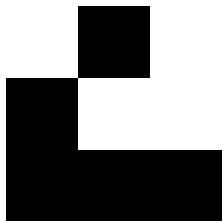


# Rapport Game Of Life

Brice Andrieux, Clément Bartolone, Mathieu Goudal, Loïc Laloux

March 2023



# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Algorithmes cellulaires neuraux</b>	<b>3</b>
2.1	Fonctionnement . . . . .	4
2.2	Implémentation . . . . .	6
<b>3</b>	<b>Voisinages de Margolus</b>	<b>8</b>
3.1	Fonctionnement . . . . .	8
3.2	Implémentation . . . . .	9
<b>4</b>	<b>Hashlife</b>	<b>11</b>
4.1	un peu d'histoire . . . . .	11
4.2	Aperçu . . . . .	12
4.3	Compression de l'espace . . . . .	13
4.3.1	QuadLife . . . . .	13
4.3.2	Canonicalisation du Quadtree . . . . .	18
4.4	Compression du temps . . . . .	19
4.4.1	Memoization : Almost-Hashlife . . . . .	19
4.4.2	Superspeed : Hashlife . . . . .	20
4.5	résultats . . . . .	23
<b>5</b>	<b>Interface Graphique</b>	<b>25</b>
5.1	Menu . . . . .	26
5.2	Fenêtre principale . . . . .	27
5.2.1	Barre d'option . . . . .	27
5.2.2	Affichage automate . . . . .	27
5.2.3	Déplacement . . . . .	28
5.2.4	Zoom . . . . .	28
5.3	Fenêtre de chargement . . . . .	28
<b>6</b>	<b>Architecture du Projet</b>	<b>29</b>
6.1	Package Automate . . . . .	30
6.1.1	package gridlife . . . . .	32
6.1.2	package hashlife . . . . .	33
6.2	Package Graphique . . . . .	34
6.3	Package Lecteur fichier . . . . .	36
6.4	Package Test . . . . .	36

# 1 Introduction

Le projet "**Game of Life**" est une simulation informatique développée par le mathématicien britannique **John Horton Conway** dans les années 1970. Cette simulation permet de créer des modèles de l'évolution de populations en utilisant des règles simples et automatisées. Notre but était de concevoir une application permettant de lancer une simulation de "**Game of life**".

Cependant, malgré sa simplicité apparente, la simulation "**Game of Life**" présente des problématiques complexes, notamment dans la mise en œuvre d'algorithmes performants pour gérer les évolutions massives de cellules sur un grand nombre de générations. Parmi les points durs de la problématique, on peut citer le développement de l'algorithme "**Hashlife**", qui permet d'accélérer considérablement les calculs en optimisant la gestion des cellules inactives.

Pour ce projet, Le travail a été réparti entre nous quatre. Pour cela chacun a fait un package. Le package automate/gridlife a été réalisé par **Brice Andrieux**, le package automate/hashlife a été réalisé par **Clément Bartolone**, le package graphique et la moitié du package lecteur fichier a été réalisé par **Mathieu Goudal** et le reste du package lecteur fichier et le package test a été réalisé par **Loïc Laloux**.

Dans ce rapport, nous allons aborder trois aspects clés de notre projet "**Game of Life**". Tout d'abord, nous allons explorer l'utilisation d'algorithmes neuraux. Ensuite, nous allons nous concentrer sur l'algorithme "**Hashlife**" et son implémentation dans la simulation "**Game of Life**". Enfin, nous allons présenter notre interface graphique et vous expliquer comment l'utiliser.

## 2 Algorithmes cellulaires neuraux

Les algorithmes cellulaires neuraux sont des automates cellulaires qui fonctionnent sur une grille bi dimensionnel de cellules contenant une valeur situé entre 0 et 1, et qui à l'aide d'une règle qui prend en compte les cases voisines d'une cellule via un voisinage de Moore, crée une grille où la règle a été appliquée à toutes les cellules de la grille de base. Le plus célèbre d'entre eux est bien sur le jeu de la vie, créé par John Conway, mais il est possible d'en faire une infinité d'algorithmes différents, dont les plus intéressant sont

sur [ce site](#) (et c'est aussi la liste d'algorithmes que nous avons implémenté).

## 2.1 Fonctionnement

Pour faire fonctionner un algorithme cellulaire neural, il faut avoir une règle, et une valeur qui est attribué à chaque cellule, qui sera déterminante pour appliquer la règle.

La règle est une fonction qui va prendre en entrée une valeur qui dépend de la **cellule principale** dont on veut changer la valeur mais aussi de son **voisinage** (les 8 cellules adjacentes orthogonalement ou diagonalement), et renvoie une nouvelle valeur pour la cellule principale, qui doit être comprise entre 0 et 1. La règle est inhérente à l'algorithme que l'on réalise, et peut ressembler à :

```
public float regle (float x){  
    if (x == 3f || x == 11f || x == 12f) return 1f;  
    return 0f; }
```

Dans cet exemple, si la valeur donné en argument est 3, 11 ou 12, alors la règle renvoie 1, sinon 0. Il n'y a pas de valeur intermédiaire, donc c'est équivalent au binaire, mais il est préférable d'utiliser des float car la plupart des algorithmes cellulaires neuraux ont besoin de float.

Pour ce qui est de la valeur associée à la cellule principale, la façon dont la valeur est calculé est par convolution avec une matrice de coefficients de taille 3 par 3 : chaque coefficient est associé à une des cellules parmi la principale et son voisinage selon leur positionnement : un coefficient pour la cellule en haut à gauche de la cellule principale, un autre pour celle en dessous etc...

On va alors faire la somme de chaque multiplication de la valeur de la cellule et de son coefficient associé. On peut par exemple avoir une matrice de coefficients telle que :

1	1	1
1	9	1
1	1	1

TABLE 1 – Exemple de matrice de coefficients

Dans cet exemple simple de coefficients, si la cellule principale vaut 0 et que 5 de ces voisines valent 1 (le reste valant 0), alors la convolution vaudra 5. si la cellule principale vaut 1, et que 2 de ses voisines valent 1 (le reste

valant 0), alors la convolution vaudra 11. C'est donc cette valeur qui sera donné à la règle pour déterminer la prochaine valeur de la cellule principale.

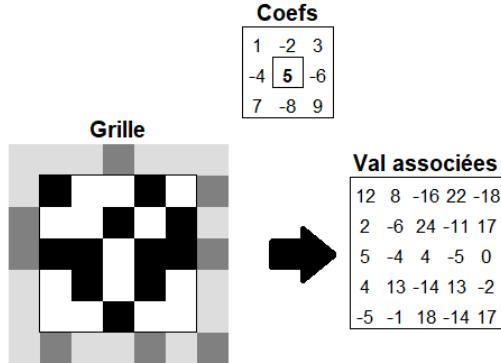


FIGURE 1 – Illustration du fonctionnement avec des valeurs arbitraires

**Totalelement** par coïncidence, la règle et les coefficients donnés en exemple sont ceux de Conway's Game Of Life : si une cellule est "morte" (vaut 0), alors la valeur x donné à la règle vaut entre 0 et 8, la cellule ne sera "vivante" (vaut 1) à la prochaine étape que si x vaut 3 donc si elle possède exactement 3 cellules voisines vivantes ; tandis que si elle est vivante, x vaut entre 9 et 17, elle ne sera vivante à la prochaine étape si x vaut 11 ou 12, donc si elle possède 2 ou 3 cellules voisines vivantes. A noter que si une cellule se trouve sur un des bords de la grille, les cellules situées sur le bord opposé de la grille peuvent être considérées comme parmi ses voisines ; ainsi toutes les cellules ont toujours 8 voisines.

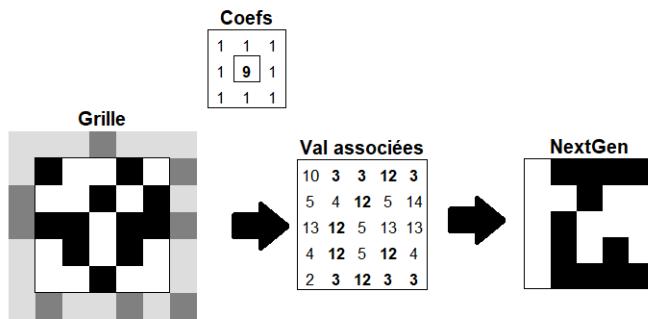


FIGURE 2 – Illustration du fonctionnement avec le jeu de la vie

Ces règles peuvent être plus que de simples égalités, et vont souvent renvoyer une valeur entre 0 et 1, plutôt que seulement 0 et 1. Ainsi, avec une combinaison de règle et coefficients minutieuse, il est possible d'observer différents comportements d'algorithmes cellulaires neuraux.

## 2.2 Implémentation

*Veuillez noter que, pour une partie des classes utilisées dans les algorithmes cellulaires neuraux et des voisinages de margolus, le nom de celles ci commence par "GOL", ce qui signifie "Game Of Life", et que c'est parfois un abus de langage car le projet porte sur le jeu de la vie, mais ce n'est pas pour autant que toutes les classes sont en rapport au jeu de la vie en tant que tel.*

Pour l'implémentation des algorithmes neuraux dans notre projet, on utilise une classe *Grille*, qui est un tableau de taille par taille cellules, taille étant une constante donné à l'initialisation de la Grille, et chaque cellule étant un *Float*. *Grille* implémente l'interface *Univers*, qui est commun au principe de grilles et de quadtree. *Grille* possède plusieurs méthodes, dont les principales sont *getValAt*, *setValAt*, et *Afficher*. *getValAt* retourne la valeur contenu dans la cellule dont on donne les coordonnées, *setValAt* permet de remplacer les valeur dans une cellule dont on donne les coordonnées par une nouvelle valeur donnée, et *Afficher* permet d'afficher la grille dans l'interface selon plusieurs arguments.

Cette *Grille* est utilisé pour les algorithmes neuraux, via la classe *GOLGrilleLifeLike*. Cette classe hérite de *GOLGrille*, qui elle même hérite de la classe abstraite *GOL*, qui sont 2 classes qui mettent les bases pour le fonctionnement des algorithmes neuraux, des voisinages de Margolus, mais aussi de Hashlife cependant uniquement pour la classe *GOL*.

*GOLGrilleLifeLike* a besoin de la taille, d'une règle, d'un mode, d'un *randType*, et optionnellement d'une couleur (qui n'est donc pas importante pour le fonctionnement de la classe) afin de faire fonctionner ses méthodes les plus importantes. Voici l'utilisation de ces variables, ainsi que les méthodes importantes :

- La taille est la largeur et la hauteur de la grille, et cette dernière est créée dans le constructeur.
- La règle est utilisé pour créer une instance de *DecodeGrilleLifeLike*, qui est une classe utilisé pour transformer la règle, qui est un *String*, en un tableau de coefficients, si la règle est conforme aux attentes

pour une règle d'algorithmes neuraux : 9 nombres séparé par des "/", chaque nombre correspondant au coefficient pour l'une des positions de cellules pour la convolution.

- Le randType détermine, lorsqu'on tente de remplir la grille de valeur aléatoire via la méthode *randomizeUnivers*, si la grille est rempli de 0 et de 1, ou si elle est rempli de valeurs flottantes entre 0 et 1, ou si la grille ne peut pas être rempli aléatoirement, en appelant respectivement les méthodes *randomizeUniversBin*, *randomizeUniversFloat*, ou aucun appel de méthodes. Le randType est utile car certains algorithmes fonctionnent sur des grilles où les valeurs ne peuvent avoir que 2 états, comme le jeu de la vie qui a des cellules qui ne peuvent être que "vivantes" ou "mortes", et d'autres qui sont adapté à avoir une situation initiale et ne pas être modifié via une grille aléatoire.
- Le mode est une valeur qui permet à la méthode *tick* de savoir quel est l'algorithme neural qui doit être utilisé. La méthode *tick* crée une nouvelle grille vide, et la rempli des valeurs de l'étape suivante de l'algorithme à l'aide du mode et du tableau de coefficient créé par l'instance de *DecodeGrilleLifeLike*, et la renvoie.
- La couleur est un tableau de 3 valeurs entre 0 et 1, qui permettent d'accentuer une couleur lors de l'affichage de la grille (chaque valeur accentue respectivement le rouge, le vert, et le bleu). La plupart du temps, les valeurs sont de 0, ce qui permet un affichage en noir pour les cellules valant 1, blanc pour celles valant 0, et en teintes de gris pour les valeurs entre 0 et 1.

Ainsi, le fonctionnement des Algorithmes cellulaires neuraux est assuré par les méthodes décrites ci dessus. Il est ensuite possible d'appeler depuis une classe qui coordonne toutes les autres classes une méthode *update*, qui va changer la grille contenue dans l'instance de *GOLGrilleLifeLike* en son étape suivante selon la méthode *tick*, donc selon l'algorithme, et cette grille peut être affichée grâce à sa méthode *Afficher*.

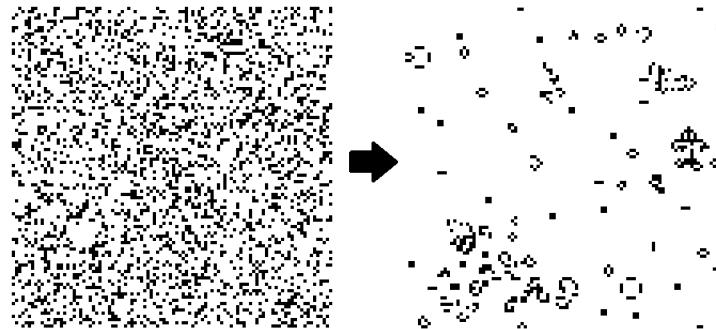


FIGURE 3 – Jeu de la vie : début aléatoire → 1000 générations plus tard

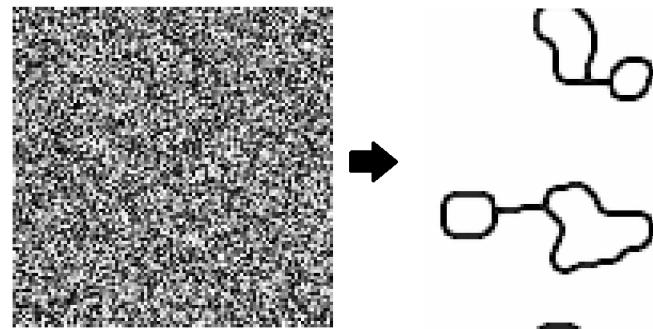


FIGURE 4 – Algorithme Pathways : début aléatoire → 1000 générations plus tard

### 3 Voisinages de Margolus

Le voisinage de Margolus est une méthode de voisinage différente du voisinage de Moore, ce qui permet de faire des automates cellulaires différent des algorithmes cellulaires neuraux. Tout comme ce dernier, il existe énormément de règles pour les voisinages de Margolus pouvant être intéressante, par exemple celles de [ce site](#), dont on a implémenté les règles données.

#### 3.1 Fonctionnement

Le voisinage de Margolus sont utilisés sur des grilles de taille finie de cases, ou cellules, qui possèdent une valeur qui est soit 0, soit 1, et consistent à regrouper les cellules en groupe de 2x2 cellules, auquel on associe une valeur

selon si les cellules du groupe valent 0 ou 1, et cette valeur accompagné par la règle de l'automate cellulaire, remplace le groupe de 2x2 cellules par un autre.

Les groupes de 2x2 cellules sont déterminé par leur position, mais aussi par le nombre de générations déjà effectués : à la première génération, on crée les groupes en partant de chaque duos de coordonnées paires : un groupe qui commence en (0,0) (qui englobe donc les cases (0,0), (0,1), (1,0) et (1,1)), en (0,2), en (2,0) etc... tandis qu'à la seconde génération, On crée les groupes en partant de chaque duos de coordonnées impaires ; on alterne ensuite entre chaque génération. A noter que dans le cas de notre implémentation, des cellules situées sur bord de la grille peuvent créer un groupe avec des cellules situées sur le bord opposé de la grille ; Ainsi si la grille a une taille paire, toutes les cellules feront partie d'un groupe à chaque génération.

La façon dont les valeurs sont calculées est similaire à du binaire : dans chaque groupe de 2x2 cellules, on associe la valeur de la cellule en haut à gauche avec le premier bit en partant de la droite, la cellule en haut à droite avec le second, celle en bas à gauche avec le troisième, et celle en bas à droite le quatrième. Ainsi, un groupe dont les 2 cellules du haut valent 1, et celle en dessous 0, la valeur associée à ce groupe est 0011 en binaire, soit 3 en base 10 ; un groupe dont la seule cellule qui vaut 1 est en bas à droite a comme valeur associé 1000 en binaire, soit 8 en base 10. Un groupe peut donc avoir une valeur associée entre 0000 et 1111, c'est à dire entre 0 et 15.

Image de grille à un moment pair ou impair + val des groupes

Le format de règle pour un voisinage de Margolus est une façon d'associer chaque groupe possible avec un autre, qui le remplacera lors de la prochaine génération. Le plus simple est d'avoir une liste de 16 valeurs, chacune entre 0 et 15, généralement toutes uniques (mais cela n'est pas obligatoire).

Ainsi, chaque groupe de cellules est converti en une valeur décimale, qui est associé à une autre valeur via la règle, est qui de nouveau converti en un groupe de cellules, qui remplacera le groupe de départ lors de la prochaine génération.

## 3.2 Implémentation

Similairement aux algorithmes cellulaires neuraux, les voisinages de Margolus se servent de la classe *Grille*, que l'on ne va pas réexpliquer.

La classe principale pour les voisinages de Margolus est **GOLGrilleMargolus**, qui hérite de *GOLGrille*, aussi déjà expliquée dans les algorithmes

cellulaires neuraux. Celle classe nécessite une taille, une règle, et optionnellement une couleur. La taille est celle de la grille, et la règle est utilisé pour initialiser une instance de *DecodeGrilleMargolus*.

*DecodeGrilleMargolus* fonctionne similairement à *DecodeGrilleLifeLike*, à l'exception que la règle doit être une suite de 16 nombres entiers entre 0 et 15 séparés par des "/", et si la règle est correcte, alors ils sont stocké dans l'ordre dans un tableau, et pour une valeur "x" représentant un groupe de cellule donnée, tab[x] sera la valeur correspondant au groupe de cellule de la génération suivante.

**GOLGrilleMargolus** possède aussi des méthodes similaire avec son homologue des algorithmes cellulaires neuraux, **GOLGrilleLifeLike**, excepté la méthode *tick* : cette méthode crée tous les groupes de 2x2 cellules nécessaires, en prenant en compte si on est sur une génération paire ou impaire afin de décaler ou non la création des groupes, récupère la valeur associé à ce groupe, et crée un nouveau groupe de cellules qui est placé dans une nouvelle instance de *Grille*, qui sera renvoyé à la fin de la méthode.

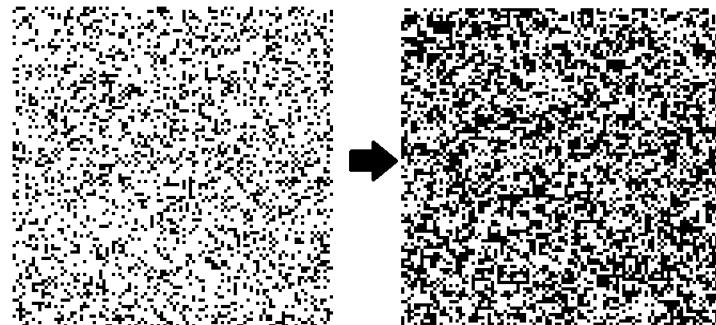


FIGURE 5 – Algorithme Tron : début aléatoire → 1000 générations plus tard

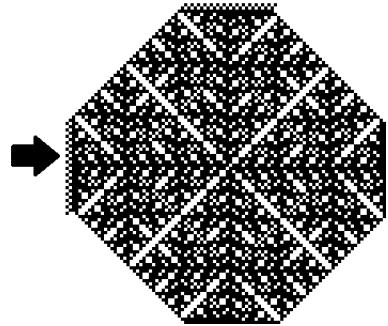


FIGURE 6 – Algorithme Toothpick : début prédéfini → quelques dizaines de générations plus tard

## 4 Hashlife

### 4.1 un peu d'histoire



FIGURE 7 – William (aussi appelé Bill) Gosper l'inventeur de Hashlife

Hashlife fut initialement développé en 1984 par le mathématicien William Gosper qui avait pour but d'optimiser la résolution de formes Régulières dans les automates cellulaires, notamment le jeu de la vie de John Conway. A l'époque les mathématiciens et les informaticiens essayaient de développer leur propre algorithme pour optimiser le jeu de la vie. Parmi les méthodes d'optimisations communes, on trouve par exemple l'utilisation complète des registres afin d'économiser de l'espace. Il est aussi possible de reconnaître les

oscillateurs et de les pré-calculer pour ne pas avoir à les prendre en compte dans le calcul complet. Cependant, ces optimisations devenaient lourdes et fournissaient des résultats moins intéressants.

En essayant de simplifier les choses, William Gosper crée alors l'algorithme Hashlife. Il publie donc "*Exploiting Regularities in Large Cellular Spaces*", un article détaillant son algorithme.

Toutefois, étant donné la complexité de l'algorithme, aucune implémentation ne sera créée directement après la publication de cet article, ce qui le cachera initialement du grand public et retardera son implémentation. Hashlife devient alors la source de rumeurs pendant les années 80 lorsque Rudy Rucker déclara avoir été témoin de simulations impressionnantes du jeu de la vie, lors d'une visite chez William Gosper.

Épaté, il dira bien plus tard :

*"Gosper showed me some incredible game of life simulations that were based on a weird speed-up algorithm of his, called 'Hashlife'. I remember him sharing the source code for the trick, but for me that was strictly a case of the Mathematician Godfather : He makes you an offer you can't understand"*

Plusieurs années plus tard, en 2005, sort GOLLY l'une des premières implémentations de Hashlife, ainsi que de Quicklife. Cette implémentation deviendra rapidement la référence dans le monde du jeu de la vie. Elle permet en effet de simuler des univers du jeu de la vie absolument gigantesques - grâce à des optimisations de mémoire - à des vitesses hallucinantes. En effet, GOLLY utilise Quicklife dans les cas où Hashlife est trop lent. L'intérêt du public pour ce logiciel est sans précédent et permet de populariser le jeu de la vie.

## 4.2 Aperçu

Comme dit précédemment, Hashlife est un algorithme utilisé pour optimiser les formes de vie régulières dans le jeu de la vie. Cependant, grâce à l'augmentation de la taille de la mémoire de nos ordinateurs, l'algorithme est plus généraliste. Il peut optimiser plus de conditions, et il n'est donc plus très pertinent de développer une solution hybride pour un projet simple.

Mais qu'optimise Hashlife ?

Dans le jeu de la vie classique, il existe deux grandes limitations :

- La vitesse
- La taille

William Gosper décide donc d'orienter son attention vers ces limitations.

Pour régler le problème de la taille de la simulation, on change de structure de données. C'est à dire que l'on utilise un quadtree sur lequel on effectue une génération du jeu de la vie sans décompresser le quadtree.

Pour optimiser la vitesse, on mémorise les nodes déjà utilisées afin de les réutiliser quand on en a besoin.

Pour aller plus loin, on utilise ce que Gosper appelle la "superspeed", cette partie de l'algorithme qui est responsable du gain de vitesse phénoménal de Hashlife.

## 4.3 Compression de l'espace

### 4.3.1 QuadLife

La base de Hashlife réside dans l'utilisation de quadtrees pour stocker les cellules de notre univers. Un quadtree, aussi appelé "arbre quaternaire" dans le pays des baguettes, est une structure de données arborescente représentée de cette manière :

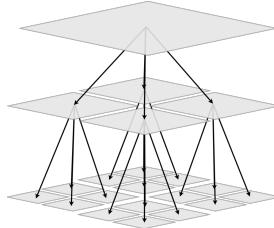


FIGURE 8 – représentation simple d'un quadtree

Elle est plus communément utilisée dans la compression de couleurs pour les images, mais cette structure de données a bien d'autres applications, l'une d'entre elle étant évidemment Hashlife !

Rajouter des cellules vivantes à un quadtree est en réalité assez simple. Il faut juste avoir défini au préalable la taille de notre univers (forcément en puissance de 2 car un quadtree ne peut que représenter des puissances de 2) puis utiliser une fonction récursive pour traverser le quadtree jusqu'à atteindre le niveau 0 et mettre la case à la valeur désirée. On peut aussi obtenir la valeur d'une cellule spécifique exactement de la même manière. On peut donc, si on le veut, implémenter le jeu de la vie de cette manière, mais cela serait extrêmement inefficace et empirerait notre problème de vitesse ainsi que de taille car on aurait alors décompressé notre quadtree.

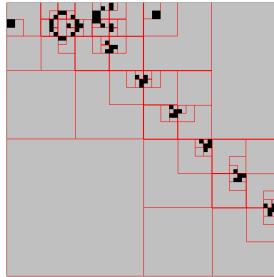


FIGURE 9 – utilisation de Quadtree pour stocker un pistolet à Gliders dans notre projet

Il nous faut donc une routine qui effectue le jeu de la vie directement sur un quadtree sans rien décompresser. Cette partie est très complexe mais je vais faire de mon mieux pour essayer de simplifier sans perdre des informations clés.

On veut faire une fonction dans laquelle on met un Quadtree en argument et la fonction nous renvoie le Quadtree de prochaine génération qu'on peut en déduire.

Premier problème : on ne peut pas déduire la valeur de toutes les cases du quadtree car celles qui sont collées à la bordure ont des voisins impossibles à déduire car externes à la node concernée. Il faudrait prendre la node parent pour trouver les voisins dans les nodes adjacentes, mais il faudrait alors prendre tout le quadtree (car la node parent a aussi des nodes adjacentes) donc ce n'est pas une solution viable.

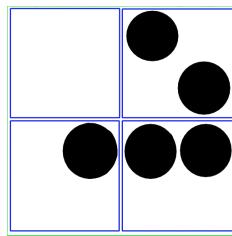


FIGURE 10 – on ne peut pas vérifier tous les voisins des cellules depuis les arbres bleus

A la place nous allons nous contenter de renvoyer la node centrale de notre quadtree, c'est ce qui va nous permettre d'éviter de décompresser le quadtree. Essentiellement, nous ne pouvons être sûrs de calculer que les nodes qui ont

des voisins dans notre quadtree, nous ne calculons que la node centrale pour laisser les enfants de notre quadtree faire le reste du boulot car la fonction est récursive.

Notre fonction prend donc un quadtree en entrée et renvoie la node centrale de celui-ci à la prochaine génération. Il peut donc sembler correct d'appliquer la fonction sur chaque node mais cela laisserait des "trous" non calculés dans notre quadtree, la fonction ne fonctionne toujours pas complètement...

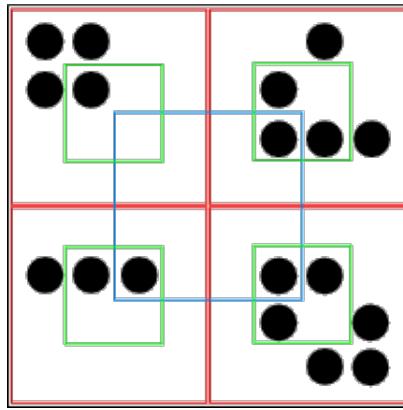


FIGURE 11 – voici pourquoi on ne peut pas simplement renvoyer la node centrale de chaque node. Beaucoup d'espace reste non-calculé et la node centrale initiale n'est pas complètement calculée

Notre fonction est cependant "correcte" pour un cas spécifique de notre calcul. Si notre node est juste au-dessus du niveau cellulaire, on peut l'adapter pour calculer directement les quatre cellules qui sont centrales à cet arbre de manière très similaire au jeu de la vie de base. Cette étape est appelé le calcul lent.

On lui crée donc sa propre fonction "calculLent()" qui prend un quadtree de niveau 2 et renvoie un arbre de niveau 1 à la prochaine génération. Notre objectif est à présent d'appeler calcul lent dans toutes les nodes qui arrivent au niveau cellulaire.

Pour ce faire, on doit vérifier chaque node non-vide ce qui n'équivaut **pas** à décompresser le quadtree, vu qu'on ne vérifie pas chaque cellule et qu'on saute les grands espaces vides lorsqu'il y en a.

C'est là que vient la solution de génie : Prenons une node de niveau 3 pour laquelle on veut calculer la prochaine génération de sa node centrale.

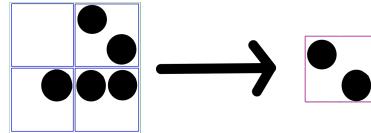


FIGURE 12 – l’arbre violet est le résultat obtenu en appliquant "calculLent()" sur l’arbre vert

Pour ce faire, nous allons générer 9 nodes intermédiaires de niveau 1, puis les rassembler en groupe de 4 pour obtenir des nodes de niveau 2 sur lesquelles on peut utiliser "calculLent()" pour assembler la node centrale de notre node. Rappelez-vous : notre fonction renvoie un arbre deux fois plus petit ! Ainsi on a effectué notre prochaine génération sur un arbre de niveau 3.

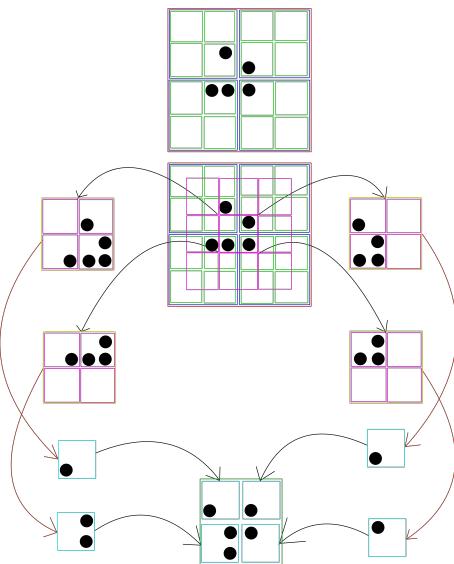


FIGURE 13 – on groupe les 9 nodes en groupes de 4 sur lesquelles on appelle nextGen (flèches rouges), on assemble le résultat, ce qui nous donne la node centrale une génération en avant de l’arbre initial

Chouette ! Mais devons-nous répéter l’opération pour chaque node ? Heureusement non ! Si on met juste une condition pour vérifier si le niveau est égal à 2, alors : si oui, on fait "calculLent()", sinon on fait l’algorithme décrit

précédemment. Appelons notre fonction "nextgen()".

```

if level = 2 then
|   calculLent();
else
    /* on génère les 9 nodes
    node1← CréerNodeAuxiliaire_1;
    node2← CréerNodeAuxiliaire_2;
    node3← CréerNodeAuxiliaire_3;
    node4← CréerNodeAuxiliaire_4;
    node5← CréerNodeAuxiliaire_5;
    node6← CréerNodeAuxiliaire_6;
    node7← CréerNodeAuxiliaire_7;
    node8← CréerNodeAuxiliaire_8;
    node9← CréerNodeAuxiliaire_9;
    /* on assemble les 9 nodes en 4 nodes
    /* on appelle récursivement l'algorithme afin d'obtenir la node centrale
    /* des 4 nodes à la prochaine génération
    tmpNode1← assembleNode(node1, node2, node4, node5).nextGeneration();
    tmpNode2← assembleNode(node2, node3, node5, node6).nextGeneration();
    tmpNode3← assembleNode(node4, node5, node7, node8).nextGeneration();
    tmpNode4← assembleNode(node5, node6, node8, node9).nextGeneration();
    /* on assemble les 4 nodes et on renvoie le résultat
    resultat← assemblerNode( tmpNode1,tmpNode2,tmpNode3,tmpNode4 );
end

```

### Algorithme 1 : nextgen()

A ce stade, la partie difficile de cette partie est terminée, mais nous n'avons pas encore fini, en effet si notre algorithme diminue la taille de notre node par 2 à chaque génération, nous n'allons pas pouvoir simuler tout notre quadtree et il rétrécira au fur et à mesure des générations.

La solution est très simple : créer un arbre "bordure" dont la node centrale est l'arbre sur lequel on veut passer à la prochaine génération. Le reste de ses nodes est vide. On appellera cette fonction "extend()".

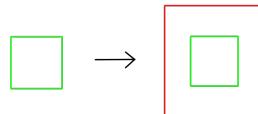


FIGURE 14 – l'arbre vert est l'arbre initial, l'arbre rouge est l'arbre bordure, *extend* retourne l'arbre bordure

Maintenant il ne nous reste plus qu'à assembler nos fonctions pour créer "fullnextgen()"

Dans un premier temps, on utilise "extend", puis on appelle "nextGen()" et on retourne le résultat. Nous avons maintenant calculé une génération du jeu de la vie dans un quadtree, ce qui n'est pas particulièrement rapide (même plus lent qu'un algorithme classique), de plus l'arbre actuel n'est pas hashable, il faut donc le modifier. Nous n'avons pourtant pas travaillé en vain,

notre algorithme permet d'avoir un univers "infini" tant que la population de celui-ci n'est pas trop grosse ce qui résout le premier problème du Jeu de la Vie.

#### 4.3.2 Canonicalisation du Quadtree

J'ai mentionné dans la partie précédente que notre arbre n'était pas hashable, en effet, pour obtenir ce statut nous allons devoir procéder à la canonicalisation de notre quadtree.

Qu'est-ce que la canonicalisation d'un quadtree ?

En termes simples, "Canonicaliser un arbre" revient à rassembler ses nodes similaires en blocs Canoniques. Ainsi, si une forme est trouvée de multiples fois dans l'arbre, nous ne la stockons qu'une seule fois et nous en passons la référence là où elle est nécessaire. L'intérêt primordial est de réduire la consommation de mémoire, mais dans le cas de Hashlife cela nous permet d'implémenter la mémoization de manière efficace.

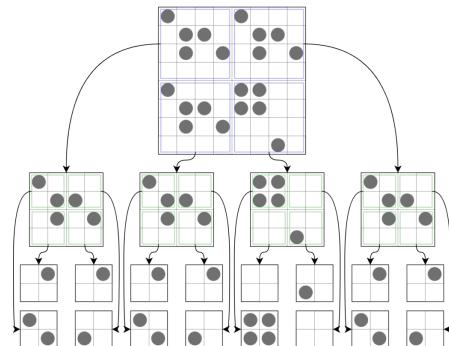


FIGURE 15 – voici la représentation de notre arbre actuel, certaines nodes sont identiques et devraient être regroupées !

Mais alors, comment allons-nous canoniser notre Quadtree ? En théorie c'est en réalité très simple. En effet, il suffit de stocker notre node dans une hashmap avec elle-même comme objet, ainsi quand une node a exactement les mêmes enfants et la même valeur, on peut considérer que les deux objets sont égaux. On peut alors supprimer le nouvel objet et renvoyer la node déjà stockée dans la hashmap. On peut appeler cette fonction "canonicaliser()" ou encore "integre()". Étant donné quelle est moins verbale je vais choisir

la deuxième option. Finalement, on modifie le constructeur des nodes canoniques afin de les canoniser lors de leur création, on a juste à intégrer la node résultante de la création.

Si on appelle systématiquement le constructeur des nodes canoniques pour créer nos nodes, notre quadtree se canonisera tout seul.

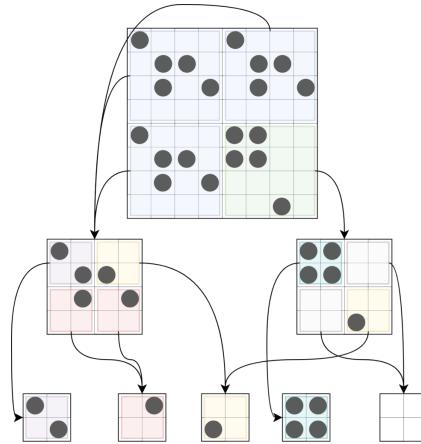


FIGURE 16 – voici la représentation de notre arbre canonisé, les nodes identiques on été regroupées !

Même si nous gérons mieux notre mémoire, notre jeu de la vie reste lent. Heureusement, cette étape est la fondation de la partie à suivre qui nous permettra enfin d'aller plus vite qu'une implémentassions classique du jeu de la vie. Il est aussi intéressant de noter que les espaces de forte densité prennent beaucoup moins d'espace que précédemment si ils sont réguliers. Ainsi, nous avons quasiment totalement fait abstraction de la notion d'espace du jeu de la vie.

## 4.4 Compression du temps

### 4.4.1 Memoization : Almost-Hashlife

Maintenant que notre quadtree est canonisé il est trivial d'implémenter la Mémoisation il suffit simplement de rajouter une node appelée next dans notre structure de node, on l'initialisera à null avec le constructeur, cependant on ne prendra pas ce nouveau membre en compte lors de la canonicalisation. il ne suffit plus que de modifier la fonction "nextGen()" pour vérifier si next

est null, si ce n'est pas le cas, on fait le calcul comme précédemment et on initialise next à la valeur du résultat, en revanche si next n'est pas null (et a donc déjà été calculé) on renvoie simplement next.

```

if next = null then
    /* si la node n'est pas calculé on la calcule ici
    |   next ← calculLent();
else
end
/* on renvoie la node précalculée
resultat ← next;

```

### **Algorithme 2 :** nouvelle définition de nextGeneration()

Ainsi on ne refait pas le même calcul deux fois, et après quelques générations énormément de nodes ont été mémoisées, à ce stade l'ordinateur n'effectue quasiment que des lectures dans une hashmap ce qui est très rapide, et ainsi la vitesse d'exécution explose, on est capable de simuler des centaines de génération en une secondes, de plus si le pattern simulé est régulier, il est possible que l'arbre entier soit mémoisé totalement, à ce stade la simulation devient stable et on n'utilise plus du tout "calculLent()" ce qui va nous permettre de simuler des dizaines de milliers de générations en une seconde, ce qui est vraiment incroyable, mais vu que la simulation n'évolue plus, un peu ennuyant...

On a maintenant un algorithme extrêmement rapide, probablement bien plus rapide que tout autre implémentation du jeu de la vie, même quicklife ne peut pas nous rattraper en nombre de génération par secondes, on pourrait s'arrêter là si on le voulait, la vitesse est suffisante pour simuler des patterns comme le triangle de Sierpinski à des échelles remarquables. Il est à noter que la vitesse est suffisante, nous voulons aller au-delà, et c'est ce que va nous permettre Hashlife.

#### 4.4.2 Superspeed : Hashlife

Almost-Hashlife nous permet de calculer génération par génération le jeu de la vie dans un espace quasi-infini, le temps de calcul entre deux générations est le plus petit possible et ne peut être réduit que grâce à des optimisations de bas niveaux comme l'utilisation complète et efficace des registres, du parallelisme SIMD ou du multithreading "simple". Mais d'un point de vue purement algorithmique nous avons atteint le temps de calcul minimal. Mais alors comment accélérer encore plus la simulation ? la solution est en réalité plutôt simple : calculer plus de générations pour chaque appel de nextGen(). Pour

bien illustrer l'optimisation que nous allons implémenter nous devons revenir a l'implémentation de Quadlife que nous avons abordé précédemment :

Pour passer a la prochaine génération nous générerons neufs nodes intermédiaires que nous assemblons pour former quatre nodes sur lesquelles on appelle nextGen() récursivement afin de passer a la prochaine génération Comme montré dans la figure 7.

Pour un arbre de niveau 3 nos 9 nodes temporaires sont de niveau  $3-2 = 1$ , cela ne doit pas changer, cependant, nous allons changer d'approche pour les générer, a la place d'aller chercher individuellement les 4 nodes nécessaires pour la fabrication de chacune des 9 nodes temporaires individuelles, nous allons prendre des arbres un niveau plus haut ainsi la node temporaire centrale sera la node centrale de l'arbre sur lequel on a appellé nextGen qui est dans notre exemple de niveau  $3-1 = 2$  les nodes temporaires nw , ne , sw et se sont simplement les enfants correspondants de notre arbre initial, les nodes n, s , w et e sont des mélanges entre les nodes diagonales ex : le bas de nw et le haut de sw combiné donne w.

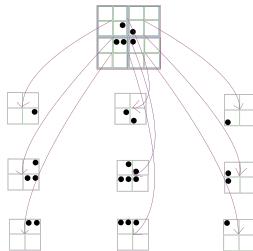


FIGURE 17 – Première étape de l'algorithme de Hashlife

Ces nouvelles nodes temporaires sont de niveau 2 comme dit précédemment, il faut que nous retrouvons nos nodes de niveau 1, pour les retrouver on pourrait prendre leur node centrale, mais on n'aurais rien fait de très intéressant, a la place nous allons utiliser nextGen pour trouver leur node centrale une génération en avant puis les assembler en nos 4 nodes.

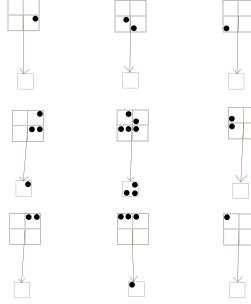


FIGURE 18 – Deuxième étape de l'algorithme de Hashlife

Nous pouvons alors réutiliser nextGen dessus comme "d'habitude" afin de trouver notre node évoluée cette fois ci non pas d'une génération mais de 2 générations ! si on applique l'algorithme sur un niveau plus haut (4) on remarque que l'on ne fait pas 2 mais 4 génération en un seul appel ! Et la tendance continue, pour un arbre de niveau 8 on fait 64 générations ! Si vous avez l'oeil vous remarquerez que c'est une croissance exponentielle d'écrite par l'équation  $nbGen = 2^{niveau-2}$ .

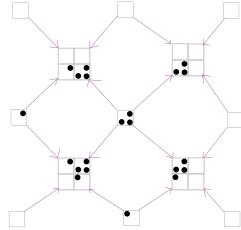


FIGURE 19 – Troisième étape de l'algorithme de Hashlife, ensuite on applique le même algorithme que almost hashlife sur les 4 nodes générées.

Pourtant un tel algorithme ne semble pas être bénéfique pour la performance car on calcule beaucoup de nodes déjà calculées, Cependant l'utilisation de la memoization va nous permettre d'exploiter ces propriétés sans trop affecter la durée entre deux changements de génération. Ainsi la première génération de Hashlife peut prendre beaucoup de temps à calculer (dans notre projet cela peut prendre entre 10 secondes et 5min dans le modèle **metacell-galaxy.rle**), mais dès qu'assez de nodes ont été memoizées la performance devient difficile à représenter, alors que l'on calculait des dizaines de milliers de générations en une seconde avec almost-Hashlife on peut maintenant

calculer si vite que le nombre de génération est difficile à retenir dans un entier long non-signé dont la limite est **18,446,744,073,709,551,615**, sans compter le fait qu'on peut augmenter la taille de l'arbre car plus l'arbre est gros plus on saute de générations ! Il a donc été nécessaire d'imposer un court délai (*5ms*) entre deux saut de génération de Hashlife afin que l'on puisse observer ce que fait l'algorithme.

Ainsi Hashlife nous permet de simuler le jeu de la vie de la manière la plus rapide possible, que pouvons nous faire avec ?

## 4.5 résultats

Voici une section un peu plus intéressante ! Tout les modèles que nous allons vous montrer sont importés et simulés dans notre projet grâce à un lecteur RLE vous en trouverez plus d'informations dans la partie interface graphique. Tous ces modèles proviennent de [ce site](#). Cessons donc parler d'algorithme voyons voir se qu'ils ont dans le ventre :



FIGURE 20 – un pistolet à gros vaisseaux, almost hashlife suffit pour voir ce modèle en action

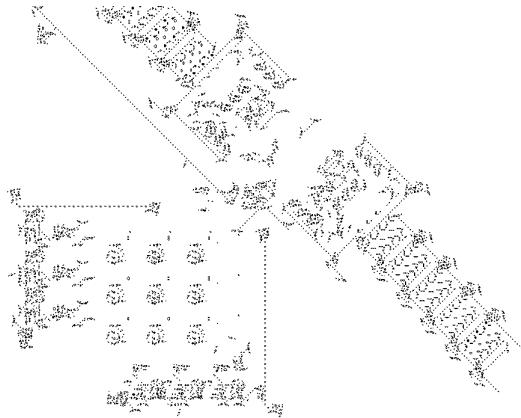


FIGURE 21 – une machine de turing qui calcule les nombres premiers, on peut voir la sortie de celle ci en bas a gauche en flux de gliders

Les modèles suivants sont visionnables en temps réel grâce a hashlife

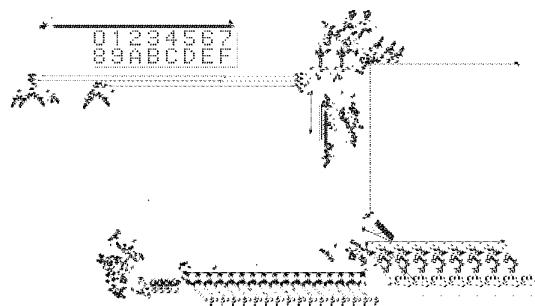


FIGURE 22 – un affichage de lettres et chiffres on peut le voir afficher en temps réel chaque ligne de l'affichage. La première génération peut prendre 45s

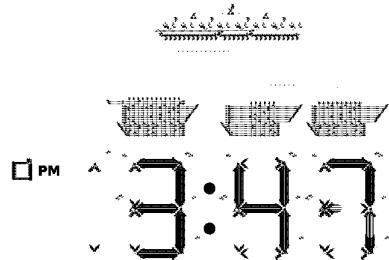


FIGURE 23 – une horloge digitale on peux la voir changer d'heure la première génération peux prendre *1min20*

Et un dernier pour la route, et aussi le plus impressionnant !

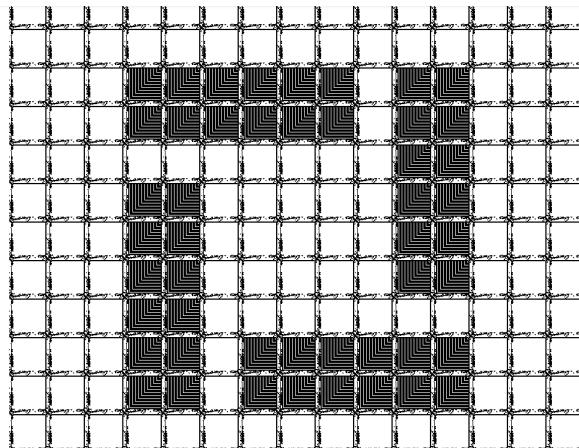


FIGURE 24 – le saint graal, le jeux de la vie dans le jeu de la vie grâce aux metacells, ici la galaxie de Kok la première génération peux prendre environ *3-5min*, ce pattern est très dur a simuler sur mon petit laptop

Voila qui conclus cette explication de Hashlife.

## 5 Interface Graphique

L'interface graphique de notre application se décompose en 3 composants principaux :

- Le Menu

- La fenêtre principale de l'application
- La fenêtre de chargement d'un modèle

## 5.1 Menu

Le Menu est la page d'accueil de l'application, il sert à choisir le type d'automate cellulaire souhaité pour la simulation de notre "Game of life". Le choix de l'automate cellulaire ce fait grâce aux boutons :

- **Hashlife** bouton (initialise un automate cellulaire utilisant l'algorithme hahslife)
- **Margolus** bouton (initialise un automate cellulaire utilisant les algorithmes de margolus)
- **Neural** bouton (initialise un automate cellulaire utilisant les algorithmes neuronaux)



FIGURE 25 – Bouton Hashlife du menu

Chaque bouton est accompagné sur sa droite d'un bouton d'information. Chaque bouton ouvre une fenêtre pop-up contenant un lien qui renvoie vers un site permettant d'avoir plus d'informations sur l'option à sa gauche.

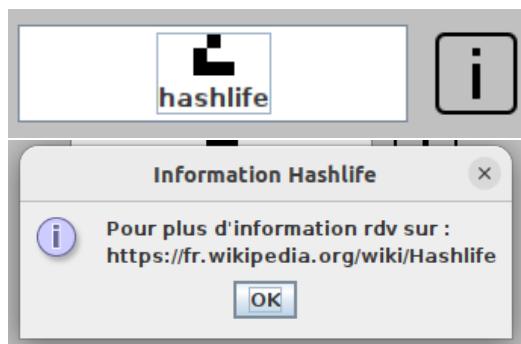


FIGURE 26 – Bouton d'information de Hashlife

## 5.2 Fenêtre principale

La fenêtre principale permet l'affichage de l'automate cellulaire ainsi que son contrôle grâce de multiples boutons présent sur une **JToolBar** : la barre d'options

### 5.2.1 Barre d'option

La barre d'option contient 10 boutons :

- **Home** (permet de revenir dans le menu pour changer d'automate cellulaire)
- **Move** (permet de ce déplacer dans l'affichage de l'automate et de zoomer)
- **Secret** (je vous laisse découvrir le secret par vous même)
- **Rule** (affiche la règle courante suivit par l'automate cellulaire)
- **Choix rules** (un menu déroulant contenant les différentes règles que l'automate peut suivre (change en fonction de l'automate))
- **Play** (permet de lancer/mettre en pause la simulation de l'automate cellulaire)
- **Random** (permet de remplir l'automate cellulaire aléatoirement)
- **Clear** (permet de vider l'automate cellulaire)
- **Speed** (permet d'accélérer la vitesse de la simulation, il n'est disponible que si l'automate utilise l'algorithme hashlife)
- **Modele** (un menu déroulant contenant différents modèle importable du jeu de la vie (les modèles changent en fonction de l'automate initialisé))

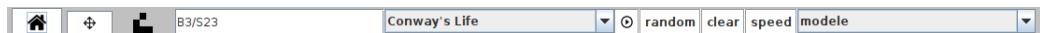


FIGURE 27 – Barre d'options

### 5.2.2 Affichage automate

Comme dis précédemment l'affichage automate affiche la simulation du jeu de la vie en cours. Dans cette affichage vous pouvez cliquer affin d'ajouter une cellule vivante (représenter par un pixel noir) sur l'endroit cliqué. Pour se déplacer et zoomer dans l'affichage c'est très simple :

### 5.2.3 Déplacement

- **z/Z** déplacement vers l'avant
- **q/Q** déplacement vers la gauche
- **d/D** déplacement vers la droite
- **s/S** déplacement vers l'arrière

### 5.2.4 Zoom

- **a/A** zoom +
- **e/E** zoom -

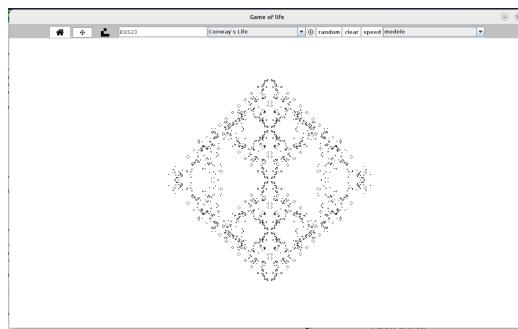


FIGURE 28 – Affichage d'une simulation

## 5.3 Fenêtre de chargement

Lorsque l'on importe un nouveau modèle alors la fenêtre de chargement s'ouvre et une barre de progression est affiché. La barre selon le niveau d'importation du modèle.

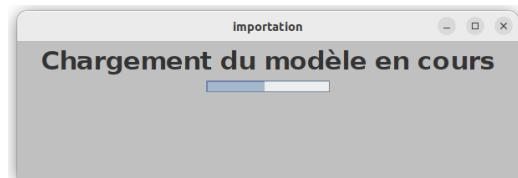


FIGURE 29 – Fenêtre de chargement

## 6 Architecture du Projet

Pour notre Projet, nous avons créé plusieurs package différents afin d'avoir un espace de travail ordonné et rigoureux. Ainsi nous avons un package **build** contenant toute les classes après compilation, un package **doc** contenant toute la javadoc compilé, un package **lib** contenant la librairie afin d'exécuter correctement les tests, un package **ressource** qui contient les package CSV, img et RLE contenant respectivement les fichiers au format .csv, .png et .rle et enfin nous avons le package **src** qui contient tout notre code.

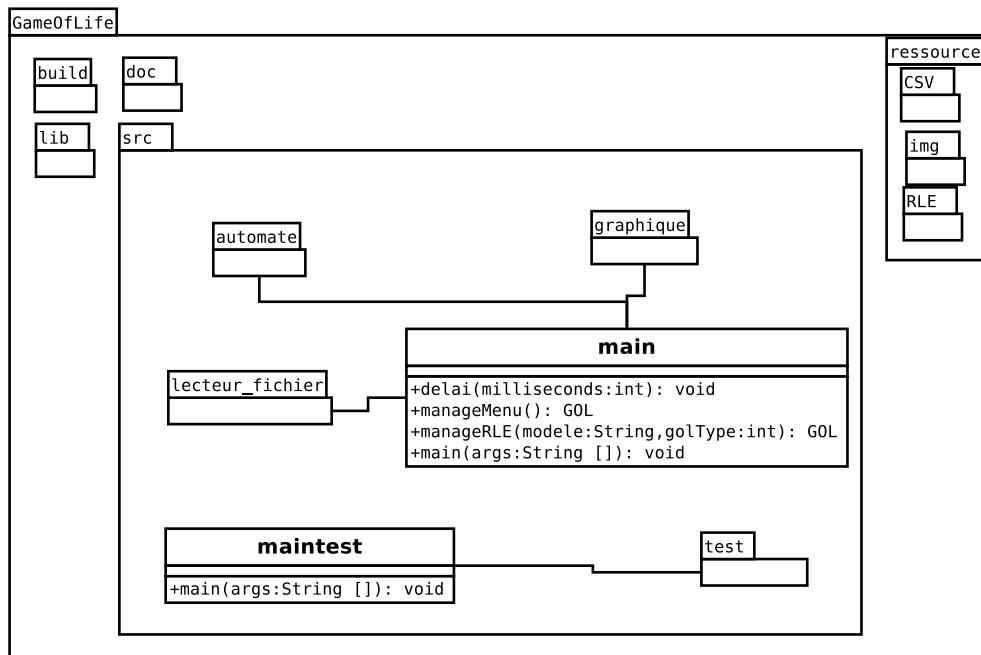


FIGURE 30 – Illustration du package GameOfLife

Dans le package src nous avons différents package ainsi que les deux classes main :

- La classe **main** qui lance le jeu de la vie.
- La classe **maintest** qui lance tout les tests présents dans le package test et affiche la réussite ou l'échec de chacun.

Parmi les différents package nous avons les quatre suivants :

- Le package **automate**
- Le package **graphique**

- Le package **lecteur fichier**
- Le package **test**

## 6.1 Package Automate

Le package **automate** va initialiser les différentes versions du jeu de la vie ou de l'automate cellulaire. Il est composé de deux sous-package :

- Le package **gridlife** initialisant les différentes versions d'automates cellulaires sur une grille.
- Le package **hashlife** initialisant le jeu de la vie sur un ensemble de quadtree.

Ce package compte aussi deux classes abstraites et une interface qui initialise les méthodes communes au deux package. La classe abstraite **GOL** généralise les variables communes et les fonction utilisées dans les classes qui l'héritent (**GOLqTree**, **GOLGrille**), cette classe va être utilisée dans l'interface et le main afin de pouvoir changer rapidement d'instance de **GOL** et initialiser les modèles RLE.

La classe abstraite **Decode** va servir à décoder les règles du jeu de la vie ou de l'automate cellulaire. Elle contient les règles ainsi qu'une méthode abstraite qui doit vérifier si les règles sont valides ainsi qu'un accesseur. Cette classe sera hérité par une classe pour chaque instance différente de **GOL** :

- la classe **DecodeqTree** dans le package hashlife qui va décoder les règles de format standard du jeu de la vie (B.../S...) exemple : Conway's Life (premier jeu de la vie) a pour règles B3/S23 .
- la classe **DecodeGrilleLifeLike** dans le package gridlife qui va décoder les règles de format (././././././././././.) prenant 15 chiffres entier entre 0 et 15 (inclus) qui est utilisé pour les algorithmes neuraux.
- la classe **DecodeGrilleMargolus** dans le package gridlife qui va décoder les règles de format (./././././.) prenant 8 chiffres entre 0 et 1 qui est utilisé pour les voisinages de Margolus.

L'interface **Univers** va être implémentée par les "supports" du jeu de la vie ou des automates cellulaires. La classe **Grille** dans le package gridlife et la classe **GOLqTreeNode** dans le package hashlife. Cette interface va initialisé les accesseurs et modificateurs de valeur à une certaine coordonnée ainsi qu'une méthode permettant l'affichage sur un Graphics.

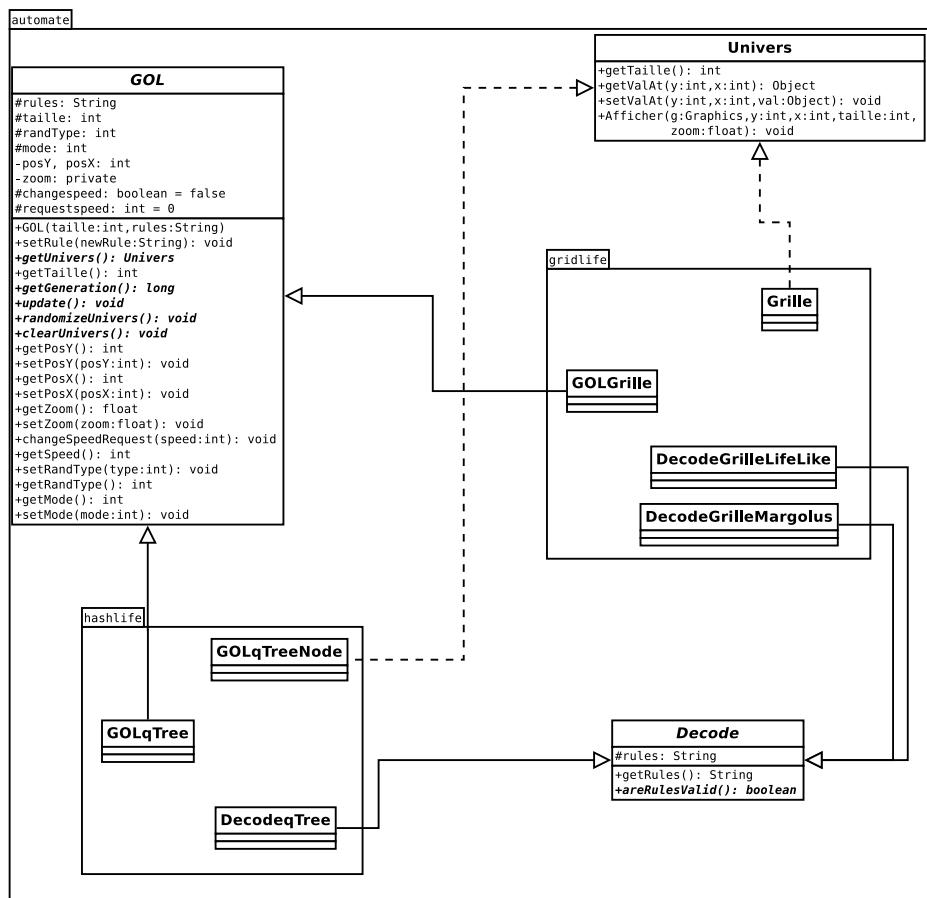


FIGURE 31 – Illustration du package automate

### 6.1.1 package gridlife

Le package gridlife va représenter les automates cellulaires sur grille. Il est composé des différentes classes héritant du package automate :

- la classe **Grille**
- la classe **GOLGrille**
- la classe **DecodeGrilleLifeLike**
- la classe **DecodeGrilleMargolus**

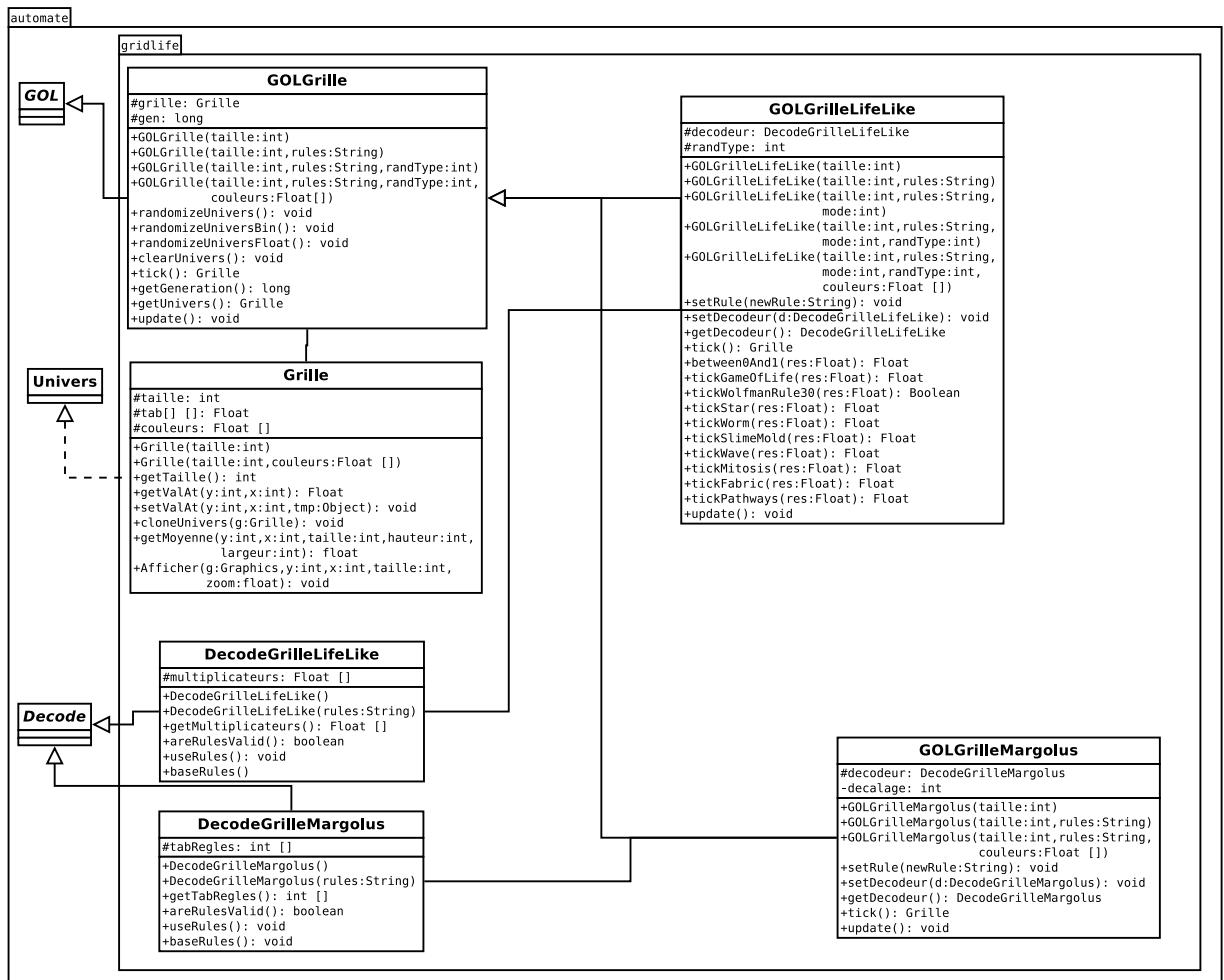


FIGURE 32 – Illustration du package automate/gridlife

Il contient aussi les classes **GOLGrilleLifeLike** et **GOLGrilleMargo-**

lus héritant de **GOLGrille** qui définissent et lancent les différentes versions des automates cellulaires et utilisent les différents décodeur respectivement les algorithmes neuraux et les voisins de Margolus.

### 6.1.2 package hashlife

Le package hashlife regroupent toutes les classes permettant d'initialiser le jeu de la vie avec des quadtrees. Il est composé des différentes classes héritant du package automate :

- La classe **GOLqTreeNode**
- La classe **GOLqTree**
- La classe **DecodeQtree**

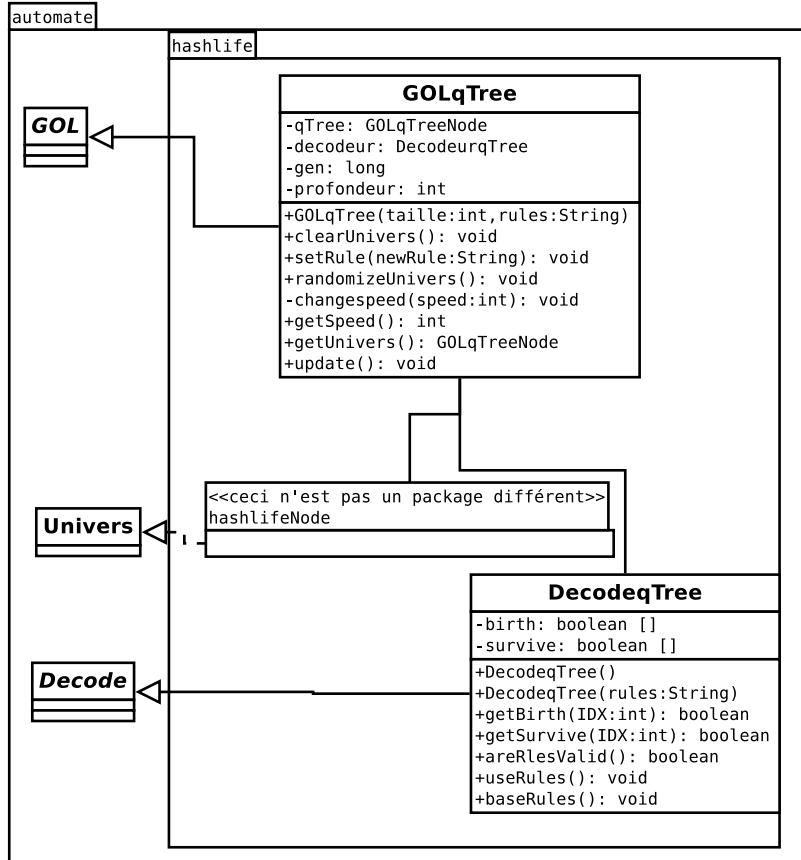


FIGURE 33 – illustration du package automate/hashlife

Il contient également les classes permettant de placer le quadtree dans une hashMap (**CanonicalizedqTree**), une autre classe implémentant Almost-Hashlife (**MemoizedqTree**) ainsi qu'une classe implémentant hashlife (**HashlifeqTree**).

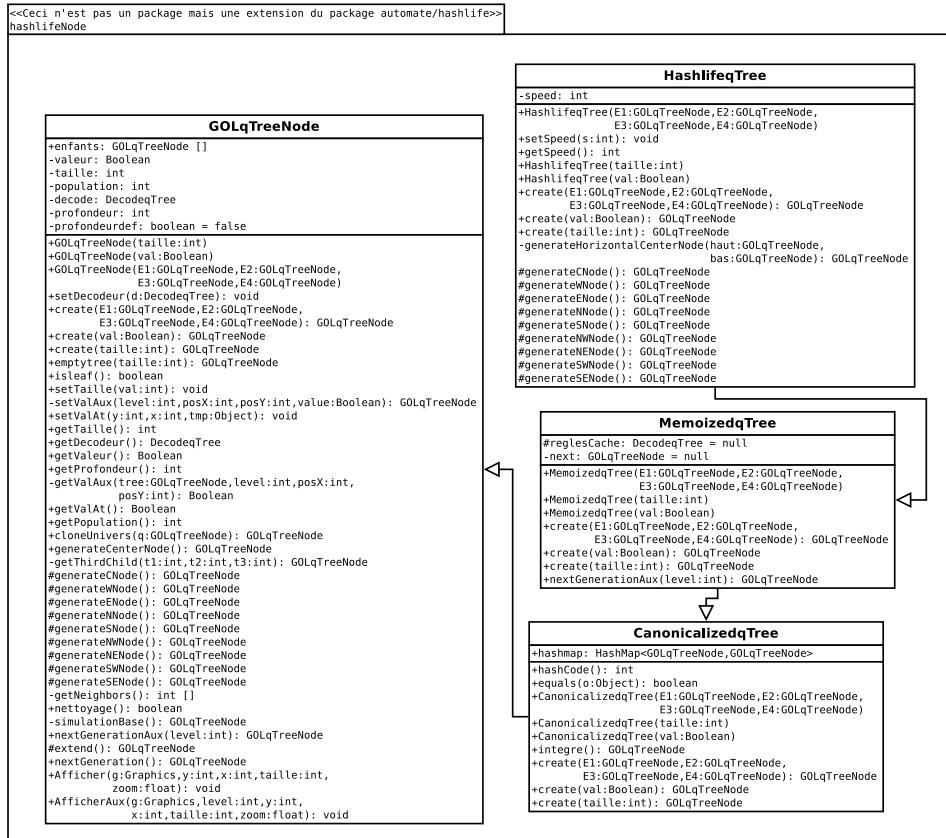


FIGURE 34 – Illustration de la partie hashlifeNode du diagramme précédent

## 6.2 Package Graphique

Le package graphique regroupe tout ce qui a un rapport avec l'interface comme les interaction entre le clavier et l'interface via la classe **Clavier** qui implémente l'interface KeyListener ou encore l'interface elle même (**InterfaceGameOfLife**) avec ces différents composants (**AffichagePrincipal**, **Menu**).

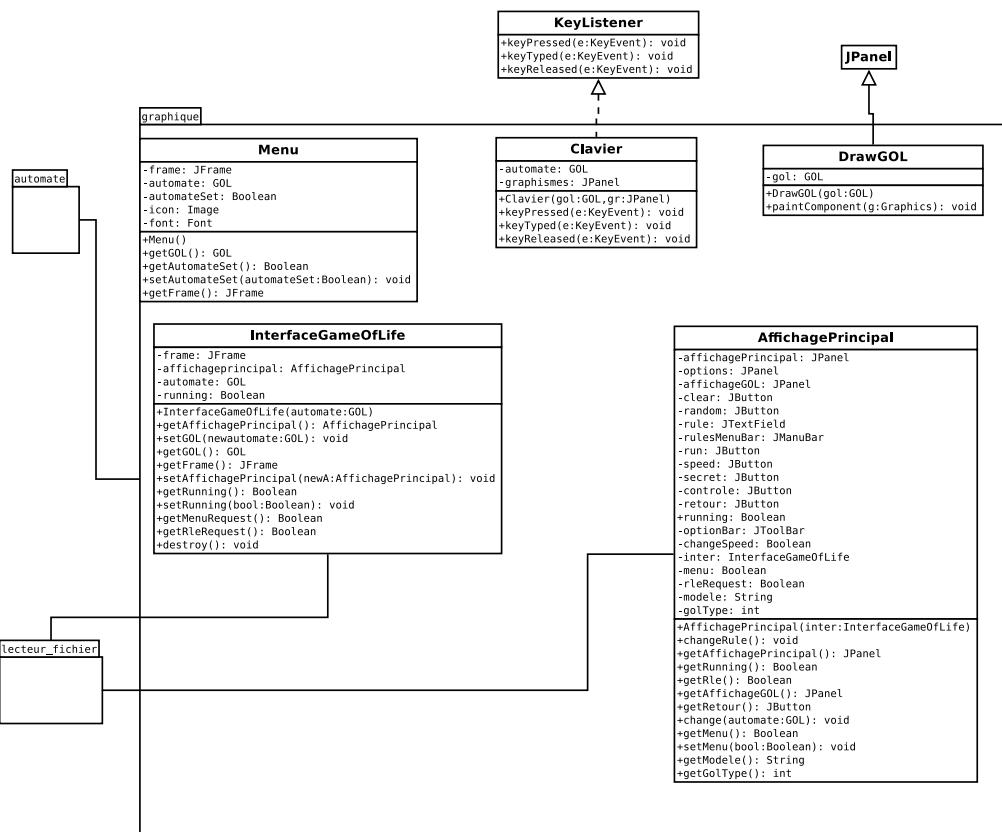


FIGURE 35 – Illustration du package graphique

### 6.3 Package Lecteur fichier

Le package lecteur fichier contient les classes permettant la lecture des différents fichiers en ressources.

La classe **Lecture CSV** permet de lire un fichier CSV et renvoie les informations dans une liste de tableaux de chaînes de caractères (un tableau de chaînes de caractères représentent une ligne du fichier CSV). Cette classe a aussi d'autres méthodes permettant de rechercher le code (ou mode ou type d'univers aléatoire (uniquement pour les instance de **GOLGrilleLifeLike**)) du jeu de la vie ou de l'automate cellulaire grâce aux noms (ou au code) du jeu de la vie ou de l'automate cellulaire.

La classe **LecteurRLE** permet de charger un modèle RLE sur une instance de **GOL** après l'avoir récupéré et décodé de l'archive.

Ce package va être réutilisé par les classes du package graphique ainsi que par le main.

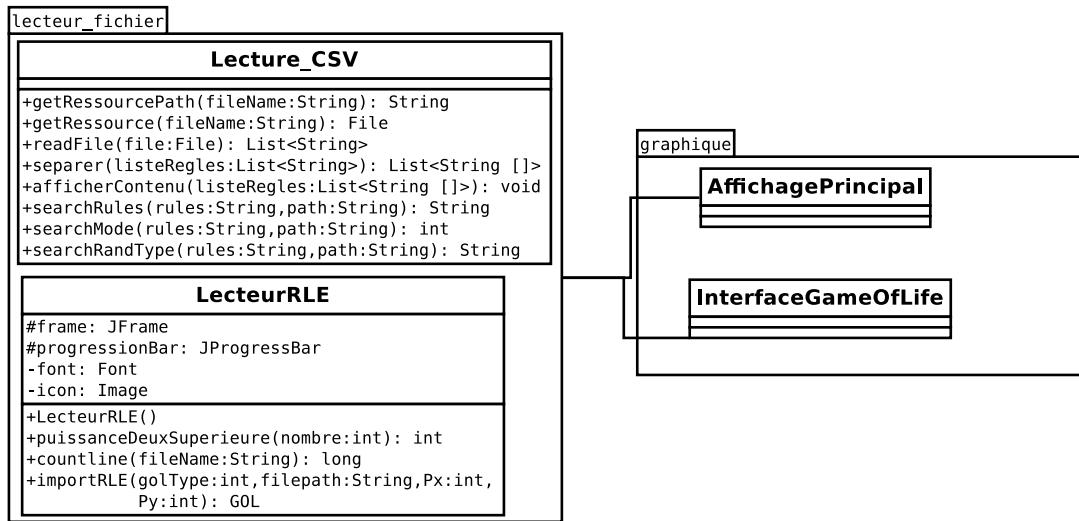


FIGURE 36 – illustration du package lecteur fichier

### 6.4 Package Test

Ce package contient tout les test unitaires du projet permettant de savoir la validité des méthodes utilisées dans les différentes classes des package automate et lecteur fichier.