devient alors :

```
1    struct rule {
2        int (*match)(const struct world*, unsigned int, unsigned int);
3        unsigned int len_changes;
4        struct next_state next_state[MAX_STATE];
5    };
```

Pour ce qui est des méthodes, seul la vérification de la correspondance des règles changent, puisqu'il n'est plus nécessaire de vérifier la superposition d'un motif mais simplement à exécuter la fonction pointée par l'attribut `match` de la règle. `rule_match()` devient alors :

```
1    int rule_match(const struct world* w, const struct rule* r, unsigned int i, unsigned
     int j)
2    {
3        return r->match(w, i, j);
4    }
```

2.3.3 Avantages de la généricité des fonctions

Améliorer les règles en passant d'une correspondance par motif à une vérification par exécution d'une fonction possède plusieurs avantages.

Le premier est la simplicité d'écriture des règles, puisqu'il est naturel de les imaginer avec une fonction. Ensuite, le cas du jeu de la vie montre qu'on obtient un gain non négligeable en espace (passage de 2^9 à 2 règles). En réalité, certaines règles sont plus sensibles que d'autres à ce gain. Par exemple, un monde régit par des règles de chute de sable (comme utilisé dans les *achievements* 2 et 3) ne possède quasiment aucun gain avec la généricité des règles, ceux-ci étant imaginés comme pour une correspondance de motif. En revanche, une correspondance de motif reste quand même implémentable avec une fonction, ce que faisait d'ailleurs `find_neighbours()` et `compare_pattern ()`dans `rule_match`.

Enfin, on réalise que l'utilisation d'une structure abstraite dans l'en-tête facilite grandement cette transition, puisqu'il a été aisé de faire cette transition sans devoir modifier le projet principal.

3 | Changements entre deux mondes

Après avoir implémenté le monde et les règles, il faut maintenant mettre en place une stratégie d'applications de ces changements sur le monde.

3.1 File d'attente des changements

Une méthode possible pour stocker les changements à effectuer entre deux monde est d'utiliser une file, qui sera alors défilé au besoin. Cette méthode permet de ne stocker que les changements réels à appliquer tout en conservant leur ordre d'application grâce à la structure en file.

3.1.1 Représentation d'un changement par un nœud

Avant même de travailler sur la file de changements, il faut dans un premier temps concevoir et implémenter un élément (un changement) de cette file.

Nous avons fait le choix de modéliser un changement à effectuer par un nœud de liste simplement chaînée, comportant les attributs suivants :

— les coordonnées de la case à modifier ;
— le numéro de la règle à appliquer ;
— l'indice de l'état à appliquer à cette case ;
— un pointeur vers un autre nœud : le nœud suivant dans la file.

En C, l'implémentation d'un tel objet se fait à l'aide de la structure suivante :

```
1 struct change {
2     unsigned int i, j, idx_rule, idx_next_state;
3     struct change* next;
4 };
```

Cette structure est incorporée tel quel dans l'en-tête `queue.h` afin de rendre ses attributs accessibles aux autres parties du programme. Cela permet de récupérer et appliquer aisément la règle et l'état du changement.

3.1.2 Structure de la file en liste chaînée

Maintenant que l'on a modélisé les changements sous forme de nœuds, il nous suffit de les chaîner pour obtenir une liste chaînée de nœuds, que l'on pourra manipuler selon le fonctionnement d'une file.

Cette file est implémentée sous forme d'une structure avec les attributs suivants :

— un tableau de changements, qui constitue l'espace de stockage des nœuds de la file ;
— le nombre d'éléments de cette liste ;
— le pointeur vers le premier nœud de la file des changements en attente ;
— le pointeur vers le dernier nœud des changements en attente ;
— le pointeur vers le premier nœud de la liste chaînée des changements déjà effectués.

Ceci donne en C l'implémentation suivante :

```
1 struct queue {
2     unsigned int len_list_changes;
3     struct change list_changes[MAX_QUEUE_SIZE];
4     struct change* first_to_do;
5     struct change* first_done;
6     struct change* last_to_do;
7 };
```

Cette file était dans un premier temps déclarée entièrement dans le `queue.h`, comme l'est la structure des changements. Cependant, afin de préserver les attributs de la file, nous avons décidé de déclarer uniquement une structure abstraite `struct queue` dans l'en-tête `queue.h`, à l'image des règles. On ne travaille qu'avec une seule file `struct queue queue` déclarée comme variable globale dans le code source de la file `queue.c`.

En plus de la file souhaitée, cette structure comporte une liste chaînée des changements déjà effectués. Celle-ci fait ainsi office de « corbeille » de changements, sur lequel on peut éventuellement opérer et permet de « supprimer » un défilement sans devoir libérer un espace du tableau des changements de la file par allocation dynamique.

3.1.3 Manipulation et optimisation de la file

Une fois la structure de la file établie, il ne reste plus qu'à définir les méthodes permettant de manipuler la file. Pour cela, on définit trois méthodes principales :

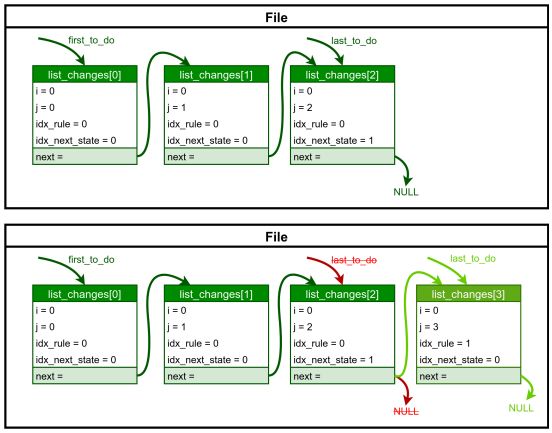
- `void queue_init()` pour initialiser une file déjà crée ;
- `int queue_is_not_empty()` qui indique si la file comporte un élément ou pas ;
- `void queue_append(unsigned int i, unsigned int j, unsigned int idx_rule, unsigned int idx_next_state)` pour ajouter un changement en queue d'une file existante ;
- `struct change* queue_pop()` pour récupérer le changement en tête de file et le supprimer de celle-ci.

Il est aussi utilisé deux fonctions supplémentaires :

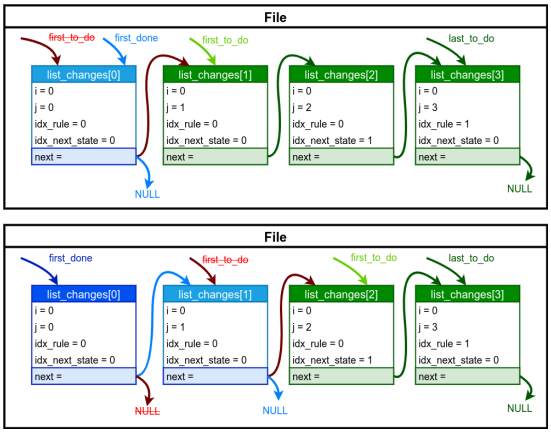
- `create_change()`, utilisée par `queue_append()` pour (dans une première version) ajouter simplement un changement dans le tableau et en définir les attributs du changement souhaité.
- `queue_is_not_empty()` qui indique si une file est vide ou non.

L'initialisation d'une file se fait simplement en mettant à zéro le nombre de ses éléments et en mettant tous ses pointeurs à NULL, ce qui permet de réinitialiser la file à chaque nouvelle image sans devoir effacer le contenu du tableau `list_changes`.

L'ajout d'un changement à la file se faisait dans un premier temps uniquement en éditant l'élément `list_change[len_list_changes]` (comme décrit dans la figure 3.1a). Cependant, cette stratégie d'apparence simple pose rapidement des problème au niveau du tableau `list_changes`, qui peut rapidement déborder et provoquer des erreurs de segmentations. Pour palier à cet inconvénient autrement qu'en agrandissant arbitrairement `MAX_QUEUE_SIZE` = `WIDTH*HEIGHT`, on réécrit les nouveaux changements en suivant la liste chaînée « corbeille » `first_done` dès que celle-ci possède un élément (voir figure 3.1c). Ceci résout les problèmes d'espace mémoire sans augmenter la taille du tableau des changements tant qu'on opère bien un seul changement par cellule du monde, ce qui est bien le cas dans le programme principal.

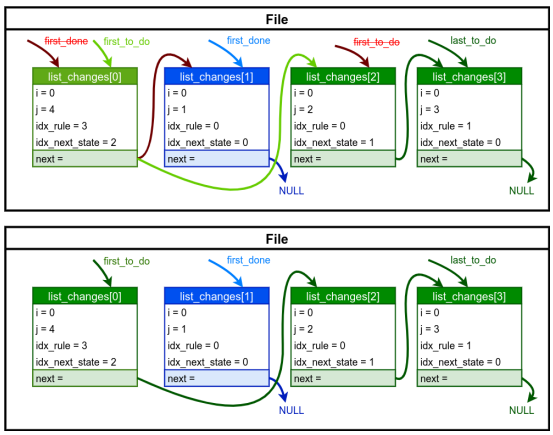


(a) Ajout d'un élément à la file



(b) Suppression d'un élément de la file

FIGURE 3.1 – Fonctionnement interne de la file



(c) Ajout d'un élément après une suppression (après optimisation)

FIGURE 3.1 – Fonctionnement interne de la file (suite)

3.1.4 Correction et complexités de la file

Pour vérifier le bon comportement de la file, nous avons créé l'exécutable `test_queue` (intégré aux tests exécutés par `makefile test`. Celui-ci permet de tester le bon fonctionnement des méthodes de la file, c'est-à-dire l'ajout, la suppression d'un changement et l'existence d'un élément dans la file ou non.. À chaque fois, on vérifie le bon nombre d'éléments et on vérifie leur bon emplacement dans l'espace mémoire, par un test et un affichage sur la sortie standard par les fonctions void queue_view_to_do() et void change_view(struct change* change).

D'après ce qu'on a vu précédemment, la complexité temporelle des différentes fonctions de manipulations de la file sont constantes, puisqu'elles ne sont constituées que d'un nombre constant de manipulations de pointeurs. De même, la complexité en espace de la file ne sera caractérisée que par l'espace pris par le stockage des changements par list_changes, qui est égale à la taille du monde.

En définitive, l'utilisation de cette file est de complexité temporelle constante et cette dernière possède une complexité spatiale linéaire en la taille du monde.

3.2 Gestion des conflits

On a vu précédemment que l'on parcourt chaque cellule d'un monde à chaque image pour estimer s'il y a un changement à effectuer ou pas. Cependant, dès que l'on inclut un déplacement de cases dans les règles, il peut arriver que deux cases veulent aller au même endroit à l'image suivante, ce qui pose problème. Ces situations constituent des conflits, que l'on doit gérer.

3.2.1 Marquage d'un conflit

Gérer un conflit commence d'abord par le détecter. Pour cela, on utilise un tableau de la taille du monde qui stocke le nombre de cellules qui souhaitent se déplacer vers une autre. On conserve également le nombre de conflits restant à gérer. Un conflit est ainsi implémentée en C par la structure suivante :

```
1 struct conflict {
2     unsigned int nb_conflicts;          //Nombre de cellules devant aller à celle-ci
3     unsigned int conflict_to_process;   //Nombre de conflits à gérer/appliquer
4 };
```

Marquer l'ensemble des conflits d'un monde revient alors à créer un tableau de struct conflict

Ensuite, il suffit d'incrémenter ces valeurs dès qu'une case doit se déplacer sur une cellule vide. Si la case d'arrivée n'est pas vide, la particule ne peut pas se déplacer, il n'y a donc pas de conflit. On obtient alors la détection des conflits suivante, au sein même de `project.c` :

```
if ((w.t[index_tmp] == EMPTY && (dx_tmp || dy_tmp)) || (!(dx_tmp || dy_tmp)) {
    t_conflicts[index_tmp].nb_conflicts += 1;
    t_conflicts[index_tmp].conflict_to_process += 1;
    queue_append(k, l, j, idx_change);
} else {
    fprintf(stderr, "Conflit perdant en %d %d car déplacement dans une case non vide.\n",
     k + dx_tmp, j = dy_tmp);
}
```

On remarque alors que l'ajout d'un changement à la file dépendant de l'état de la case d'arrivée. Si elle est vide, un changement est ajouté à la file. Si elle ne l'est pas, la cellule de départ reste inchangée. Ceci constitue donc déjà un début de résolution de conflit.

3.2.2 Résolution des conflits

Un fois les conflits sauvegardés, on les résout au moment de vider la file et d'appliquer les changements.

Pour cela, on regarde pour chaque changement combien de conflits concerne la case où doit s'appliquer le changement :

- Si la case n'a pas de conflit, alors on n'applique pas le changement, la cellule ayant déjà été modifiée ;
- Si la case ne comporte qu'une seule cellule voulant s'installer là, il n'y a pas de conflit et alors le changement est appliqué ;
- Si la case comporte plus d'une cellule voulant se déplacer là, alors on effectue un tirage au sort (suivant une loi uniforme ; $\frac{1}{\texttt{nb\_conflict}}$) pour appliquer ou non le changement :
  - Si le tirage est gagné, alors la cellule est marquée comme n'ayant plus de conflit et on on applique le changement ;
  - Sinon, la case est marquée comme ayant une cellule en moins à gérer et on n'applique pas le changement.

Dans notre programme, c'est la fonction solve_conflict () qui conditionne l'application d'un changement par world_apply_rule() :

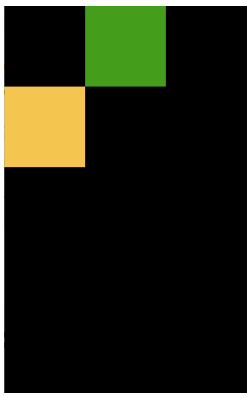
```
int solve_conflict(struct conflict t_conflicts[], unsigned int i, unsigned int j)
{
    struct conflict c = t_conflicts[i * WIDTH + j];
    int random = rand();
    if (c.nb_conflicts == 0) { //conflit déjà résolu
        return 0;
        fprintf(stderr, "Conflit perdant en %d %d car nb_conflicts =0\n", i, j);
    } else if (c.conflict_to_process == 1) { // conflit à résoudre avec ce changement
        return 1;
    } else if ((random % c.nb_conflicts) == 0) { // changement gagnant
        t_conflicts[i * WIDTH + j].nb_conflicts = 0;
        return 1;
    } else { // changement perdant
        t_conflicts[i * WIDTH + j].conflict_to_process -= 1;
        return 0;
    }
}
```

Enfin, il suffit d'appliquer le conflit si solve_conflict retourne 1. Pour cela, on change la cellule de destination selon le changement à appliquer et on rend la cellule de départ vide s'il y a eu un déplacement. La fonction world_apply_rule devient alors :

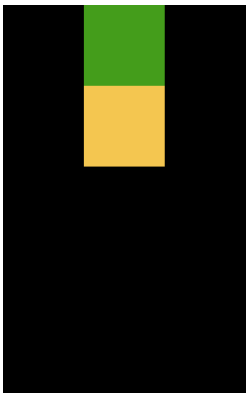
```c
void world_apply_rule(struct world* w, struct rule* r, int i, int j,
    unsigned int idx_change, struct conflict t_conflicts[])
{
    unsigned int dx = rule_change_dx(r, idx_change);
    unsigned int dy = rule_change_dy(r, idx_change);
    int s = solve_conflict(t_conflicts, modulo(i + dx, HEIGHT), modulo(j + dy, WIDTH));
    if (s) { //si la résolution de conflit dit qu'on doit appliquer le changement
        if (dx || dy) { // il y a déplacement
            w->t[modulo(i + dx, HEIGHT) * WIDTH + modulo(j + dy, WIDTH)] = rule_change_to
    (r, idx_change);
            w->t[i * WIDTH + j] = DEAD; // la cellule de départ est vide
        } else { // changement de couleur uniquement
            w->t[i * WIDTH + j] = rule_change_to(r, idx_change);
        }
    }
}
```

3.2.3 Correction et efficacité de la gestion des conflits

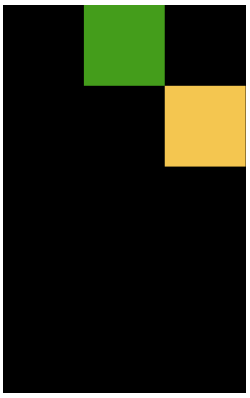
La résolution des conflits se faisant de manière aléatoire, il est difficile d'automatiser les tests pour vérifier si les conflits sont biens gérés comme tel. Nous avons donc choisi de faire un test simplement « visuel », en recréant un monde simple mettant en situation un conflit entre deux cellules voulant aller au même endroit, de manière cyclique au fur et à mesure des images. En générant un grand nombre d'images et donc de conflits, on doit pouvoir apercevoir le caractère aléatoire de la résolution de conflit. Les résultats de ces tests sont illustrés figure 3.2



(a) Création d'un conflit entre deux cellules allant au centre



(b) Victoire de la cellule jaune, attente de la cellule verte



(c) Attente de la cellule verte car case d'arrivée non-vide

(d) Victoire de la case verte dans la situation vu en figure 3.2a

FIGURE 3.2 – Tests de la résolution des conflits

Pour ce qui est de l'efficacité de la résolution des conflits, notre implémentation semble cohérente au reste de notre programme. En effet, nous ne parcourons jamais le tableau des conflits en entier (ce qui permet de garder les avantages temporaires de la file). De plus, les conflits sont stockés lors de l'ajout à la file et les conflits sont résolus lors des défilements des changement. On en déduit que la complexité spatiale de la résolution n'est uniquement impactée par le tableau de marquage des conflits, qui possède la même taille que le monde. De plus, la résolution ne nécessite aucune boucle, puisqu'elle ne fait que décider d'une application d'un changement ou non. Au final, on en déduis que la complexité de la gestion des conflits est constante en temps et linéaire en espace suivant la taille du monde.

4 | Conclusion

En conclusion de ce rapport, nous souhaitons faire un bilan des apports techniques, organisationnels et humains de ce projet ainsi que des difficultés rencontrées

D'un point de vu purement technique, ce projet nous a appris l'utilisation de Git et de `make` via le `Makefile`, des outils nouveaux pour deux étudiants tout droit sortis des classes préparatoires. La gestion des conflits a été particulièrement formateur, tout comme celle de la compilation séparée aidée par le `Makefile`. Pour ce qui est de la partie programmation en elle-même, c'est la mise en pratique des connaissances vues en cours et parfois abstraites qui ont été le plus enrichissant. Savoir quand et comment utiliser des structures, des pointeurs (variables et fonctions), utiliser des librairies (`getopt`, `SDL`) afin d'optimiser au mieux nos codes sont une première expérience qui nous semble fondamental pour la consolidation de nos connaissances. On a aussi pu sentir l'aspect « orienté objet » que proposait ce projet, notamment avec la structure abstraite des règles permettant de les modifier isolément du reste du programme. Enfin, tester notre code est aussi une compétence importante que l'on a commencé à acquérir grâce à ce projet, même si ce point nous a posé des difficultés, notamment savoir exactement quoi tester dans un code, et encore plus comment le tester.

Mais gérer ce premier projet nous a surtout obligé a acquérir des compétences organisationnelles et humaines essentielles pour le mener à bien. Respecter les contraintes, anticiper l'évolution du code et le tester au fil de son avancement ont été des compétences à apprendre sur le tas, surtout pour une première véritable expérience de projet. Penser à commenter de manière juste et concise nos algorithmes a été une autre de nos difficultés, même si peu contraignantes dans le cadre d'un projet comme celui-ci. La répartition de notre code dans de nombreux fichiers nous a posé quelques difficultés. Cela nous a poussé à repenser l'organisation de notre code et de réaliser l'importance d'un graphe de dépendances.

Pour finir, un projet en binôme se mène à deux, et il est fondamental de gérer le travail de chacun pour ne pas se marcher dessus et rester sur la même longueur d'onde. La bonne-entente et la coopération est nécessaire et essentielle pour avancer, et nous avons eu la chance de conserver ces avantages sans qui le projet n'aurait pas été mené à bout.