Jeu de la Vie

YOCOLI Konan Jean Épiphane - Fotso Kamel - Li Jie

23 avril 2025

Table des matières

Introduction

Le Jeu de la Vie, créé en octobre 1970 par le mathématicien John Horton Conway, professeur à l'Université de Cambridge au Royaume-Uni, est un automate cellulaire fascinant qui ne nécessite aucun joueur. Malgré des règles simples, ce jeu génère des figures complexes et apparemment imprévisibles, comme l'a souligné son inventeur. Inspiré par des problèmes mathématiques profonds, le Jeu de la Vie est devenu une référence dans l'étude des systèmes complexes et a contribué à la célébrité de Conway.

Conway s'est d'abord intéressé à un problème posé par le mathématicien John Leech dans le domaine de la théorie des groupes, concernant l'empilement dense de sphères en 24 dimensions (connu sous le nom de réseau de Leech). En 1968, il publie des résultats remarquables sur ce sujet. Parallèlement, il explore un autre défi lancé dans les années 1940 par le célèbre mathématicien John von Neumann, qui cherchait à concevoir une machine hypothétique capable de s'auto-reproduire. Von Neumann avait développé un modèle mathématique complexe sur un repère cartésien, mais Conway simplifie ces idées en combinant ses travaux sur les réseaux de Leech avec ses recherches sur les machines auto-réplicatives. C'est ainsi qu'il donne naissance au Jeu de la Vie.

Dans le reste de ce document, nous présenterons le Jeu de la Vie, l'algorithme Hashlife, ainsi que notre travail sur le sujet.

1 Présentation du Jeu de la Vie

1.1 Les règles du jeu de la vie

Le Jeu de la Vie se déroule sur une grille infinie de cellules, chacune pouvant être dans un état *Vivant* ou *Mort*. Les règles qui régissent l'évolution des cellules sont simples :

— Une cellule morte avec exactement trois voisines vivantes devient vivante (reproduction).

— Une cellule vivante avec deux ou trois voisines vivantes reste vivante, sinon elle meurt (isolement ou surpopulation).

Malgré cette simplicité, le jeu engendre des motifs dynamiques et souvent surprenants, allant de structures stables à des formes oscillantes ou mobiles, comme le célèbre planeur (glider). Ces comportements ont captivé les mathématiciens, les informaticiens et les amateurs du monde entier.

1.2 Les structures dans le jeu de la vie

Il existe de nombreuses structures connues dans le Jeu de la Vie, que l'on peut diviser en deux groupes :

1.2.1 Les structures classiques

— **Les structures stables**
Les structures stables représentent un ensemble de cellules qui ont arrêté leur évolution. Ex : un bloc de quatre cellules est la plus petite structure stable possible.

— **Les structures périodiques, ou oscillateurs**
Les oscillateurs sont des structures périodiques prenant plusieurs états et revenant toujours à leur état initial. Des figures de ce type sont très nombreuses : on en connaît actuellement des centaines.

— **Les vaisseaux**
Les vaisseaux sont des structures capables d'"avancer", c'est-à-dire qu'après un certain nombre de générations, elles produisent une copie d'elles-mêmes, mais décalées dans l'univers.

— **Les mathusalems**
Un mathusalem est un petit objet qui, dans le jeu de la vie, finit par "exploser" en plusieurs objets stables après une longue évolution. Les plus intéressants sont ceux qui, partant d'une petite structure d'une dizaine de cellules, mettent beaucoup de temps à se stabiliser, révélant des comportements complexes.

1.2.2 Les autres structures

— **Les puffeurs**
Un puffeur est un objet mobile dans le jeu de la vie qui, à la différence d'un vaisseau, génère en se déplaçant des débris. Ces débris varient entre un seul type de débris et un large amas de débris variés.

— **Les canons**
Un canon (appelé *gun* dans la littérature anglophone) est un motif fini dont la structure centrale se répète de manière périodique, à la manière d'un oscillateur, et qui émet des vaisseaux à intervalles réguliers.

— **Les jardins d'Éden**
Un jardin d'Éden est un motif fini qui n'a aucun prédécesseur dans le jeu de la vie. Cela signifie qu'il n'existe aucune configuration initiale pouvant évoluer, en une ou plusieurs générations, pour donner naissance à ce motif. Par conséquent, un jardin d'Éden ne peut être obtenu que par une création manuelle.

— **Les spacefillers**
Un *spacefiller* (terme anglais signifiant "remplissage d'espace") est une structure qui s'étend de manière exponentielle en se développant à partir d'un agar (un oscillateur ou une structure stable, infinie et bidimensionnelle).

1.3 Les optimisations du jeu

1.3.1 Hashlife

Hashlife est un algorithme développé par Bill Gosper dans les années 1980 afin d'optimiser la vitesse de calcul des motifs du Jeu de la Vie. Cet algorithme repose sur l'utilisation d'une table de hachage et d'une structure de données appelée Quadtree, ce qui lui permet de traiter des configurations extrêmement complexes de manière très efficace, en particulier pour des motifs répétitifs ou de grande taille.

La structure Quadtree est un arbre dans lequel chaque nœud interne possède exactement quatre fils (ou aucun). Cette structure permet de diviser récursivement la grille (représentée par la racine de l'arbre) en quatre sous-grilles de taille égale (les nœuds fils). Cette subdivision se poursuit jusqu'à atteindre le niveau des cellules individuelles de la grille, qui correspondent aux feuilles de l'arbre.

La table de hachage, quant à elle, permet de stocker les motifs déjà calculés et de les restituer si un motif identique est rencontré à nouveau. Cela évite de recalculer des résultats déjà obtenus.

Grâce à cette approche, Hashlife exploite la redondance et la répétition des motifs dans le Jeu de la Vie, en mémorisant les résultats des calculs pour des sous-grilles identiques. Cela permet d'éviter des calculs redondants et d'accélérer considérablement la simulation, même pour des configurations de grande envergure ou sur de longues périodes.

1.3.2 Lenia

Lenia est davantage une évolution conceptuelle du Jeu de la Vie qu'une simple optimisation, mais nous verrons pourquoi cela est le cas dans un instant.

Lenia est un système de simulation d'automates cellulaires continus, développé par Bert Wang-Chak Chan, qui étend les concepts du Jeu de la Vie de John Conway à un espace continu et à des règles plus complexes. Contrairement au Jeu de la Vie, qui opère sur une grille discrète avec des cellules vivantes ou mortes, Lenia utilise des fonctions mathématiques lisses pour définir les états des cellules et leurs interactions, permettant ainsi la création de motifs organiques, fluides et souvent surprenants. Ces motifs, appelés "formes de vie", peuvent présenter des comportements dynamiques et évolutifs, ressemblant parfois à des organismes vivants microscopiques. Grâce à sa flexibilité et à sa richesse visuelle, Lenia a captivé la communauté scientifique et artistique, ouvrant de nouvelles perspectives dans l'étude des systèmes complexes et de la vie artificielle.

Lenia ne suit pas des règles fixes comme le Jeu de la Vie de Conway, mais repose plutôt sur un cadre mathématique flexible et paramétrable. Voici une explication simplifiée de son fonctionnement :

- **Grille continue** : Lenia opère sur un espace continu (par opposition à une grille discrète), où chaque point possède une valeur réelle comprise entre 0 (mort) et 1 (vivant), représentant l'état de la cellule.
- **Noyau de voisinage** : Chaque cellule est influencée par ses voisines via un noyau de convolution (ou kernel). Ce noyau définit comment les

cellules interagissent entre elles, avec une portée et une intensité paramétrables. Il est souvent représenté par une fonction gaussienne ou une autre forme lisse.

— **Fonction de croissance** : L'état futur d'une cellule est déterminé par une fonction de croissance, qui transforme la somme pondérée des interactions de voisinage (calculée via le noyau) en une nouvelle valeur d'état. Cette fonction est généralement une courbe en cloche ou une fonction lisse, permettant des transitions douces entre les états.

— **Mise à jour continue** : Les états des cellules sont mis à jour de manière continue dans le temps, contrairement au Jeu de la Vie qui opère par étapes discrètes. Cela permet des évolutions fluides et organiques des motifs.

— **Paramètres ajustables** : Lenia offre une grande flexibilité grâce à des paramètres tels que la taille du noyau, la forme de la fonction de croissance, la résolution de la grille et la vitesse de mise à jour. Ces paramètres permettent de générer une immense variété de "formes de vie" complexes et esthétiques.

En résumé, Lenia repose sur des interactions locales continues et des fonctions mathématiques lisses, ce qui permet de simuler des comportements dynamiques et émergents, souvent comparés à des organismes vivants microscopiques. Sa nature paramétrable en fait un outil puissant pour explorer les systèmes complexes et la vie artificielle.

2 Structuration du Projet

2.1 Description du projet

Dans le cadre de ce projet, nous avons pour objectif de concevoir une application en Java permettant d'implémenter le Jeu de la Vie ainsi que l'algorithme Hashlife, une méthode d'optimisation destinée à accélérer les calculs sur de grandes grilles, notamment lorsqu'elles présentent des motifs périodiques. L'application doit non seulement permettre de visualiser l'évolution des cellules dans le Jeu de la Vie, mais aussi offrir une interface flexible pour configurer les règles de comportement des cellules, choisir le type de voisinage, et ajuster d'autres paramètres personnalisables afin de rendre le simulateur le plus générique possible. Une seconde étape du projet consiste à intégrer l'algorithme Hashlife, afin d'améliorer significativement les performances lors de la simulation de grilles complexes et de grande taille.

2.2 Organisation du projet

Notre projet a été structuré selon une architecture modulaire en packages et classes. Voici l'organisation générale :

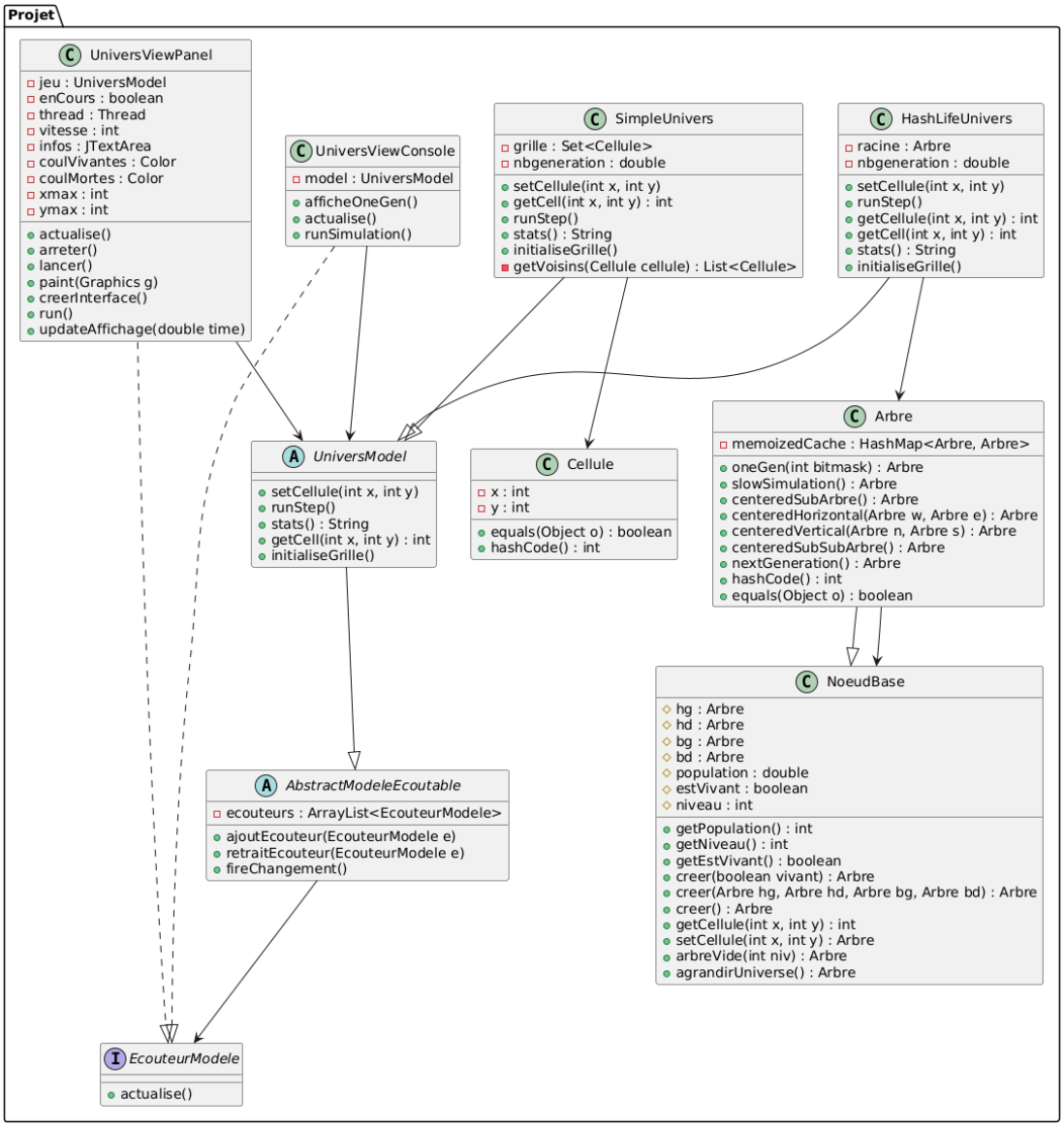


FIGURE 1 – Diagramme UML de l'architecture du projet

2.2.1 Structure du projet

Projet Contient l'ensemble des classes nécessaires à l'application
- `AbstractModeleEcoutable.java` - Classe abstraite pour les modèles écoutables
- `Arbre.java` - Gestion des arbres pour l'optimisation
- `Cellule.java` - Représentation d'une cellule dans la grille
- `EcouteurModele.java` - Interface pour les écouteurs de modèle
- `HashLifeUnivers.java` - Implémentation de l'algorithme Hashlife
- `Main.java` - Point d'entrée de l'application
- `MainSwing.java` - Interface utilisateur avec Swing
- `ModelisationUml.dia` - Diagramme UML du projet
- `NoeudBase.java` - Classe de base pour les nœuds de l'arbre
- `pattern.rle` - Fichier de motifs pour la simulation
- `SimpleUnivers.java` - Implémentation simple de l'univers
- `UniversModel.java` - Modèle abstrait de l'univers
- `UniversViewConsole.java` - Affichage console de l'univers
- `UniversViewPanel.java` - Affichage graphique de l'univers

2.2.2 Principales fonctionnalités implémentées

- Système de grille modifiable dynamiquement
- Gestion des états des cellules (vivant/mort)
- Calcul du voisinage selon différents modes
- Contrôle de la simulation (play/pause/step)
- Visualisation des performances (temps de calcul)

3 Conception et Développement

Notre travail a été subdivisé en trois grandes parties. Dans un premier temps, nous avons implémenté le jeu de la vie « simple ». Ensuite, nous avons développé l'algorithme Hashlife. Enfin, nous avons réalisé diverses optimisations telles que la fusion des univers Simple et Hashlife, la mise en place des tests, ainsi que l'implémentation de l'architecture MVC (Modèle-Vue-Contrôleur).

3.1 Jeu de la vie « Simple »

Pour cette première version, nous avons utilisé la structure suivante :

```
.
|-- Cellule.java
|-- Grille.java
|-- Jeu.java
|-- UtilitaireConsole.java
```

avec :
- `Cellule.java` : représente l'état d'une cellule (vivante ou morte) ;

— `Grille.java` : représente l'ensemble des cellules. Elle permet d'initialiser le tableau avec la structure de départ et implémente les règles du jeu de la vie au sens « littéral » pour calculer la génération suivante ;
— `Jeu.java` : lance le jeu ;
— `UtilitaireConsole.java` : améliore l'affichage en console.

Ce point a constitué la base de notre projet et nous a permis de construire la suite sur des fondations solides.

3.2 L'algorithme Hashlife

Comme présenté, Hashlife est une version optimisée du jeu de la vie, basée sur une structure en quadtree. Pour cela, nous avons utilisé la structure suivante :

```
.
|-- Arbre.java
|-- HashLifeUnivers.java
|-- Main.java
|-- NoeudBase.java
|-- Univers.java
|-- pattern.rle
```

avec :

— `NoeudBase.java` : représente la logique d'un nœud de base dans un quadtree. Un nœud possède soit quatre enfants (une sous-grille), soit c'est une feuille (une cellule). Il permet de créer des nœuds, d'obtenir leur population (si interne) ou leur état (si feuille), et de modifier ou consulter une cellule à des coordonnées données. Il contient aussi deux fonctions importantes : l'une pour créer un arbre vide, l'autre pour l'agrandir.
— `Arbre.java` : représente la grille et implémente les règles du jeu ainsi que la gestion des tables de hachage. Le calcul de la génération suivante s'effectue via la méthode `nextGeneration()`, qui repose sur une approche adaptée aux arbres.

Algorithm 1: Fonction `nextGeneration()`

Input: *this*, *memoizedCache*, *population*, *niveau*
Output: *nextGen*

if *this est dans memoizedCache* **then**
  return $memoizedCache[this]$
Déclarer *nextGen*;
if $population == 0$ **then**
  $nextGen \leftarrow hg$ `// Retourne un quadrant vide`
else if $niveau == 2$ **then**
  $nextGen \leftarrow slowSimulation()$ `// Calcule manuellement la prochaine génération`
else
  $n00 \leftarrow hg.centeredSubArbre()$;
  $n01 \leftarrow centeredHorizontal(hg, hd)$;
  $n02 \leftarrow hd.centeredSubArbre()$;
  $n10 \leftarrow centeredVertical(hg, bg)$;
  $n11 \leftarrow centeredSubSubArbre()$;
  $n12 \leftarrow centeredVertical(hd, bd)$;
  $n20 \leftarrow bg.centeredSubArbre()$;
  $n21 \leftarrow centeredHorizontal(bg, bd)$;
  $n22 \leftarrow bd.centeredSubArbre()$;
  $fA \leftarrow Async(creer(n00, n01, n10, n11).nextGeneration())$;
  $fB \leftarrow Async(creer(n01, n02, n11, n12).nextGeneration())$;
  $fC \leftarrow Async(creer(n10, n11, n20, n21).nextGeneration())$;
  $fD \leftarrow Async(creer(n11, n12, n21, n22).nextGeneration())$;
  $nextGenA \leftarrow fA.join()$;
  $nextGenB \leftarrow fB.join()$;
  $nextGenC \leftarrow fC.join()$;
  $nextGenD \leftarrow fD.join()$;
  $nextGen \leftarrow creer(nextGenA, nextGenB, nextGenC, nextGenD)$;
$memoizedCache[this] \leftarrow nextGen$;
return *nextGen*;

— **Vérification du cache** : si l'arbre courant (*this*) est présent dans le cache, on retourne directement le résultat mémorisé pour éviter des calculs inutiles.
— **Cas de base** :
  — Si la population est nulle, on retourne simplement *hg* (le quadrant haut-gauche, vide).
  — Si le niveau est 2 (grille 4×4), on utilise `slowSimulation()` pour simuler directement la génération suivante.

Algorithm 2: Fonction `slowSimulation()`

```
   Output: nextGen
1  allbits ← 0;
2  for y ← −2 to 1 do
3  |  for x ← −2 to 1 do
4  |  |  allbits ← allbits ≪ 1;
5  |  |  allbits ← allbits + getCellule(x, y);
6  cellule1 ← oneGen(allbits ≫ 5);
7  cellule2 ← oneGen(allbits ≫ 4);
8  cellule3 ← oneGen(allbits ≫ 1);
9  cellule4 ← oneGen(allbits);
10 return creer(cellule1, cellule2, cellule3, cellule4);
```

Algorithm 3: Fonction `oneGen`(bitmask : entier)

```
   Input: bitmask
   Output: État de la cellule
1  if bitmask == 0 then
2  |  return creer(false) // Toutes les cellules sont mortes
3  self ← (bitmask ≫ 5)&1;
4  bitmask ← bitmask&0x757;
5  neighborCount ← 0;
6  while bitmask ≠ 0 do
7  |  neighborCount ← neighborCount + 1;
8  |  bitmask ← bitmask&(bitmask − 1);
9  if neighborCount == 3 or (neighborCount == 2 and self == 1) then
10 |  return creer(true)
11 else
12 |  return creer(false)
```

La méthode `slowSimulation()` calcule la génération suivante pour un petit arbre (niveau 2) de 4×4 cellules. Elle utilise une approche directe basée sur un masque binaire :

— Elle parcourt les 16 cellules autour du centre (coordonnées relatives de −2 à 1) ;
— Elle construit un entier *allbits* représentant l'état binaire de chaque cellule ;
— Ce masque est découpé en 4 sous-grilles 3×3 :
  — Haut-Gauche (HG) ;
  — Haut-Droit (HD) ;
  — Bas-Gauche (BG) ;
  — Bas-Droit (BD).
— Chaque sous-grille est évaluée via la fonction `oneGen()`, qui applique les règles de Conway.

Dans le cas récursif (niveau > 2) :

  - L'arbre est divisé en 9 sous-arbres via les fonctions d'extraction centrée ;
  - Les 4 quadrants principaux sont calculés en parallèle via `CompletableFuture` ;
  - Une fois les calculs terminés, les résultats sont assemblés en une nouvelle génération ;
  - Enfin, le résultat est mis en cache avant d'être retourné.
- `Univers.java` : est l'interface permettant de créer un Univers de Jeu ici HashlifeUnivers, plus tard on ajoutera SimpleUnivers.
- `HashLifeUnivers.java` : est "l'orchestreur" du jeu de la vie utilisant l'algorithme Hashlife et créant une grille de type Arbre.
- `pattern.rle` : contient la configuration de départ du jeu.
- `Main.java` : est le fichier principal et contient la fonction de lecture du pattern.rle.

4 Les optimisations

4.1 Fusion des univers Simple et Hashlife

Afin de permettre aux utilisateurs d'expérimenter différentes implémentations du jeu de la vie, notre projet intègre deux implémentations du jeu de la vie : *SimpleUnivers* et *HashLifeUnivers*. Comme expliqué plus tôt, la première est une implémentation basée sur une grille commune avec une structure simple et facile à comprendre, tandis que la seconde, bien que légèrement plus complexe, utilise un quadtree et la mémorisation pour accélérer le processus d'évolution cellulaire.

Objectif de la fusion. Notre objectif en intégrant les deux implémentations différentes est de permettre aux deux implémentations d'être utilisées dans la même interface graphique avec la même logique de fonctionnement.

Stratégie. Pour ce faire, nous faisons en sorte que `simpleUnivers` et `HashlifeUnivers` héritent d'une classe abstraite appelée `UniversModel`, dans laquelle certaines méthodes uniformes sont définies, par exemple `setCellue`, `getCellule`,`runStep`. Nous pouvons ainsi utiliser la classe `UniversModel` pour représenter les différentes implémentations.

4.2 les tests unitaire

Dans ce projet, nous avons mis en place quelques tests unitaires pour s'assurer que les classes principales du simulateur du Jeu de la Vie fonctionnent correctement. Les tests ont été réalisés avec JUnit 5, un outil courant pour tester en Java. Nous avons utilisé des annotations telles que `@BeforeEach` et `@Test`, ainsi que des assertions simples comme `assertEquals` ou `assertTrue` pour vérifier les résultats attendus.

1. Ce que nous avons testé :

— `NoeudBase` : nous avons testé les méthodes `getPopulation`, `getNiveau`, `getEstVivant`, `setCellule`, `arbreVide` et `agrandirUniverse`.
— `Cellule` : nous avons testé `equals` et `hashCode`, pour vérifier si les cellules sont bien reconnues dans une collection.
— `Arbre` : nous avons testé `oneGen`, `slowSimulation`, `centeredSubArbre`, `nextGeneration`, `equals`, et `hashCode` pour s'assurer que les arbres évoluent correctement et se comparent de manière appropriée.
— `SimpleUnivers` : nous avons testé l'ajout de cellules, l'évolution d'une génération, et les statistiques.
— `HashLifeUnivers` : similaire à `SimpleUnivers`, mais avec une version optimisée utilisant un quadtree.
— `AbstractModeleEcoutable` : nous avons testé l'ajout et le retrait d'écouteurs, ainsi que la bonne transmission des notifications.

Ces tests nous ont aidés à vérifier que le code fonctionnait comme prévu, à identifier les bugs plus rapidement et à maintenir un projet plus solide tout au long du développement.

4.3 M-VC (Modèle - Vue/Contrôleur)

En accord avec la structure du jeu, plusieurs objectifs sont apparus au cours de la phase de modélisation.
`Le premier objectif` est de concevoir un simulateur modulable, car celui-ci est amené à évoluer, notamment en ce qui concerne la structure de données. Il est donc essentiel de rendre l'architecture claire et compréhensible, afin que toute personne puisse facilement suivre le raisonnement adopté lors de sa conception.
`Le deuxième objectif` est d'isoler les structures de données du reste du simulateur. Le but est de permettre au cœur du simulateur de fonctionner indépendamment de l'affichage, afin de pouvoir modifier ou remplacer les interfaces existantes (comme le terminal ou l'interface graphique Swing) sans affecter la logique interne.
`Le troisième objectif` est de permettre à la fois l'affichage du simulateur et l'interaction avec celui-ci. Cependant, certaines interfaces comme le terminal ne permettent aucune interaction directe avec la simulation (par exemple, cliquer sur une cellule pour la modifier), une fois celle-ci lancée.
Pour répondre à ces trois objectifs, nous avons adopté une architecture M-VC (Modèle - Vue/Contrôleur), dans laquelle la Vue joue également le rôle de Contrôleur. Elle observe le modèle, affiche son état, et déclenche certaines actions (comme faire avancer le simulateur) sans passer par une classe Contrôleur distincte, qui n'aurait pas pu être appliquée de façon cohérente à toutes les interfaces. Ce modèle architectural permet ainsi de construire un simulateur :
— Modulable et évolutif,
— Indépendant de l'affichage et des moyens de contrôle,
— Avec une claire séparation entre les données et l'interface utilisateur.

Composants du M-VC
— **Le Modèle :** HashlifeUnivers ou SimpleUnivers

- Contient l'état de la grille (cellules vivantes ou mortes).
- Gère l'évolution de la simulation (génération suivante).
- Notifie toutes les vues enregistrées via ajoutEcouteur(EcouteurModele) lorsqu'un changement a lieu (par exemple après un appel à runStep()).

— **La Vue / Contrôleur :** UniversViewPanel ou UniversViewConsole
- Affiche la grille en mode graphique (Swing) ou en mode texte (console).
- Joue également le rôle de contrôleur en déclenchant des actions (comme faire avancer la simulation).
- Observe le modèle via le pattern observateur.
- Appelle les méthodes du modèle.
- Se met à jour automatiquement quand le modèle notifie un changement, via la méthode actualise().

Interaction entre les composants

1. L'utilisateur lance la simulation (ex. : appuie sur un bouton ou une touche).
2. La Vue-Contrôleur intercepte l'événement.
3. Elle appelle une méthode du modèle (comme runstep()).
4. Le modèle met à jour l'état interne de la grille et notifie les vues via fireChangement().
5. La vue reçoit la notification et actualise l'affichage grâce à la méthode actualise().

5 Expérimentations et usages

Notre implémentation permet de visualiser plusieurs structures classiques du Jeu de la Vie à travers différents motifs prédéfinis :

— **Planeur (Glider)** - La structure mobile la plus emblématique qui se déplace en diagonale
— **Canon à planeurs (Glider Gun)** - Structure périodique générant des planeurs
— **Pulsar** - Oscillateur de période 3 particulièrement visuel
— **Planeur 2** - Variante du planeur classique

L'interface utilisateur (Fig. 2) permet de sélectionner facilement ces motifs via un menu déroulant. Chaque sélection charge le fichier RLE correspondant (`pattern.rle`, `pattern.rle1`, etc.) et initialise la grille avec la configuration choisie.

Ces expérimentations démontrent la capacité de notre implémentation à gérer :

— Les structures mobiles (planeurs)
— Les générateurs périodiques (canon)
— Les oscillateurs complexes (pulsar)
— Le chargement et l'interprétation correcte des fichiers RLE

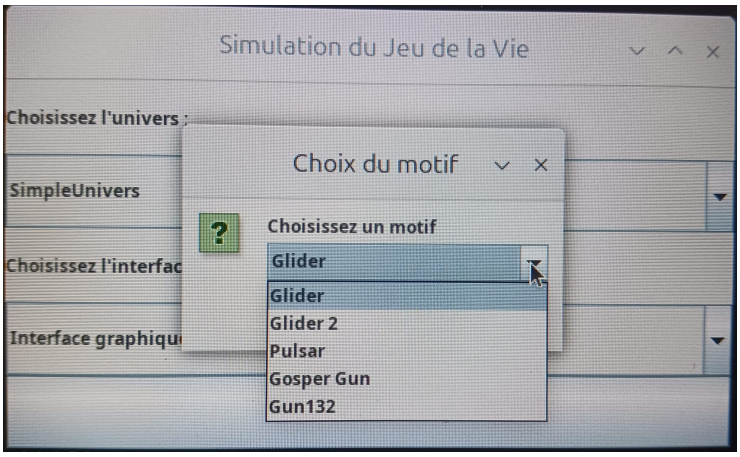
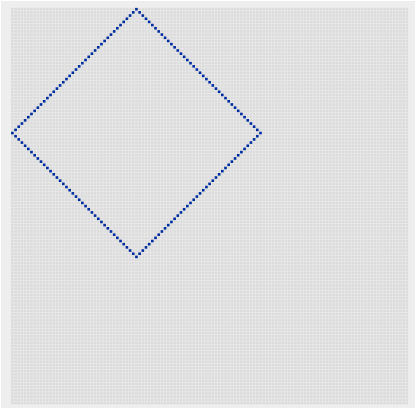
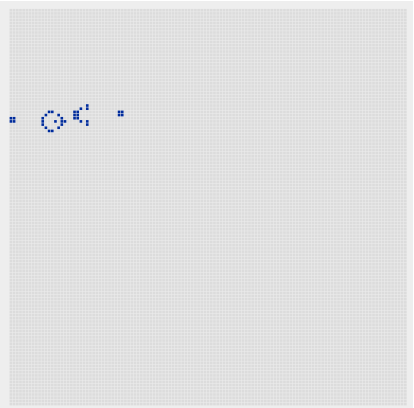


FIGURE 2 – Interface de sélection des motifs avec aperçu des structures disponibles

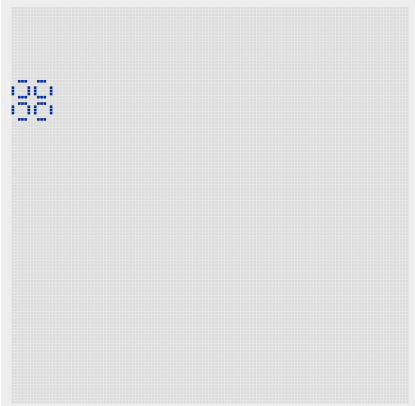
Le système de sélection par menu, couplé à la visualisation en temps réel, offre une excellente base pour explorer d'autres motifs complexes et observer leurs interactions.



(a) Planeur en mouvement après 10 générations



(b) Canon à planeurs produisant sa première salve



(a) Pulsar en milieu de cycle d'oscillation

(b) Planeur 2 montrant sa trajectoire caractéristique

Conclusion

Ce projet sur le Jeu de la Vie a été une expérience extrêmement enrichissante, tant sur le plan technique que théorique. En implémentant à la fois une version simple et une version optimisée avec l'algorithme HashLife, nous avons pu explorer les mécanismes fascinants des automates cellulaires et mesurer l'impact des optimisations sur les performances.

L'architecture MVC nous a permis de structurer notre code de manière modulaire, facilitant ainsi l'intégration des différentes fonctionnalités et la gestion des interactions entre les composants. Les tests unitaires nous ont aidés à garantir la robustesse du code, tandis que l'interface graphique a rendu la simulation visuelle et interactive.

Cependant, certains défis sont restés présents, comme la gestion de la mé-

moire pour de très grandes grilles ou l'optimisation fine de HashLife. Ces difficultés ouvrent des pistes d'amélioration, comme l'intégration d'autres algorithmes (par exemple Lenia) ou l'ajout de fonctionnalités avancées (zooms dynamiques, sauvegarde de motifs).

Enfin, ce projet a renforcé notre compréhension des systèmes complexes et de l'importance de l'optimisation algorithmique. Il a également souligné l'équilibre délicat entre simplicité des règles et complexité des comportements émergents, un concept au cœur du Jeu de la Vie.

Bibliographie

1. **Science Étonnante**. (2025). *Le Jeu de la Vie* [Vidéo YouTube]. Disponible à : `https://www.youtube.com/watch?v=S-W0NX97DB0`
2. **Wikipedia**. (2025). *Jeu de la Vie de Conway*. Consulté en 2025. Disponible à : `https://fr.wikipedia.org/wiki/Jeu_de_la_vie`
3. **LifeWiki**. (2025). *ConwayLife Wiki*. Consulté en 2025. Disponible à : `https://conwaylife.com/wiki/`
4. **Wikipedia**. (2025). *Hashlife*. Consulté en 2025. Disponible à : `https://en.wikipedia.org/wiki/Hashlife`
5. Gosper, B. (1984). *Exploitation de la redondance dans le Jeu de la Vie : l'algorithme Hashlife*. Dans *Actes du Symposium sur les Automates Cellulaires*.
6. Chan, B. W. C. (2018). *Lenia : Une nouvelle génération d'automates cellulaires continus. ArXiv*. Disponible à : `https://arxiv.org/abs/1812.05433`
7. Conway, J. H. (1970). *The Game of Life. Scientific American*, 223(4), 4-5.
8. Gardner, M. (1970). *Mathematical Games : The Fantastic Combinations of John Conway's New Solitaire Game "Life". Scientific American*, 223(4), 120-123.