Projet : Implémentation du Jeu de la Vie en JavaFX

AMOUSSOU Mensanh Boris
L2, Groupe 2B Informatique – Université de Caen

16 avril 2025

Table des matières

1 Introduction

1.1 Description générale du projet

Le Jeu de la Vie est un automate cellulaire inventé par le mathématicien britannique John Horton Conway en 1970. Il s'agit d'un modèle qui simule la vie d'un ensemble de cellules selon des règles simples. Chaque cellule peut être soit vivante, soit morte, et son état évolue au cours du temps en fonction du nombre de ses voisins vivants. Le projet consiste à implémenter cette simulation sous forme d'une interface graphique utilisant JavaFX, permettant à l'utilisateur d'interagir avec la grille et de voir l'évolution des cellules au fil du temps.

1.2 Identification de la problématique

La principale problématique réside dans l'implémentation des règles du jeu et l'interaction avec l'interface graphique. En effet, il est nécessaire de bien gérer l'évolution de la grille. La mise à jour de la grille doit être rapide tout en respectant les règles de John Conway.

2 Structuration du projet

2.1 Description des besoins du projet

Le projet nécessite la création d'une grille de cellules, avec la possibilité de définir l'état initial des cellules (vivantes ou mortes). La grille doit évoluer selon les règles du Jeu de la Vie, mises à jour régulieres. Une interface graphique pour afficher cette grille et permettre à l'utilisateur d'interagir avec elle.

2.2 Fonctionnalités implémentées

Les principales fonctionnalités implémentées sont les suivantes :

1. **Initialisation de la grille :** La grille est initialisée avec des cellules mortes, et l'utilisateur peut activer ou désactiver certaines cellules pour définir l'état initial.

2. **Simulation du jeu :** L'état de la grille évolue en fonction des règles du Jeu de la Vie (comptage des voisins et application des règles de John Conway).

3. **Interface utilisateur :** Une interface graphique permet de visualiser la grille et d'interagir avec elle.

4. **Animation :** L'évolution de la grille est animée à l'aide de la classe Timeline de JavaFX, avec un intervalle de mise à jour prédéfini à 500 ms.

2.3 Organisation du projet

Le projet a été divisé en trois classes principales :

1. **JeuDeLaVieModel :** Gère la logique du jeu, y compris les calculs des voisins et cellules vivantes.

2. **MaGrilleView :** Gère l'affichage de la grille et l'interface graphique.

3. **MaGrilleController :** Gère l'interaction entre JeuDeLaVieModel et MaGrilleView, en contrôlant le déroulement de la simulation.

3 Éléments techniques

3.1 Description des structures de données

La grille est un tableau à deux dimensions contenant des objets(cellules) de type Color[][], où chaque cellule peut être soit vivante (représentée par la couleur verte), soit morte (représentée par la couleur beige). La grille est de taille fixe (par défaut 25x25), mais cette taille peut être modifiée.

3.2 Description des algorithmes

Les algorithmes principaux du projet sont :

1. **Calcul des voisins vivants :** Pour chaque cellule, on compte le nombre de cellules voisines vivantes en tenant compte de la grille cyclique (les bords se rejoignent). Cet algorithme est implémenté dans la méthode `compterVoisin` dans la classe JeuDeLaVieModel.

```
22      // Méthode de calcul des voisins
23  @   public int compterVoisin(Color[][] grille, int i, int j, int n) {  1 usage
24          int voisins = 0;
25          // Haut
26          if (grille[i][(j - 1 + n) % n] == couleurVivante) voisins++;
27          // Bas
28          if (grille[i][(j + 1) % n] == couleurVivante) voisins++;
29          // Gauche
30          if (grille[(i - 1 + n) % n][j] == couleurVivante) voisins++;
31          // Droite
32          if (grille[(i + 1) % n][j] == couleurVivante) voisins++;
33          // Haut-Gauche
34          if (grille[(i - 1 + n) % n][(j - 1 + n) % n] == couleurVivante) voisins++;
35          // Haut-Droite
36          if (grille[(i + 1) % n][(j - 1 + n) % n] == couleurVivante) voisins++;
37          // Bas-Gauche
38          if (grille[(i - 1 + n) % n][(j + 1) % n] == couleurVivante) voisins++;
39          // Bas-Droite
40          if (grille[(i + 1) % n][(j + 1) % n] == couleurVivante) voisins++;
41          return voisins;
42      }
43
```

FIGURE 1 – implémentation de la méthode compterVoisin

2. **Mise à jour de la grille :** À chaque génération, l'état de chaque cellule est mis à jour en fonction des règles du jeu (sous-population, sur-population, reproduction). Cette opération est réalisée dans la méthode `MiseAJour` dans la classe MaGrilleView.

```java
// Méthode pour mettre à jour la grille
public void MiseAJour(Color[][] grille, GridPane gridPane, int n) {  1 usage
    Color[][] grilleSuivante = new Color[n][n];
    boolean stable = true;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            int voisins = model.compterVoisin(grille, i, j, n);
            if (grille[i][j] == model.getCouleurVivante()) {
                grilleSuivante[i][j] = (voisins < 2 || voisins > 3) ? model.getCouleurMorte() : model.getCouleurVivante();
            } else {
                grilleSuivante[i][j] = (voisins == 3) ? model.getCouleurVivante() : model.getCouleurMorte();
            }
            if (!grilleSuivante[i][j].equals(grille[i][j])) {
                stable = false;
            }
        }
    }
```

FIGURE 2 – une partie de la méthode MiseAJour

3. **Nombre de cellules vivantes :** À chaque génération, le nombre de cellules vivantes est compté. Cette opération est réalisée dans la méthode `celluleVivante` dans la classe JeuDeLaVieModel.

```java
44      // Méthode pour compter les cellules vivantes
45      public int celluleVivante(Color[][] grille, int n) {  1 usage
46          int count = 0;
47          for (int i = 0; i < n; i++) {
48              for (int j = 0; j < n; j++) {
49                  if (grille[i][j] == couleurVivante) {
50                      count++;
51                  }
52              }
53          }
54          return count;
55      }
```

FIGURE 3 – implémentation de la méthode celluleVivante

4 Architecture du projet

4.1 Diagrammes des classes

Le projet est organisé en trois classes principales, qui interagissent de manière suivante :

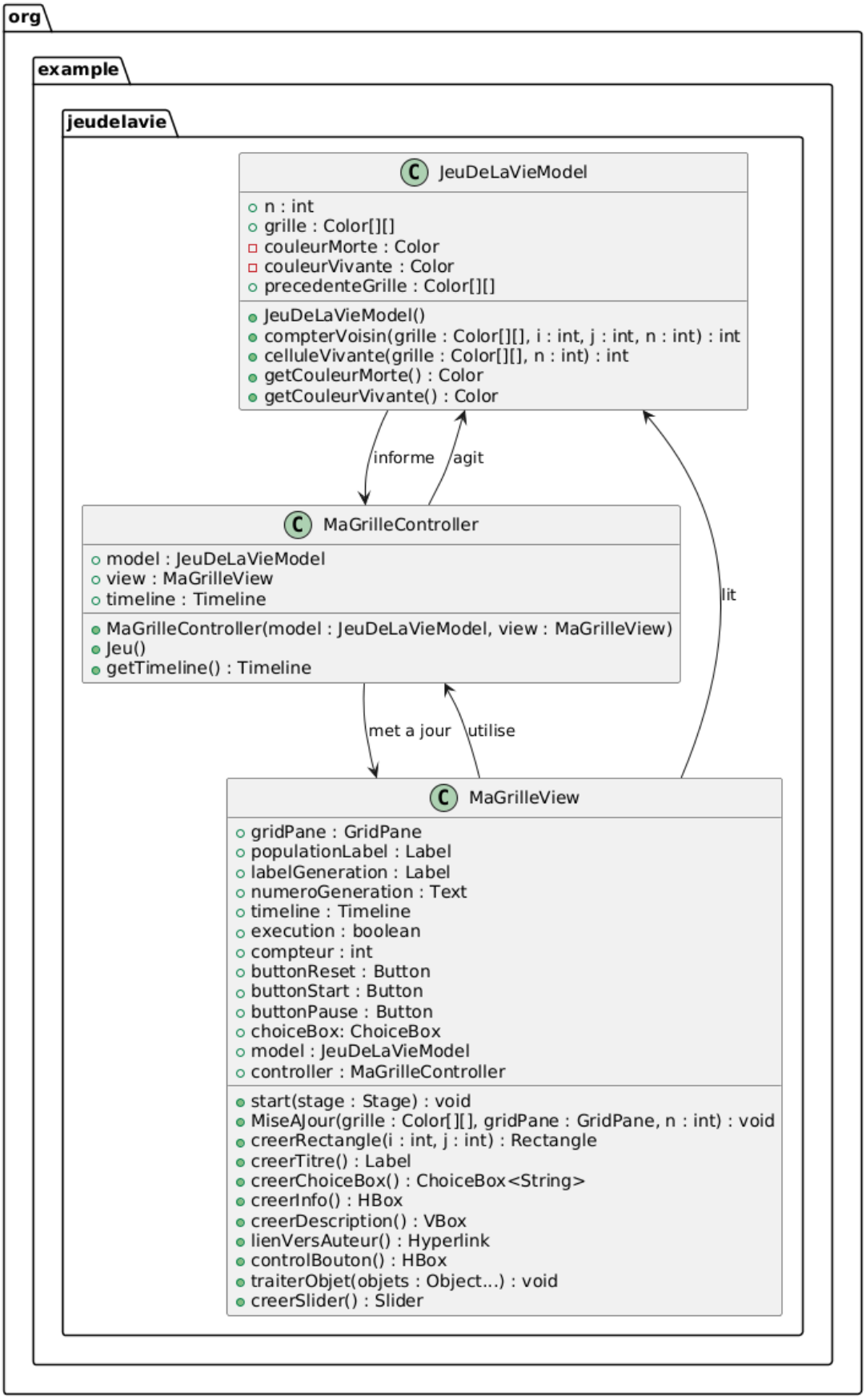

FIGURE 4 – Le Diagramme des classes commenté

4.2 Chaînes de traitement (comment les classes interagissent et pourquoi)

Mon projet suit une structure inspirée du modèle MVC (Modèle - Vue - Contrôleur), afin de séparer la logique du jeu, l'affichage graphique, et la gestion du déroulement du jeu.

- **La classe** `JeuDeLaVieModel` représente le **modèle**. Elle contient les données principales du jeu, notamment la grille (`grille`) et sa taille (`n`).
- **La classe** `MaGrilleView` représente la **vue**. Elle est responsable de l'affichage graphique. Elle possède la méthode `MiseAJour` qui permet de mettre à jour l'affichage à partir des données du modèle.
- **La classe** `MaGrilleController` représente le **contrôleur**. Elle fait le lien entre le modèle et la vue. Elle utilise le `Timeline` (animation JavaFX) pour actualiser le jeu automatiquement toutes les 500 millisecondes. À chaque intervalle, la méthode `Jeu()` est appelée, ce qui permet de transmettre les données du modèle à la vue pour mettre à jour l'affichage.

5 Expérimentations et usages

5.1 Cas d'utilisation

- **Simulation du Jeu :** L'utilisateur peut observer l'évolution de la grille en appuyant sur le bouton "Start", qui lance la simulation. La grille évolue automatiquement selon les règles du Jeu de la Vie.

- **Mettre en pause la simulation :** L'utilisateur peut mettre en pause le jeu en appuyant sur le bouton "Pause".

- **Réinitialiser la grille :** L'utilisateur peut vider la grille (mettre toutes les cellules en état mort) en appuyant sur le bouton "Reset".

- **Modification manuelle de la grille :** L'utilisateur peut cliquer sur les cellules de la grille pour les rendre vivantes ou mortes avant de démarrer la simulation.

- **Modification aléatoire de la grille :** L'utilisateur peut rendre les cellules vivantes de façon aléatoire en choisissant le motif "Aléatoire" du choicebox avant de démarrer la simulation.

- **Gestion de la vitesse :** L'utilisateur peut contrôler la vitesse de la simulation grace au slider.

5.2 Quelques images de la simulation en cours

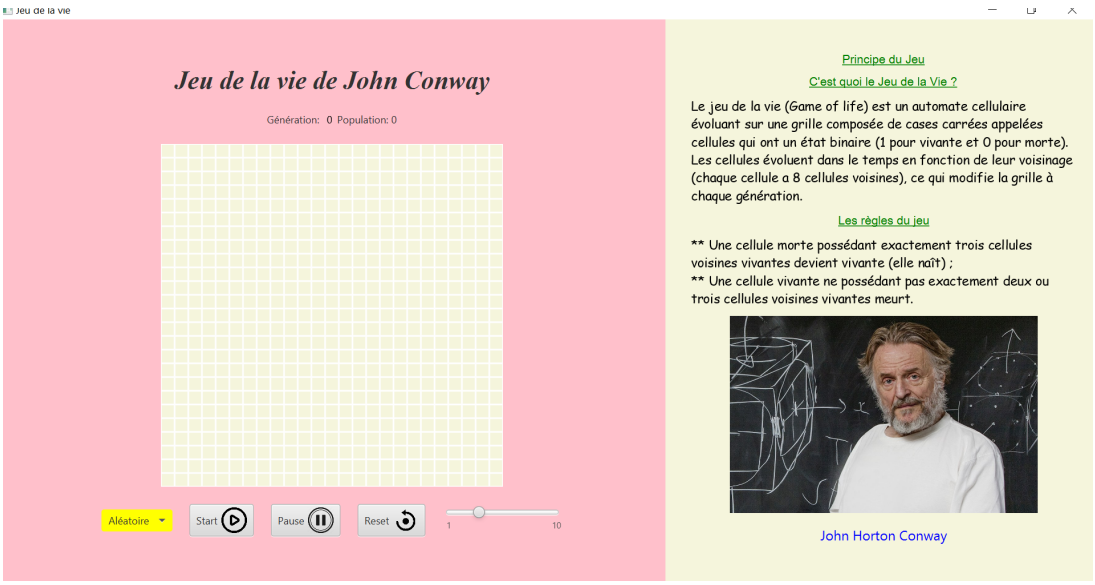

FIGURE 5 – Interface utilisateur du jeu

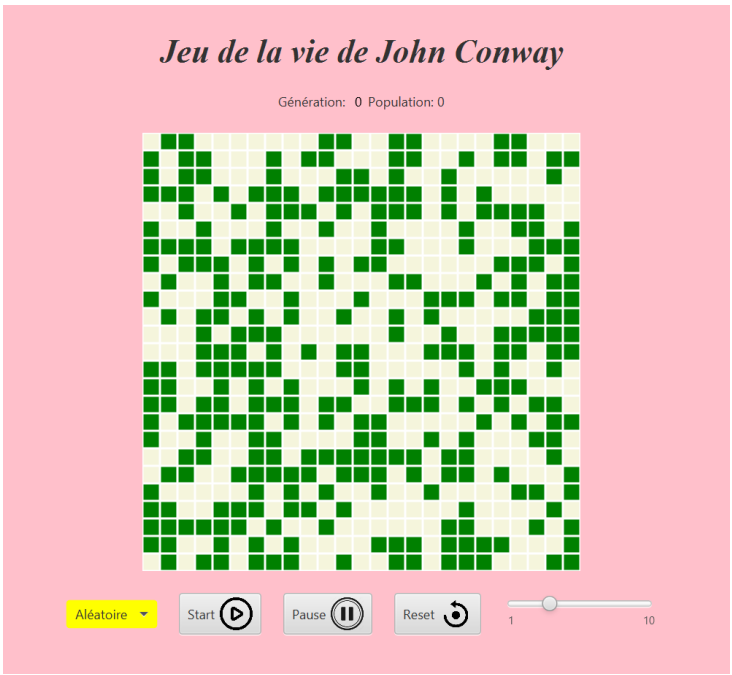

FIGURE 6 – Cellules remplies avec le motif aléatoire

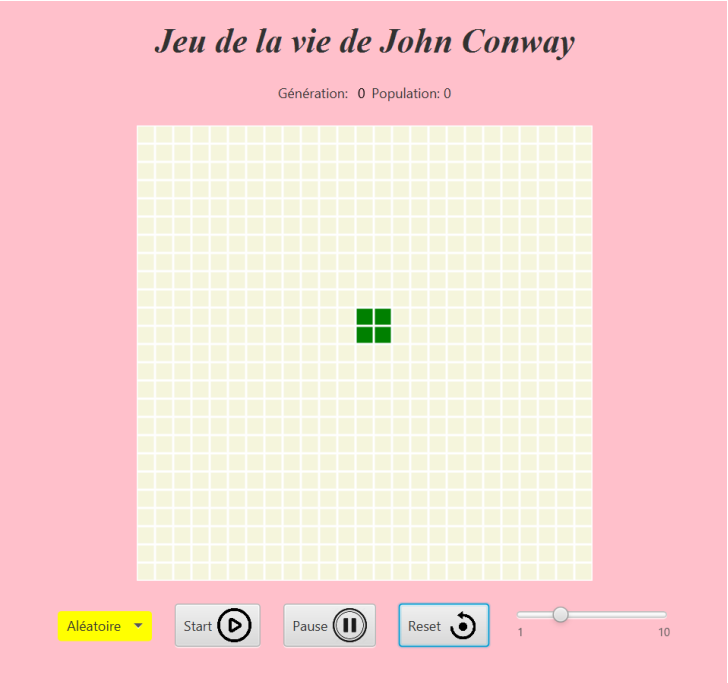

FIGURE 7 – Jeu à l'état stable

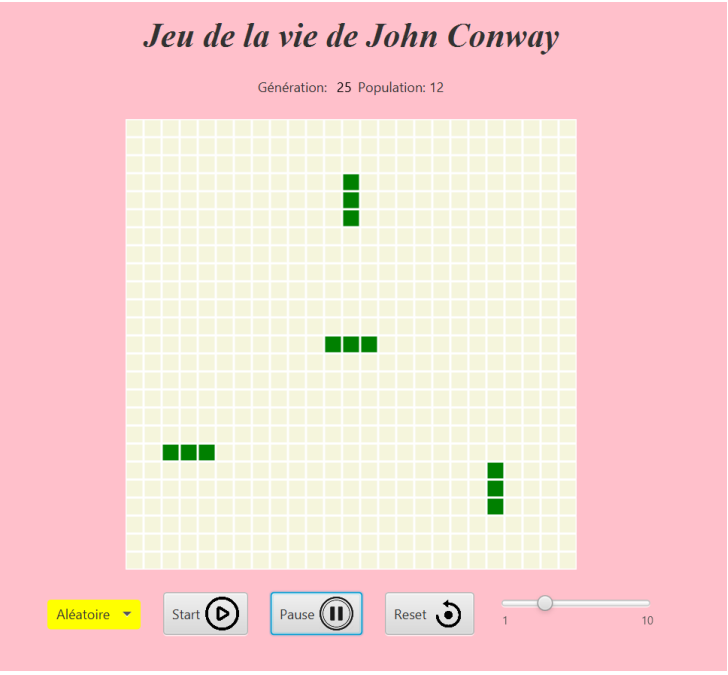

FIGURE 8 – Mouvement oscillatoire

5.3 Résultats quantifiables

Le projet n'implique pas des calculs lourds, mais juste la fluidité de l'animation et la gestion de la grille qui doivent être efficaces. Les performances dépendent de la taille de la grille et de la fréquence des mises à jour (intervalle de 500 ms dans mon projet ci).

6 Conclusion

6.1 Récapitulatif des fonctionnalités principales

— Initialisation de la grille

— Simulation du jeu

— Interface utilisateur

— Animation

6.2 Propositions d'améliorations

— **Optimisation des performances :** Améliorer l'efficacité de la mise à jour de la grille, notamment pour les grandes tailles de grille.

— **Personnalisation de la taille de la grille :** Permettre à l'utilisateur de choisir la taille de la grille au démarrage de la simulation.

— **Ajout d'autres motifs prédéfinis :** Ajouter la possibilité de charger des motifs connus du Jeu de la Vie (canon, vaisseaux , oscillateurs ...etc).

7 Liens utiles

— Page wikipedia de John H Conway : John Horton Conway

— Page wikipedia du jeu de la vie : Jeu de la vie.

8 Références

— Lien pour consulter la : Documentation de JavaFX

— Page wikipedia du jeu de la vie : Jeu de la vie

— Page dcode du jeu de la vie : dcode

— Playlist de : EvoluNoob : Javafx-cours

— Playlist de : LES TEACHERS DU NET : APPRENDRE JAVAFX

— Je me suis inspiré de l'explication d'une partie de la video ci pour créer la grille cyclique (l'idée de faire modulo "taille de la grille" dans la méthode "compterVoisin" de la classe "JeuDeLaVieModel") : Video- youtube