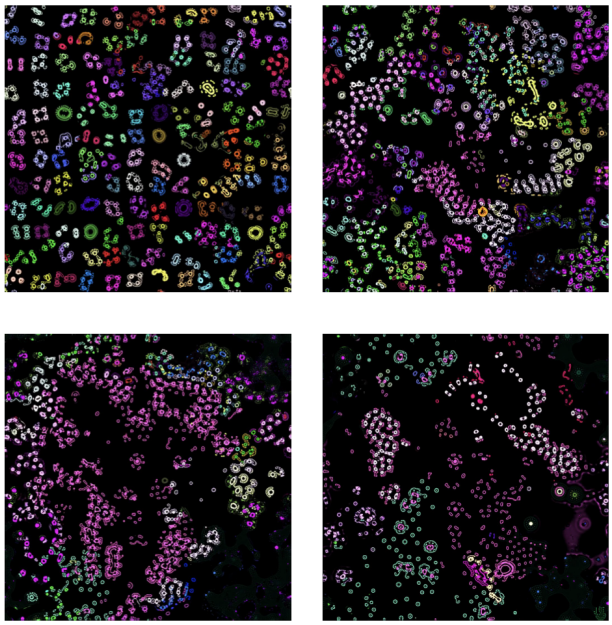
Institut GALILÉE
Université Sorbonne Paris Nord

Automates cellulaires



Simulations multi-espèces par FLOW-LENIA

Programmation orientée objet

A.DUPONT-BOUILLARD – J.CHAUSSARD

E.NICOLAS – R.HOMSI

G3SI - SUP GALILÉE

ethan.bento-nicolas@edu.univ-paris13.fr
rami.homsi@edu.univ-paris13.fr

15 juin 2024

Engagement de non-plagiat

Nous, soussignés E.NICOLAS et R.HOMSI, étudiants en 1ère année d'école d'ingénieur à Sup Galilée, déclarons être pleinement conscients que la copie de tout ou partie d'un document, quel qu'il soit, publié sur tout support existant, y compris sur Internet, constitue une violation du droit d'auteur ainsi qu'une fraude caractérisée, tout comme l'utilisation d'outils d'Intelligence Artificielle pour générer une partie de ce rapport ou du code associé. En conséquence, nous déclarons que ce travail ne comporte aucun plagiat, et assurons avoir cité explicitement, à chaque fois que nous en avons fait usage, toutes les sources utilisées pour le rédiger.

Fait à Villetaneuse, le 15 juin 2024

E.N – R.H

Table des matières

Listings

1 Introduction

1.1 Préface

Actuellement en première année d'école d'ingénieur en spécialité informatique, et dans le cadre du cours de programmation orientée objet, nous avons eu l'occasion de pouvoir coder en java des automates cellulaires en N dimensions.

1.2 Présentation des objectifs

Un automate cellulaire est un système composé d'une grille ND composée de cases (appelées cellules), qui peuvent être dans plusieurs états ; dans notre cas discret, vivante ou morte (allumé ou éteinte) mais on peut également être dans une version continue comme FLOW-LENIA [1], l'image de couverture étant tiré de leur article de recherche [2]. Une cellule peut passer d'un état à un autre selon certaines règles prédéfinies, et l'on regarde comment les cellules agissent au fur et à mesure. Nous avons pu regarder la vidéo recommandée de David Louapre sur sa chaîne Science étonnante où nous nous sommes familiariser avec un certain type de règles se rapportant au "jeu de la vie" de John Horton Conway. Nous avons donc voulu tester certaines configurations de la vidéo. Pour aller plus loin, nous avons vu une autre vidéo avec les mêmes règles faite par EGO [3], elle présente la notion de Turing complet et nous montre comment un automate cellulaire peut faire tourner lui même un automate cellulaire par le biais de portes logiques, de mémoire... étant rendu possible par la présence de petits objets mouvants : les gliders. Cette vidéo nous a vraiment motivé pour ce projet. Une seconde de Science Etonnante, le jeu de la vie 2.0 est la vidéo ou nous avons découvert flow-lenia et la possibilité de rendre une cellule continue ce qui fait une simulation beaucoup plus organique et réelle. Nous avons rajouté également une visualisation 3D dont les règles ont été changées pour se rapprocher du jeu de la vie de Conway en 2D [4].

2 Compilation et exécution du programme

1. Ouvrir un terminal
2. Se placer dans le dossier contenant les fichiers sources puis entrer :

```
javac --module-path ./javafx/lib --add-modules javafx.controls,javafx.fxml
*.java
```

3. Lancer le programme avec

```
java --module-path ./javafx/lib --add-modules javafx.controls,javafx.fxml
Main
```

4. Sélectionner le fichier de configuration XML voulu ou créer en un nouveau dans un éditeur de texte via le model.txt dans le dossier configs (voir la partie sur le fichier XML a la fin du rapport).
5. Pour supprimer les .class :

```
rm *.class
```

NOTE : Le dossier `javafx` est la bibliothèque pour la 3D (téléchargé sur Oracle).

1. https://sites.google.com/view/flowlenia/videos
2. https://arxiv.org/pdf/2212.07906 p.7
3. https://youtu.be/eMn43As24Bo?si=NnCszi1UgtSdwtJw
4. https://chrisevans9629.github.io/blog/2020/07/27/game-of-life

3 Implémentation de la Cellule

Tout le projet se repose sur la cellule, il nous faut donc implémenter cette classe efficacement : une cellule est soit morte, soit vivante, on doit donc pouvoir représenter l'état de la cellule (par le champ `private boolean value`). On se passera d'ajouter les coordonnées de la cellule, les calculs pour passer d'une génération à une autre se feront dans la grille. On ajoute enfin un setter et un getter pour l'état de la cellule.

```
1  public class STRUCT_Cell
2  {
3      private boolean value;
4
5      public STRUCT_Cell()
6      {
7          value = false; // Default value for a cell
8      }
9      public void setCellValue(boolean value)
10     {
11         this.value = value;
12     }
13     public boolean getCellValue()
14     {
15         return value;
16     }
17 }
```

Listing 1 – Classe STRUCT_Cell

4 Implémentation de la Grille

4.1 Prérequis

L'une des plus-values de ce projet est de pouvoir représenter le jeu de la vie dans des grilles en N-dimentions, l'implémentation de ces dernières ne se fera donc pas traditionnelement avec un tableau de cellules à deux ou 3 dimensions, mais de manière récursive.

4.2 Récursivité

On implémente donc ces grilles dans la classe STRUCT_Grid_ND. Parmi les méthodes récursives de la cette classe, on retrouve le constructeur `public STRUCT_Grid_ND(int... sizes)`, qui prend en entée un nombre variable de tailles de dimension : par exemple, si on veut créer une grille en 4D avec des tailles pour chaque dimension respectivement de 50, 60, 70 et 80 cellules, on l'instenciera comme suit : `STRUCT_Grid_ND grid = new STRUCT_Grid_ND(50, 60, 70, 80);`.
Ainsi, pour construire une grille, le constructeur crée récursivement des tableaux de tableaux de tableaux... jusqu'à "atteindre" la dernière dimension pour chaque tableau et la remplir de cellules :

```
private int[] dimensions;
private Object[] grid;

public STRUCT_Grid_ND(int... sizes) {
    dimensions = sizes;
    int N = sizes[0]; // Dimension size

    if (sizes.length == 1) {
        grid = new STRUCT_Cell[N]; // Derniere dim tab cell
        for (int i = 0; i < N; i++) {
            grid[i] = new STRUCT_Cell();
        }
    } else {
        grid = new STRUCT_Grid_ND[N];
        int[] newSizes = new int[sizes.length - 1];
        System.arraycopy(sizes, 1, newSizes, 0, sizes.length - 1);
        for (int i = 0; i < N; i++) {
        grid[i] = new STRUCT_Grid_ND(newSizes); // Recursive sous-grid
        }
    }
}
```

Listing 2 – Constructeur de la classe STRUCT_Grid_ND

La méthode `GetCell` qui retourne la cellule à la position entrée le fait aussi de manière récursive, en parcourant la grille de la même manière dont cette dernière a été créée, jusqu'à encore une fois atteindre la bonne position dans le bon tableau :

```
public STRUCT_Cell getCell(int... pos) {
    if (pos.length == 1) {
        return ((STRUCT_Cell) grid[pos[0]]); // Get cell
    } else {
        int[] newPos = new int[pos.length - 1];
        System.arraycopy(pos, 1, newPos, 0, pos.length - 1);
        return ((STRUCT_Grid_ND) grid[pos[0]]).getCell(newPos);
    }
}
```

Listing 3 – Méthode getCell

5 Implémentation des Voisinages

5.1 Prérequis

Pour faire fonctionner cette simulation, on doit pouvoir passer de génération en génération en suivant des règles que l'on doit définir à partir du voisinage de chaque cellule : pour passer au nouveau tour, on parcourra la grille de cellule en cellule en calculant pour chacunes d'entre elles le nombre de ses voisines vivantes, la règle qui régit le jeu décidera du nouvelle état de la cellule cible dans le prochain tour.

5.2 Les Voisinages

5.2.1 Voisinages classiques

Pour l'implémentation, nous sommes passé par une interface qui permet d'implémenter les voisinages prédéfinis `Gk` et `Gk*`. Les Gk sont des champs de la classe, des listes de positions relatives a la position centrale ; lorsque l'on a besoin d'appelé un Gk on appelle la méthode `getNeighbors` qui renvoie la liste des entiers (0 ou 1) des cases environnantes selon le voisinage demandé par le getter `getNeighborhoodByName`. Il s'agit d'un switch qui renvoie les méthodes demandées selon le nom du voisinage demandé :

```java
interface Neighborhood {
    List<int[]> getNeighbors(int... position);
}

public class TOOLS_Neighborhoods {

    // Voisinages predefinis
    public static final Neighborhood G0 = position -> List.of(position);
    public static final Neighborhood G2 = position -> List.of(
            new int[]{position[0] - 1},
            position,
            new int[]{position[0] + 1}
    );

    ... // Le reste des Gk

    public static final Neighborhood G2Star = position -> G2.getNeighbors(position).
    stream()
            .filter(neighbor -> !java.util.Arrays.equals(neighbor, position))
            .collect(Collectors.toList());

    ... // Le reste des Gk*

    public static Neighborhood getNeighborhoodByName(String name) {
        switch (name) {
            case "G0":
                return G0;
            case "G2":
                return G2;

            ...

            default:
                for (CustomNeighborhood custom : customNeighborhoods) {
                    if (custom.getName().equals(name)) {
                        return custom.getNeighborhood();
                    }
                }
                return null;
        }
    }
}
```

Listing 4 – TOOLS_Neighborhoods

5.2.2 Voisinages customs

Les voisinages customs ont été plus compliqué car on ne savait pas comment créer de nouveaux champs lors de l'exécution du programme ou un moyen de bien les stocker. Nous avons finalement décidé de faire une nouvelle classe privée et d'en faire une `ArrayList`. Nous avons fait une méthode pour ajouter un nouveau voisinage à notre liste avec son nom et sa liste de voisins relatifs ; par exemple ((1,0),(0,-1)) qui souhaite récuperer la case à gauche et a droite en 2D. Nous initialisons donc un voisinage en calculant donc les positions absolues des voisinages transmis pour avoir des `position[i] + k` et retournons ces positions absolues. Nous ajoutons ensuite à la liste ce nouveau voisinage en tant qu'instance de voisinage custom avec le nom et cette liste de voisinage absolue. Nous avons un `toString()` pour afficher les voisinages disponibles :

```
interface Neighborhood {
    List<int[]> getNeighbors(int... position);
}

public class TOOLS_Neighborhoods {

    private static final List<CustomNeighborhood> customNeighborhoods = new ArrayList<>()
    ;

    public static void addCustomNeighborhood(String name, List<int[]> neighbors) {
        Neighborhood customNeighborhood = position -> {
            List<int[]> absoluteNeighbors = new ArrayList<>();
            for (int[] neighbor : neighbors) {
                int[] absoluteNeighbor = new int[position.length];
                for (int i = 0; i < position.length; i++) {
                    absoluteNeighbor[i] = position[i] + neighbor[i];
                }
                absoluteNeighbors.add(absoluteNeighbor);
            }
            return absoluteNeighbors;
        };
        customNeighborhoods.add(new CustomNeighborhood(name, customNeighborhood));
    }

    public String toString() {
        return "********* NEIGHBORHOODS **********\n"
                + "Predefined neighborhoods: G0, G2, G4, G8, G6, G26, G2*, G4*, G8*, G6*,
     G26*\n"
                + "Custom neighborhoods: " + customNeighborhoods.stream().map(
    CustomNeighborhood::getName).collect(Collectors.joining(", ")) + "\n";
    }

    private static class CustomNeighborhood {
        private final String name;
        private final Neighborhood neighborhood;

        public CustomNeighborhood(String name, Neighborhood neighborhood) {
            this.name = name;
            this.neighborhood = neighborhood;
        }

        public String getName() {
            return name;
        }

        public Neighborhood getNeighborhood() {
            return neighborhood;
        }
    }
}
```

Listing 5 – CustomNeighborhood

6 Implémentation des Règles

6.1 Prérequis

Pour que les cellules puissent changer d'état, nous devons implémenter des règles primaires suivantes, ensuite, nous devons faire en sorte de pouvoir imbriquer ces règles pour faire des règles d'évolution du système :

- ET(val1, val2) : renverra 1 si val1 et val2 sont tous deux différents de 0, et 0 sinon.
- OU(val1, val2) : renverra 0 si val1 et val2 sont tous deux égaux à 0, et 1 sinon.
- NON(val) : renverra 1 si val vaut 0, et 0 sinon.
- SUP(val1, val2) : renverra 1 si val1 est strictement supérieure à val2, et 0 sinon.
- SUPEQ(val1, val2) : renverra 1 si val1 est supérieure ou égale à val2, et 0 sinon.
- EQ(val1, val2) : renverra 1 si val1 est égal à val2, et 0 sinon.
- COMPTER(voisinage) : renverra le nombre de cellules à l'état 1 dans le voisinage de la cellule observée.
- ADD(val1, val2) : renverra la somme de val1 et val2.
- SUB(val1, val2) : renverra la soustraction de val1 et val2.
- MUL(val1, val2) : renverra le produit de val1 et val2.
- SI(val1, val2, val3) : si val1 est différent de 0, renverra val2, et sinon, renverra val3.

6.2 L'arbre

Pour l'implémentation de ces règles nous avons choisis de les mettre sous forme d'arbre, les noeuds sont les valeurs donc des entiers ou les operateurs. Pour cela, nous avons créer la classe abstraite **TreeNode** et la méthode abstraite `getValue()` cela à donc permis de créer une classe d'opérateurs et de constantes en ayant toujours cette méthode qui permet de prendre la valeur d'un noeud donc de l'entier si c'est un noeud constant. Pour l'opérateur, la méthode getValue va dépendre de la règle à implémenter mais nous verrons les règles dans la suite. Les variables globales pour l'opérateur sont 3 puisque le SI peut prendre 3 valeurs mais les autres n'en n'ont que 2 ou 1. C'est pour cela qu'on a 3 constructeur différent qui seront appelé en fonction de la règle :

```
1  abstract class TreeNode {
2      abstract int getValue(); // 2 Nodes : operators and constants (leafs) so we have to
       make an abstract to not differentiate a constnode and operatornode
3  }
4
5  class ConstNode extends TreeNode {
6      private final int value;
7
8      public ConstNode(int value) {
9          this.value = value;
10     }
11
12     @Override
13     int getValue() {
14         return value;
15     }
16 }
17
18 abstract class OperatorNode extends TreeNode {
19     protected TreeNode left;
20     protected TreeNode middle;
21     protected TreeNode right;
22
23     public OperatorNode(TreeNode left, TreeNode middle, TreeNode right)
24     {
25         this.left = left;
26         this.middle = middle;
27         this.right = right;
28     }
29
30     public OperatorNode(TreeNode left, TreeNode right)
31     {
32         this(left, null, right);
```

```
    }

    public OperatorNode(TreeNode left)
    {
        this(left, null, null);
    }

    @Override
    abstract int getValue();
}
```

Listing 6 – STRUCT_Tree

6.3 Les règles récursives

Une fois la structure de données en arbre créée et pête à l'emploi, on a pu implémenter les règles. On commence par écrire les opérateurs unaires (NON), binaires (ET, OU, NON, SUP, SUPEQ, EQ, ADD, SUB, MUL) et ternaires (SI) primaires. Chacuns de ces opérateurs hérite de la classe `OperateurNode`, qui elle-même hérite de la classe `TreeNode`, a un construceur, qui appel le super adéquat en fonction de du nombre d'arguments de cet opérateur, et une méthode `getValue()` qui réécrit celle des classes mères et qui calcule récursivement la valeur à envoyer, en fonction de l'opérateur choisi, et du ou des fils de cet opérateur dans l'arbre qui représente la règle complète :

```
class NON extends OperatorNode {
    public NON(TreeNode left) {
        super(left);
    }

    @Override
    int getValue() {
        return (left.getValue() == 0) ? 1 : 0;
    }
}

class ET extends OperatorNode {
    public ET(TreeNode left, TreeNode right) {
        super(left, right);
    }

    @Override
    int getValue() {
        return (left.getValue() != 0 && right.getValue() != 0) ? 1 : 0;
    }
}

class SI extends OperatorNode {
    public SI(TreeNode left, TreeNode middle, TreeNode right) {
        super(left, middle, right);
    }

    @Override
    int getValue() {
        return (left.getValue() != 0) ? middle.getValue() : right.getValue();
    }
}
```

Listing 7 – Les classes d'opérateurs unaire : NON | binaire : ET | ternaire : SI

Il ne reste plus qu'à ajouter la règle COMPTER, qui est un peut différentes des autres. Premièrement, elle hérite de la classe TreeNode directement, c'est aussi la seule qui a besoin du voisinage de la cellule pour renvoyer la bonne valeur, pour cela, sa méthode `getValue()` n'appelle plus la méthode des noeuds suivant récursivement, mais itère sur le voisinage en appellant la méthode `getCellValue()` pour renvoyer le nombre de cellules vivantes dans ce voisinage.

```
class COMPTER extends TreeNode {
    private static STRUCT_Grid_ND grid;
    private static int[] position;
```

```
    private String voisinage;

    public COMPTER(String voisinage) {
        this.voisinage = voisinage;
    }

    public static void setSettings(STRUCT_Grid_ND grid, int... position) {
        COMPTER.grid = grid;
        COMPTER.position = position;
    }

    private List<int[]> getNeighbors() {
        return TOOLS_Neighborhoods.getNeighborhoodByName(voisinage).getNeighbors(position
    );
    }

    @Override
    int getValue() {
        int liveNeighbors = 0;
        List<int[]> neighbors = getNeighbors();
        for (int[] neighbor : neighbors) {
            try {
                if (grid.getCell(neighbor).getCellValue()) {
                    liveNeighbors++;
                }
            } catch (ArrayIndexOutOfBoundsException e) {
                // If the neighbor is out of bounds, treat it as a dead cell
            }
        }
        return liveNeighbors;
    }
}
```

Listing 8 – Classe de l'opérateur COMPTER

Une fois qu'on a tous les opérateurs primaires, on peut ajouter les règles traditionnelles complètes en 2D et 3D (cf. 7.2) :

```
class Rule3D {

    public boolean isAlive(STRUCT_Grid_ND grid, int ...position)
    {
        int x = position[0];
        int y = position[1];
        int z = position[2];

        COMPTER.setSettings(grid, x, y, z);
        TOOLS_EvolutionRule rule3d = new TOOLS_EvolutionRule("SI(EQ(COMPTER(G0),1), SI(OU
    (EQ(COMPTER(G26*),5),EQ(COMPTER(G26*),6)),1,0) , SI(EQ(COMPTER(G26*),4),1,0))", false
    );
        rule3d.cursor = 0;
        return rule3d.createNode(rule3d.parseFile()).getValue() == 1;
    }
}

class Rule2D {

    public boolean isAlive(STRUCT_Grid_ND grid, int ...position)
    {
        int x = position[0];
        int y = position[1];

        COMPTER.setSettings(grid, x, y);
        TOOLS_EvolutionRule rule2d = new TOOLS_EvolutionRule("SI(EQ(COMPTER(G0),1), SI(OU
    (EQ(COMPTER(G8*),2),EQ(COMPTER(G8*),3)),1,0),SI(EQ(COMPTER(G8*),3),1,0))", false);
        rule2d.cursor = 0;
        return rule2d.createNode(rule2d.parseFile()).getValue() == 1;
    }
}
```

Listing 9 – Règles du jeu de la vie

7 Implémentation de l'interface graphique

7.1 1D & 2D

Pour l'interface graphique nous avons utilisé la classe graphique fournie dans le projet. Nous avons cependant enlevé la grille de fond (mis en commentaire) et ajouter des auditeurs pour pouvoir ajouter des cellules en cliquant sur la grille (au début de la simulation) zoomer / dezoomer et également déplacer la grille dans la fenêtre. Pour ajouter une cellule ou la supprimer, on récupère la colonne et la ligne cliquée par un `MouseEvent` puis nous faisons attention avec le `scaleFactor` en fonction du zoom et de l'`offset` c'est à dire du déplacement de la grille avec les flèches. Si la cellule cliquée était déjà vivante elle meurt sinon elle vie. Les auditeurs appellent des fonctions qui modifient le scaleFactor ou l'offset puis on repeint la grille ; la grille se repeint en utiisant deux fonctions de la classe `Graphics2D` :

```
        g2d.scale(scaleFactor, scaleFactor);
        g2d.translate(offsetX / scaleFactor, offsetY / scaleFactor);
```

Listing 10 – Fonctions ajoutées pour appliquer le zoom et l'offset

Commandes :

- Cliquer sur la grille pour ajouter / supprimer des cellules avant le début de la simulation (vous pouvez récuperer les cellules cliquées au format XML dans le terminal pour en faire un fichier de congifuration
- Zoom : Z
- Dezoom : S
- Déplacements : flèches directionnelles

```
    this.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                int col = (int) ((e.getX() - taille_case * scaleFactor - offsetX) / (taille_case * scaleFactor));
                int row = (int) ((e.getY() - taille_case * scaleFactor - offsetY) / (taille_case * scaleFactor));
                if (col >= 0 && col < largeur && row >= 0 && row < hauteur) {
                    try {
                        grid.getCell(col, row).setCellValue(!grid.getCell(col, row).getCellValue());
                        if (grid.getCell(col, row).getCellValue()) {
                            System.out.println("<Cell>" + col + "," + row + "</Cell>");
                            grid.getCell(col, row).setCellValue(true);
                            colorierCase(col, row, Color.BLUE);
                        } else {
                            grid.getCell(col, row).setCellValue(false);
                            colorierCase(col, row, Color.WHITE);
                        }
                    } catch (ArrayIndexOutOfBoundsException ex) {
                        System.out.println("Error: Invalid cell");
                    }
                }
            }
        });

        this.addKeyListener(new KeyAdapter() {
            @Override
            public void keyPressed(KeyEvent e) {
                if (e.getKeyChar() == 'z') {
                    zoomIn();
                } else if (e.getKeyChar() == 's') {
                    zoomOut();
                } else if (e.getKeyCode() == KeyEvent.VK_UP) {
                    offsetY += 10;
                    repaint();
                } else if (e.getKeyCode() == KeyEvent.VK_DOWN) {
                    offsetY -= 10;
                    repaint();
                } else if (e.getKeyCode() == KeyEvent.VK_LEFT) {
                    offsetX += 10;
```

```
                repaint();
            } else if (e.getKeyCode() == KeyEvent.VK_RIGHT) {
                offsetX -= 10;
                repaint();
            }
        }
    });

    this.setFocusable(true);
    this.requestFocusInWindow();
}
```

Listing 11 – Auditeurs de la classe GFX_GrilleGraphique

Nous avons une exception en utilisant cette grille graphique dans le `Main` :
java.util.ConcurrentModificationException
le problème vient de la première coloration pour l'affichage au début de la simulation. Si nous enlevons cette coloration nous n'avons pas cette exception. Nous avons décidé de la laisse pour pouvoir voir le système de départ et pouvoir le modifier en cliquant dessus.

7.2 3D

De par notre motivation, nous avons voulu implémenter une vue 3D et ainsi pouvoir visualiser un jeu de la vie en 3D. Pour cela, après quelques recherches nous avons décidé d'utiliser la bibliothèque **JavaFX** disponible sur Oracle [5]. Cependant, pour compiler et utiliser le programme directement, nous avons intégré les fichiers de la bibliothèque dans le dossier **"javafx"**.

Nous avons pu nous renseigner sur comment faire des cubes et donc faire une grille de cubes. Nous avons cependant rencontré un problème avec les règles d'évolutions. Nous avons dû les mettre en dur dans une classe `Rule3D` car la création dans le constructeur d'une règle amenait à une exception de pointeur null. On suppose que c'est lié au fonctionnement de JavaFX avec les threads et la gestion de la mémoire. Nous n'avons pas réussi à régler le problème. La règle utilisée en dur est une imitation du "jeu de la vie" de Conway, nous avons tiré cette règle du site de Chris Evans [6] (pas l'acteur de Captain America).

— Any live cell with 5-6 live neighbors survives.
— Any dead cell with 4 live neighbors becomes a live cell.
— All other live cells die in the next generation. Similarly, all other dead cells stay dead.

Et l'avons implémenter à l'aide de nos règles :

SI(EQ(COMPTER(G0),1), SI(OU(EQ(COMPTER(G26*),5),EQ(COMPTER(G26*),6)),1,0) , SI(EQ(COMPTER(G26*),4),1,0))

Avec JavaFX, on crée un groupe rempli de box de couleur rouge qui possède une taille. On mets à jour ces cubes en utilisant la `Rule3D` mentionnée plus haut :

```
private void updateCube() {
    STRUCT_Grid_ND new_grid = new STRUCT_Grid_ND(grid.getDimensions());

    for (int x = 0; x < X; x++) {
        for (int y = 0; y < Y; y++) {
            for (int z = 0; z < Z; z++) {
                if (new Rule3D().isAlive(grid, x, y, z)){
                    new_grid.getCell(x, y, z).setCellValue(true);
                } else {
                    new_grid.getCell(x, y, z).setCellValue(false);
                }
            }
        }
    }
    STRUCT_Grid_ND temp = grid;
    grid = new_grid;
```

5. https://www.oracle.com/java/technologies/java-archive-javafx-downloads.html
6. https://chrisevans9629.github.io/blog/2020/07/27/game-of-life

```
        new_grid = temp;
    }

    private void createCube(SmartGroup group) {
        group.getChildren().removeIf(node -> node instanceof Box);
        for (int x = 0; x < X; x++) {
            for (int y = 0; y < Y; y++) {
                for (int z = 0; z < Z; z++) {
                    if (grid.getCell(x, y, z).getCellValue()) {
                        Box box = new Box(SMALL_BOX_SIZE, SMALL_BOX_SIZE, SMALL_BOX_SIZE)
;
                        PhongMaterial material = new PhongMaterial();
                        material.setDiffuseColor(Color.RED);
                        material.setSpecularColor(Color.RED);
                        box.setMaterial(material);
                        box.setTranslateX(x * (SMALL_BOX_SIZE) - (X * (SMALL_BOX_SIZE) /
2));
                        box.setTranslateY(y * (SMALL_BOX_SIZE) - (Y * (SMALL_BOX_SIZE) /
2));
                        box.setTranslateZ(z * (SMALL_BOX_SIZE) - (Z * (SMALL_BOX_SIZE) /
2));
                        group.getChildren().add(box);
                    }
                }
            }
        }
    }
```

Listing 12 – Méthodes updateCube et createCube

On ajoute des auditeurs pour se déplacer dans l'espace 3D : on zoom avec Z on dezoom avec S et on fait tourner l'objet en maintenant le clic gauche de la souris :

```
   primaryStage.addEventHandler(KeyEvent.KEY_PRESSED, event -> {
            switch (event.getCode()) {
                case Z:
                    group.translateZProperty().set(group.getTranslateZ() - 100);
                    break;
                case S:
                    group.translateZProperty().set(group.getTranslateZ() + 100);
                    break;
                default:
                    break;
            }
        });

        private void initMouseControl(SmartGroup group, Scene scene) {
        Rotate xRotate;
        Rotate yRotate;
        group.getTransforms().addAll(
                xRotate = new Rotate(0, Rotate.X_AXIS),
                yRotate = new Rotate(0, Rotate.Y_AXIS));
        xRotate.angleProperty().bind(angleX);
        yRotate.angleProperty().bind(angleY);

        scene.setOnMousePressed(event -> {
            anchorX = event.getSceneX();
            anchorY = event.getSceneY();
            anchorAngleX = angleX.get();
            anchorAngleY = angleY.get();
        });

        scene.setOnMouseDragged(event -> {
            angleX.set(anchorAngleX - (anchorY - event.getSceneY()));
            angleY.set(anchorAngleY + anchorX - event.getSceneX());
        });
    }

    class SmartGroup extends Group {

        Rotate r;
        Transform t = new Rotate();
```

```
40
41          void rotateByX(int ang) {
42              r = new Rotate(ang, Rotate.X_AXIS);
43              t = t.createConcatenation(r);
44              this.getTransforms().clear();
45              this.getTransforms().addAll(t);
46          }
47
48          void rotateByY(int ang) {
49              r = new Rotate(ang, Rotate.Y_AXIS);
50              t = t.createConcatenation(r);
51              this.getTransforms().clear();
52              this.getTransforms().addAll(t);
53          }
54      }
```

Listing 13 – Auditeurs de GFX_Cube

8 Fichiers d'entrée et Parsing

8.1 Prérequis

Nous avons suivi les recommandations de l'énoncé en utilisant des fichiers de configurations XML. Cees derniers servent donc à décrire les conditions des simulations que l'on veut étudier. L'implémentation de tels fichiers implique donc de créer un algorithme de parsing notemment pour extraire en les données des voisinages et des règles.

8.2 XML

Nous avons créé le modèle XML avec les dimensions, la coupe (disponible pour une coupe 2D de la 3D), les voisinages customs, la règle d'évolution et les cellules de départ soit en dur soit en random. Nous avons fourni quelques fichiers de configuration :

- Sierpiński comme demandé
- Random2D qui était demandé
- Glider Gun qui est une usine de glider la plus basique
- Kok's Galaxy qui est un oscilateur en forme de galaxie
- Chaotic le fichier de la question bonus
- Oscillator3D qui est un des oscillateur 3D basique en 3 générations

```
<GameOfLife>
    <Dimension>2</Dimension>
    <GridSize>
        <Size>11</Size>
        <Size>11</Size>
    </GridSize>
    <Cut>axis(x y z) value</Cut>
    <CustomNeighborhoods>
        <Neighborhood>GXX = ((a, b, ...), (c, d, ...), ...)</Neighborhood>
    </CustomNeighborhoods>
    <EvolutionRule>SI(EQ(COMPTER(G0),1), SI(OU(EQ(COMPTER(G8*),2),EQ(COMPTER(G8*),3))
    ,1,0) , SI(EQ(COMPTER(G8*),3),1,0))</EvolutionRule> # Regle 2D de Conway
    <EvolutionRule>SI(EQ(COMPTER(G0),1), SI(OU(EQ(COMPTER(G26*),5),EQ(COMPTER(G26*),6))
    ,1,0) , SI(EQ(COMPTER(G26*),4),1,0))</EvolutionRule> # Regle 3D
    <InitialCells>
        <Cell>x , y , z ...</Cell>
        OR
        <Random>k</Random>
    </InitialCells>
</GameOfLife>
```

Listing 14 – Modèle de configuration XML

8.3 Parsing des voisinages

Pour parser les voisinages suivants : **GXX = ((a, b, ...), (c, d, ...), ...)**, le code Java fonctionne de la manière suivante :

La méthode `parseFile` prend en entrée une chaîne de caractères `buffer` représentant une ligne de texte contenant le voisinage. Si `buffer` n'est pas null et n'est pas une chaîne vide après avoir été trimée (les espaces en début et fin de chaîne sont supprimés), et qu'elle contient un signe égal `=`, on peut diviser en deux parties : le nom du voisinage `name` (à gauche du `=`) et la partie contenant les coordonnées `coordinatesPart` (à droite du `=`). Les coordonnées sont ensuite extraites et traitées par la méthode `parseCoordinates`. Enfin, le voisinage avec son nom et ses coordonnées est ajouté à la liste des voisinages via la méthode `addCustomNeighborhood` de la classe `TOOLS_Neighborhoods`.

La méthode `parseCoordinates` prend en entrée la chaîne de caractères `coordinatesPart` contenenant les coordonnées comme une liste de k-uplets. On supprime les parenthèses extérieures et divise les coordonnées en utilisant `"), ("` comme séparateur. Chaque k-uplets de coordonnées est converties en entiers en supprimant la virgule qui les sépare. On retourne cette liste de coordonnées.

Par exemple, pour : `GXX = ((1, 2), (3, 4), (5, 6))`. La chaîne est valide donc elle est divisée en deux parties : `name` sera `"GXX"` et `coordinatesPart` sera `"((1, 2), (3, 4), (5, 6))"`. `coordinatesPart` est passé à `parseCoordinates`.

La méthode `parseCoordinates` prend `coordinatesPart`. Après suppression des parenthèses extérieures, il reste `"1, 2), (3, 4), (5, 6"`. La chaîne est divisée par `"), (`, ce qui donne `["1, 2", "3, 4", "5, 6"]`. Chaque paire est traité pour convertir les valeurs en entiers : `"1, 2"` devient `[1, 2]`, `"3, 4"` devient `[3, 4]`, et `"5, 6"` devient `[5, 6]`. La liste des coordonnées `[[1, 2], [3, 4], [5, 6]]` est retournée et ajoutée aux voisinages personnalisés sous le nom `"GXX"`.

```
    public class TOOLS_NeighborhoodParser {

    public void parseFile(String buffer) throws IOException {
        if (buffer != null) {
            buffer = buffer.trim();
            if (!buffer.isEmpty() && buffer.contains("=")) {
                String[] parts = buffer.split("=");
                String name = parts[0].trim();
                String coordinatesPart = parts[1].trim();
                List<int[]> neighbors = parseCoordinates(coordinatesPart);
                TOOLS_Neighborhoods.addCustomNeighborhood(name, neighbors);
            }
        }

    }

    private List<int[]> parseCoordinates(String coordinatesPart) {
        List<int[]> neighbors = new ArrayList<>();
        // Remove the outer parentheses
        coordinatesPart = coordinatesPart.substring(1, coordinatesPart.length() - 1);
        // Split the coordinates by "), ("
        String[] coordinatePairs = coordinatesPart.split("\\),\\s*\\(");
        for (String pair : coordinatePairs) {
            pair = pair.replace("(", "").replace(")", "");
            String[] coords = pair.split(",");
            int[] neighbor = new int[coords.length];
            for (int i = 0; i < coords.length; i++) {
                neighbor[i] = Integer.parseInt(coords[i].trim());
            }
            neighbors.add(neighbor);
        }
        return neighbors;
    }
}
```

Listing 15 – TOOLS_NeighborhoodParser

8.4 Parsing des règles

Le parsing des règles se trouve dans la classe `TOOLS_EvolutionRule`. Cette classe prend en entrée un chemin de fichier ou une chaîne de caractères contenant la règle d'évolution à parser. Le constructeur de la classe initialise le chemin du fichier et lit son contenu si le paramètre `isPath` est `true`. Le contenu du fichier est stocké dans la variable `fileContent`. La méthode `readFileContent` lit le fichier ligne par ligne et retourne son contenu sous forme de chaîne de caractères.

La méthode `parseFile` parcourt le contenu du fichier ou de la chaîne de caractères en utilisant un curseur (`cursor`). Elle lit les caractères un par un jusqu'à rencontrer un délimiteur (parenthèse '(', virgule ',' ou espace ' '). Lorsqu'un délimiteur est trouvé on avance le curseur au caractère suivant en sautant les délimiteurs si il y'en a d'autres (des espaces par exemple), puis retourne la chaîne de caractères lue jusqu'à ce point.

La méthode `createNode` crée des nœuds d'arbre à partir des chaînes de caractères parsées par la méthode précédente. Si la chaîne est numérique (à l'aide la méthode `isNumeric`), un noeud constante est créé. Sinon, un noeud opérateur est créé en fonction de la chaîne de caractères (switch). Chaque opérateur crée des noeuds enfants en appelant récursivement `createNode` et `parseFile` pour parser les opérandes. Par exemple, l'opérateur `ET` crée un noeud avec deux enfants, chacun étant le résultat d'un appel récursif à `createNode` et appelle `parseFile` qui trouve les 2 valeurs du ET (soit des constantes ou encore un opérateur imbriqué).

La méthode `setCursor` permet de définir la position du curseur. La méthode `isNumeric` vérifie si une chaîne de caractères est numérique en tentant de la parser en entier. Si la chaîne ne peut pas être convertie en entier, `isNumeric` retourne `false`.

Ainsi, la classe `TOOLS_EvolutionRule` permet de lire et de parser une règle d'évolution définie dans le XML et de construire l'arbre représentant cette règle. On peut ensuite utiliser `getValue()` sur la racine (**TreeNode root = evoRule.createNode(buffer)**) pour évaluer la règle.

```
    public class TOOLS_EvolutionRule {

    public String parseFile() {
        StringBuilder buffer = new StringBuilder();

        if (cursor < 0 || cursor >= fileContent.length()) {
            return "";
        }

        // Lire les caracteres jusqu'a rencontrer un delimiteur
        while (cursor < fileContent.length()) {
            char currentChar = fileContent.charAt(cursor);
            if (currentChar == '(' || currentChar == ')' || currentChar == ',' ||
    currentChar == ' ') {
                cursor++;
                while (cursor < fileContent.length() && (fileContent.charAt(cursor) == '(
    ' || fileContent.charAt(cursor) == ')' || fileContent.charAt(cursor) == ',' ||
    fileContent.charAt(cursor) == ' ')) {
                    cursor++;
                }
                break;
            } else {
                buffer.append(currentChar);
            }
            cursor++;
        }
        return buffer.toString();
    }

    public TreeNode createNode(String buffer) {
        if (isNumeric(buffer)) {
            return new ConstNode(Integer.parseInt(buffer));
        } else {
            switch (buffer) {
                case "ET":
                    return new ET(createNode(parseFile()), createNode(parseFile()));
                case "OU":
                    return new OU(createNode(parseFile()), createNode(parseFile()));
                case "NON":
                    return new NON(createNode(parseFile()));
                case "SUP":
                    return new SUP(createNode(parseFile()), createNode(parseFile()));
                case "SUPEQ":
                    return new SUPEQ(createNode(parseFile()), createNode(parseFile()));
                case "EQ":
                    return new EQ(createNode(parseFile()), createNode(parseFile()));
                case "ADD":
                    return new ADD(createNode(parseFile()), createNode(parseFile()));
                case "SUB":
                    return new SUB(createNode(parseFile()), createNode(parseFile()));
                case "MUL":
                    return new MUL(createNode(parseFile()), createNode(parseFile()));
                case "SI":
                    return new SI(createNode(parseFile()), createNode(parseFile()),
    createNode(parseFile()));
                case "COMPTER":
                    return new COMPTER(parseFile());
                default:
                    System.out.println("Error: Invalid operator");
                    return null;
            }
        }
    }
}
```

Listing 16 – Parsing de la règle d'évolution

9 Main et autres

La classe `Main` est responsable de lancer et de gérer la simulation des automates cellulaires en fonction des dimensions de la grille spécifiée dans un fichier de configuration XML. La méthode `main` crée une instance de `Main` et appelle la méthode `startApplication`. Cette dernière utilise un objet `TOOLS_ConfigLoader` pour demander le chemin d'un fichier de configuration XML à l'utilisateur ou utilise une configuration par défaut si aucun fichier n'est sélectionné (le Random2D qui est le fichier demandé avec un RANDOM(15)). La classe de choix du xml est faite à partir d'un `JFileChooser` ou l'on applique un filtre d'extension `FileNameExtensionFilter("Fichiers XML", "xml");`.

```
    public class TOOLS_ConfigLoader {

    public String askForFilePath() {
        JFileChooser fileChooser = new JFileChooser();
        fileChooser.setDialogTitle("Selectionner le fichier de configuration");

        // Ajouter un filtre pour les fichiers XML
        FileNameExtensionFilter xmlFilter = new FileNameExtensionFilter("Fichiers XML", "
    xml");
        fileChooser.setFileFilter(xmlFilter);

        fileChooser.setCurrentDirectory(new File("./configs"));

        int userSelection = fileChooser.showOpenDialog(new JFrame());

        if (userSelection == JFileChooser.APPROVE_OPTION) {
            File fileToOpen = fileChooser.getSelectedFile();
            return fileToOpen.getAbsolutePath();
        }
        return null;
    }
}
```

Listing 17 – TOOLS_ConfigLoader

Ensuite, la méthode `config` lit et parse le fichier XML, initialisant la grille (`grid`), les règles d'évolution (`evolutionRule`), et les voisinages customs si spécifiés en utilisant les classes de parsing précédemment expliquées. Elle utilise un `Document` et récupère les noeuds du fichier XML. Par un switch, ces noeuds sont gérés.

```
   public void config(String path) throws ParserConfigurationException, SAXException {
        try {
            Document document = db.parse(file);
            document.getDocumentElement().normalize();
            NodeList nList = document.getDocumentElement().getChildNodes();

            for (int temp = 0; temp < nList.getLength(); temp++) {
                Node nNode = nList.item(temp);
                if (nNode.getNodeType() == Node.ELEMENT_NODE) {
                    Element eElement = (Element) nNode;

                    switch (eElement.getNodeName()) {
                        case "Dimension":
                            System.out.println("Dimension : " + eElement.getTextContent()
    );
                            break;
                        case "GridSize":
                            NodeList sizes = eElement.getElementsByTagName("Size");
                            System.out.print("GridSize : ");
                            int[] gridSize = new int[sizes.getLength()];
                            for (int i = 0; i < sizes.getLength(); i++) {
                                gridSize[i] = Integer.parseInt(sizes.item(i).
    getTextContent());
                                System.out.print(sizes.item(i).getTextContent() + " ");
                            }
                            grid = new STRUCT_Grid_ND(gridSize);
                            System.out.println();
                            break;
                        case "Cut":
```

```
                            String[] parts = eElement.getTextContent().split(" ");
                            int axis = -1;
                            switch (parts[0].toLowerCase()) {
                                case "x":
                                    cut[0] = 0;
                                    break;
                                case "y":
                                    cut[0] = 1;
                                    break;
                                case "z":
                                    cut[0] = 2;
                                    break;
                                default:
                                    throw new IllegalArgumentException("Invalid axis. Use 'x', 'y', or 'z'.");
                            }
                            cut[1] = Integer.parseInt(parts[1]);
                            System.out.println("Cut : " + eElement.getTextContent());
                            break;
                        case "CustomNeighborhoods":
                            NodeList neighborhoods = eElement.getElementsByTagName("Neighborhood");
                            for (int i = 0; i < neighborhoods.getLength(); i++) {
                                new TOOLS_NeighborhoodParser().parseFile(neighborhoods.item(i).getTextContent());
                                System.out.println("Neighborhood : " + neighborhoods.item(i).getTextContent());
                            }
                            System.out.println(new TOOLS_Neighborhoods().toString());
                            break;
                        case "EvolutionRule":
                            evolutionRule = new TOOLS_EvolutionRule(eElement.getTextContent(), false);
                            System.out.println("EvolutionRule : " + eElement.getTextContent());
                            break;
                        case "InitialCells":
                            if (eElement.getElementsByTagName("Random").getLength() > 0) {
                                NodeList k = eElement.getElementsByTagName("Random");
                                grid.initializeCells(Integer.parseInt(k.item(0).getTextContent()));
                            } else {
                                NodeList cells = eElement.getElementsByTagName("Cell");
                                for (int i = 0; i < cells.getLength(); i++) {
                                    String[] coordinates = cells.item(i).getTextContent().split(",");
                                    int[] cellCoordinates = new int[coordinates.length];
                                    for (int j = 0; j < coordinates.length; j++) {
                                        cellCoordinates[j] = Integer.parseInt(coordinates[j].trim());
                                    }
                                    grid.getCell(cellCoordinates).setCellValue(true);
                                    System.out.println("Cell : " + cells.item(i).getTextContent());
                                }
                            }
                        default:
                            break;
                    }
                }
            }
        } catch (IOException e) {
            System.out.println(e);
        }
    }
```

Listing 18 – Méthode config de Main

Ensuite, selon les dimensions de la grille (1D, 2D, ou 3D), la méthode appropriée pour la simulation est appelée : `run1DSimulation`, `run2DSimulation`, `run3DCutSimulation`, ou `run3DSimulation`. Pour des dimensions supérieures à trois, la méthode `runNDSimulation` est utilisée, bien qu'elle ne possède pas d'affichage graphique. Lorsque l'on a de la 3D, la demande de la coupe 2D est faite dans le terminal.

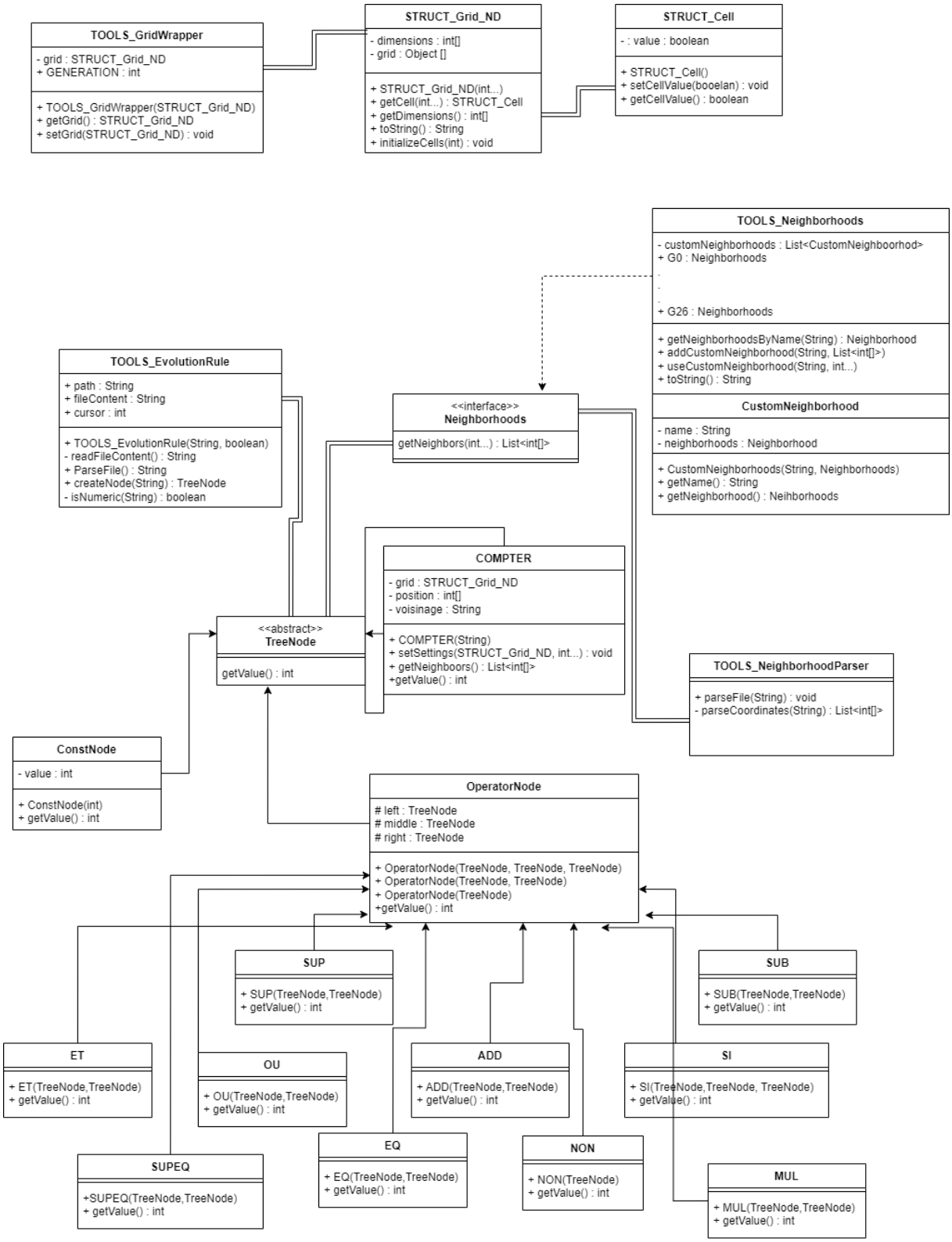
Chaque méthode de simulation met à jour la grille de cellules en fonction des règles d'évolution et met à jour l'affichage graphique correspondant. Pour les simulations 1D et 2D, les méthodes `run1DSimulationStep` et `run2DSimulationStep` gèrent les étapes de la simulation. Pour les simulations 3D, la classe `GFX_Cube` est utilisée pour l'affichage et la manipulation des cubes représentant les cellules. Pour ce faire, on lance une instance de l'application `GFX_Cube` en lui passant la grille.

Pour gérer les simulations, on a une classe `GFX_Start` qui ouvre dans une fenêtre un contrôleur de simulation (en 1D et 2D), en 3D, le contrôle est inclus dans la fenêtre.

```
public class GFX_Start extends JFrame {

    private boolean simulationRunning = false;
    private JButton startStopButton;

    public GFX_Start() {
        setTitle("Simulation Control");
        setSize(200, 100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        startStopButton = new JButton("Start");
        startStopButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                simulationRunning = !simulationRunning;
                startStopButton.setText(simulationRunning ? "Stop" : "Start");
            }
        });
        add(startStopButton);
    }

    public boolean isSimulationRunning() {
        return simulationRunning;
    }
}
```

Listing 19 – GFX_Start

10 Diagramme de classes



11 Question Bonus

Pour cette question bonus, nous devions trouver une figure fractale ou chaotique. Nous avons utilisé une règle défini par les deux cases à gauche de la case courante et les deux cases de droite (<Neighborhood>G00 = ((-2),(-1),(1),(2))</Neighborhood>). La figure obtenue ne doit pas être obtenu avec un voisinage défini seulement par la case de gauche, la case courante et celle de droite.
Nous avons cherché petit à petit en remplaçant le voisinage G00 dans la règle de Sierpiński ainsi, nous avons trouvé un Sierpiński "étalé" :

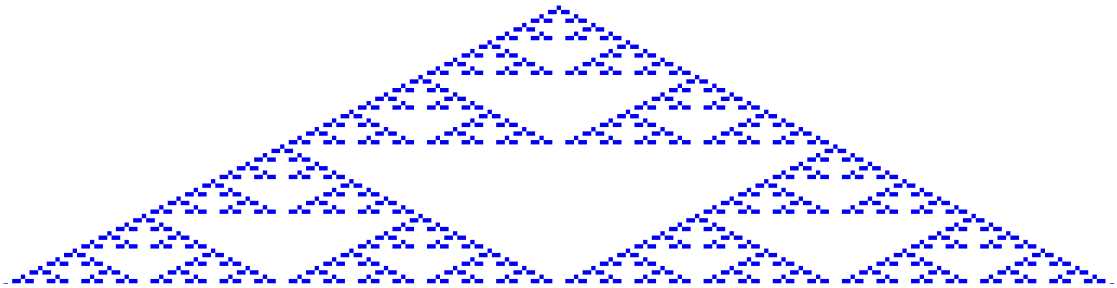


FIGURE 1 – Règle utilisée : **SI(EQ(COMPTER(G00),1), SI(EQ(COMPTER(G0),0), 1, 0), 0)**

Cependant, nous avons bien relu l'énoncé qui empêche d'avoir le **G0** dans notre règle donc nous avons continuer à chercher et avons remplacé le **G0** par le **G04** à savoir la case tout à droite de la cellule centrale. Ainsi, nous avons découvert une figure chaotique :

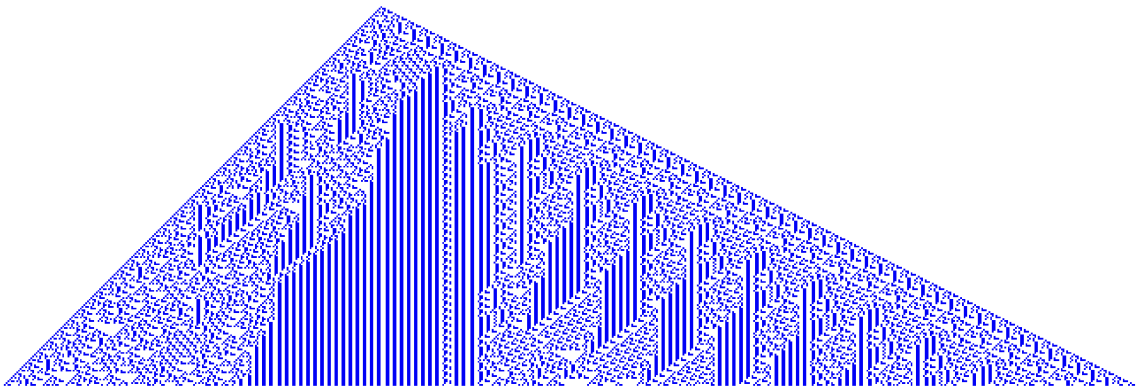


FIGURE 2 – Règle utilisée : **SI(EQ(COMPTER(G00),1), SI(EQ(COMPTER(G04),0), 1, 0), 0)**

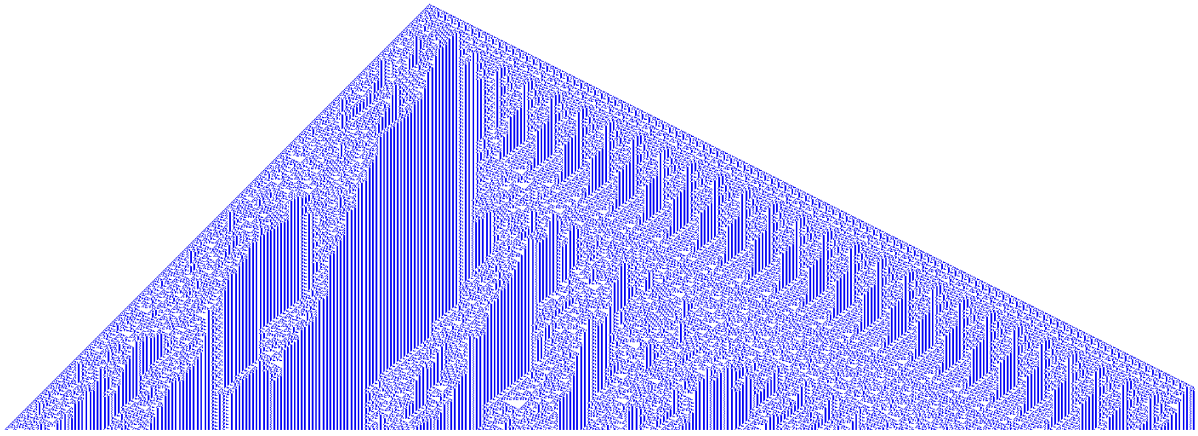


FIGURE 3 – Règle utilisée : **SI(EQ(COMPTER(G00),1), SI(EQ(COMPTER(G04),0), 1, 0), 0)**

On augmente le nombre de générations pour apercevoir comme un dessin chaotique qui se forme à gauche et pourtant le côté en haut à droite de la pyramide semble être régulier. Si vous souhaitez tester cette configuration elle est dans **"/configs/Chaotic.xml"** vous pouvez changer la dimension et la cellule de départ. Pour modifier le nombre de générations c'est dans la classe Main avec la variable **"height"**.

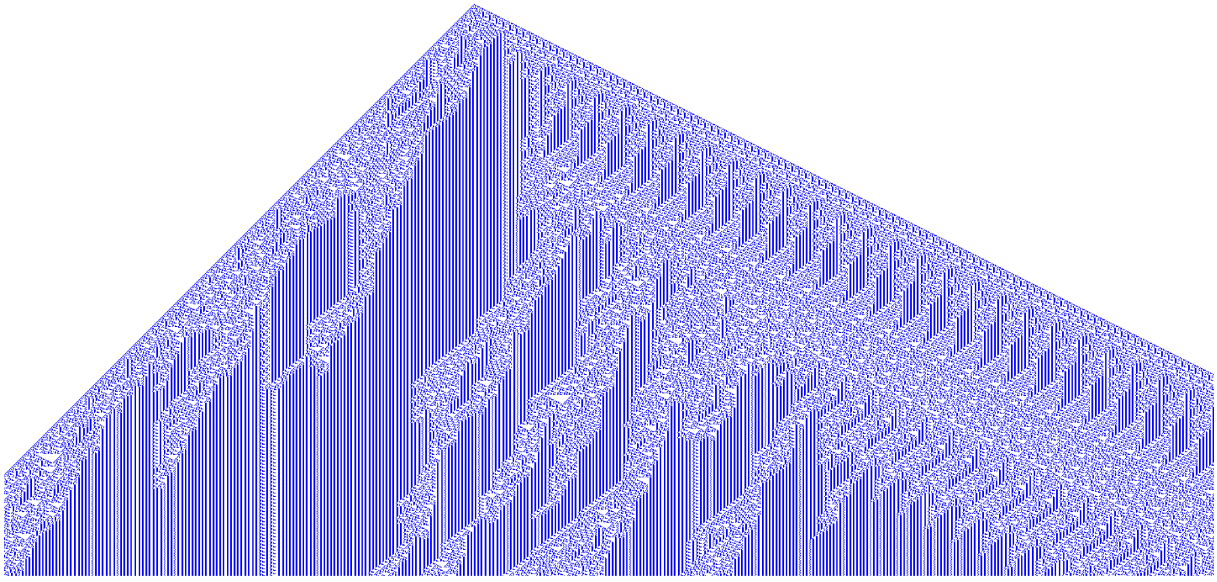


FIGURE 4 – Règle utilisée : **SI(EQ(COMPTER(G00),1), SI(EQ(COMPTER(G04),0), 1, 0), 0)**

12 Limites du projet

Malgré notre forte motivation et notre envie de rendre un projet parfait, nous nous sommes aperçu de quelques limites de notre implémentation. En premier, la partie des coupes en N dimensions n'a pas été effectuée par manque de compréhension de la méthode. Ensuite, nous pensons que notre façon d'implémenter les voisinages amène des ralentissements au bout de quelques centaines de générations mais nous ne savons pas exactement pourquoi. Nous aurions bien aimé faire la cas continu mais la rédaction du rapport et la finalisation du projet nous a pris pas mal de temps ; une prochaine fois !

13 Conclusion

De la première lecture du projet jusqu'aux derniers mots de ce rapport, nous avons pris énormément de plaisir à nous renseigner sur les automates cellulaires et sur la recherche qu'il y a encore derrière. Cette motivation nous a permis de prendre du plaisir à le coder nous même malgré les difficultées rencontrés qui vous ont précedemment été exposés. Le pouvoir de simulation et de création est aussi très stimulant et ce "jeu de la vie" en est un très bon exemple. Merci d'avoir lu notre rapport et merci pour ce très beau projet !

FIN