LO21 : Projet AutoCell

Printemps 2018

Hugo Le Moine
Hugo Paigneau
Thomas Le Gluher

Chargé de TD : Gildas Bayard

Sommaire

Description du contexte **2**
Qu’est ce qu’un automate cellulaire ? 2
Nos cinq automates 2
Automate 1D 2
Jeu de la Vie 2
Feu de forêt 3
Automate cyclique de D. Griffeath 3
Brian's Brain 4

Description de l’architecture **5**
Architecture de l’automate cellulaire 5
Architecture des interfaces homme-machine 7
AutoCell1D 7
AutoCell2D 8

Implémentation d’un nouveau type d’automate **9**

Annexes **10**
Instructions de compilation 10
UML Partiel : interfaces des automates implémentés 11
Présentation des livrables 11

Description du contexte

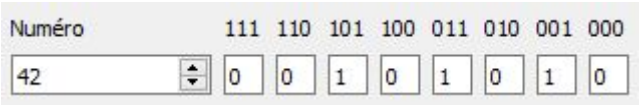
Qu’est ce qu’un automate cellulaire ?

Un automate cellulaire consiste en une grille de cellules. Une cellule peut avoir plusieurs états. Les cellules changent d'état au cours du temps. L'état d'une cellule à t+1 dépend de l'état des cellules de son voisinage au temps t et de la règle de l'automate. Pour faire avancer l'automate d'une unité de temps, on applique la règle de transition à toutes les cellules de l'état de départ simultanément. On obtient alors un nouvel état, sur lequel on peut de nouveau appliquer cette transition, et ainsi de suite...
Il existe beaucoup de types d'automates cellulaires à dimensions, nombres d'états, types de voisinages et règles de transition différentes.
Dans ce projet, nous avons implémenté cinq automates cellulaires.

Nos cinq automates

Automate 1D

Cet automate est le plus simple des automates cellulaires intéressants. Il est à une dimension et ses cellules peuvent prendre deux états (noir et blanc). Pour déterminer l'état d'une cellule après transition, on regarde l'état de ses deux voisines (droite et gauche) ainsi que le sien. Dans notre implémentation, il est possible de choisir la règle en saisissant un 1 ou un 0 pour chacun des 8 voisinages possibles. On peut également choisir la règle en indiquant un nombre entre 0 et 255, qui redonne les valeurs des 8 possibilités une fois traduit en binaire.

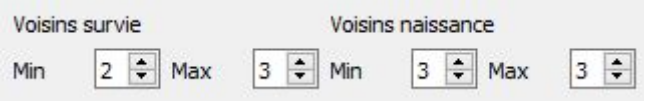


Exemple avec la règle n°42

Jeu de la Vie

L'automate du Jeu de la Vie est un des plus connu des automates cellulaires, car des comportements très intéressants émergent de règles simples. C'est un automate sur deux dimensions, et ses cellules peuvent prendre deux états (morte ou vivante). L'état futur d'une cellule est déterminé par son état actuel et par le nombre de cellules vivantes parmi les huit qui l'entourent (voisinage de Moore). A l’origine, le Jeu de la Vie tel qu’il l’a été imaginé par J. Conway, avait les règles suivantes : une cellule vivante ne survit que si elle est entourée de 2 ou 3 cellules vivantes, et une cellule naît si elle possède exactement 3 voisins. L’aspect remarquable de cet

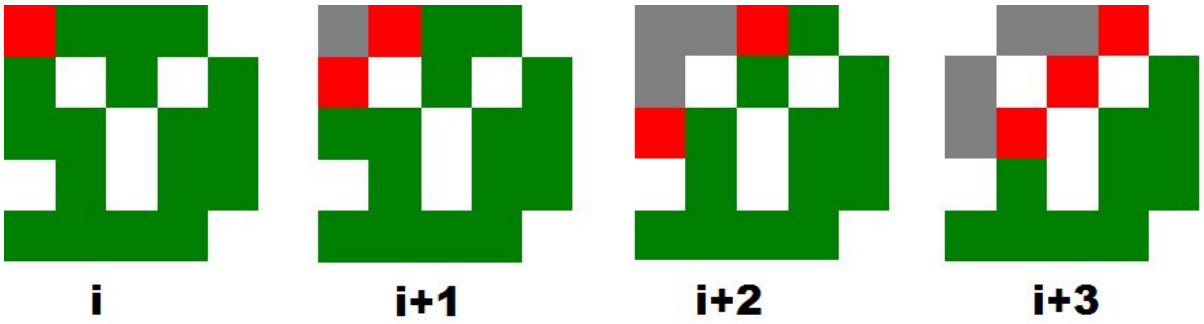
automate est qu'il est Turing-complet : tout algorithme peut alors être calculé, il est donc possible de programmer le jeu dans le jeu. Dans notre implémentation, il est possible de choisir les conditions en saisissant quatre nombres qui représentent le minimum et le maximum de cellules du voisinage nécessaires à la naissance ou à la survie d'une cellule. Initialement cette configuration est celle du Jeu de la Vie traditionnel



Configuration initiale du jeu de la vie

Feu de forêt

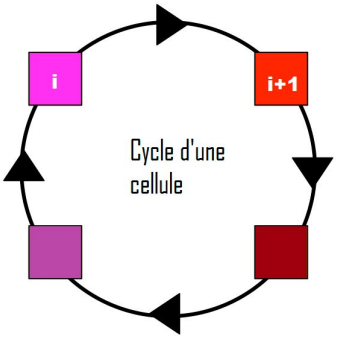
Cet automate cellulaire permet de représenter de façon simple la façon dont se propage un feu dans une forêt. Il est en deux dimensions, et ses cellules peuvent prendre quatre états différents (arbre, feu, cendre et vide). Le voisinage choisi est le voisinage de Von Neumann, c'est à dire que l'on regarde les cellules à gauche, droite, haut et bas de la cellule qui nous intéresse. Les règles de transitions sont simples. Une case arbre avec au moins un voisin en feu devient en feu, une case en feu devient en cendre, une case en cendre devient vide si elle n'a pas de voisin en feu, et une case vide reste vide.
Pour cet exemple les arbres sont verts, le feu est rouge, les cendres sont grises et le vide est blanc.



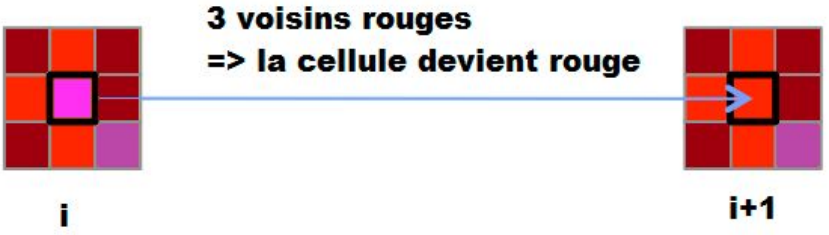
Automate cyclique de D. Griffeath

Cet automate cellulaire auto-reproducteur est en deux dimensions, et ses cellules peuvent prendre quatre etats (rouge, marron, pourpre, violet). L'état d'une cellule au temps t+1 dépend de son état au temps t et de l'état de ses 8 voisines (voisinage de Moore). Une cellule passe d'un état i à un état i+1 dans le cycle d'états lorsque l'état i+1 est présent dans au moins trois cellules voisines.

Pour notre implémentation, le cycle d'état est le suivant.

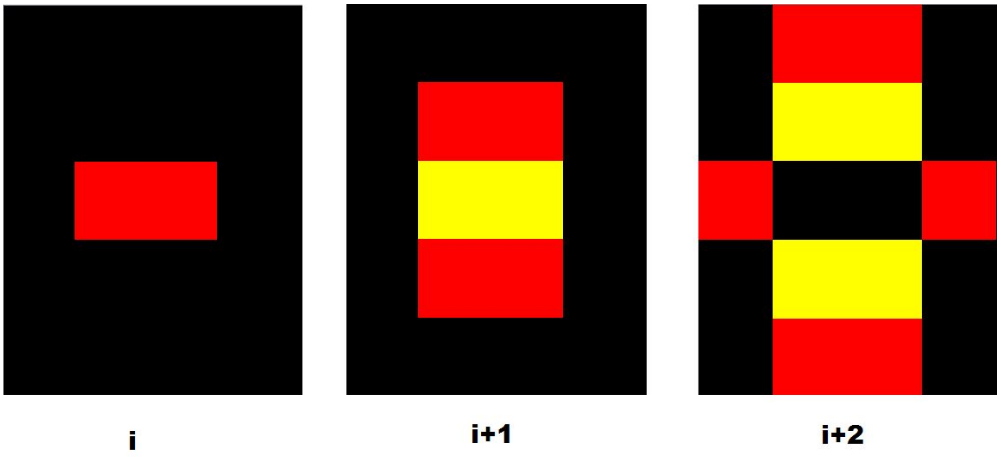


Une transition possible est par exemple



Brian's Brain

L'automate cellulaire "Brian's Brain" est un automate en deux dimensions, et ses cellules peuvent prendre trois états (vivante, mourante et morte). Le calcul de l'état après transition d'une cellule est effectué avec le voisinage de Moore. Une cellule naît si elle a exactement deux voisines vivantes, elle devient mourante si elle était en vie, et meurt si elle était mourante. Dans cet exemple, les cellules vivantes sont rouges, les cellules mourantes sont jaunes, et les cellules mortes sont noires.



Trois états successifs en partant de deux cellules vivantes

Description de l'architecture

Architecture de l'automate cellulaire

Les trois composants essentiels aux fonctionnement d'un automate sont les classes automate, état et simulateur.

Les classes **Etat1D** et **Etat2D** sont des classes qui permettent de représenter l'état d'un automate cellulaire à une dimension ou à deux dimensions respectivement. Ces deux classes héritent d'une classe abstraite **EtatMere**.

Les classes **Automate1D**, **Automate2D**, **Automate2DFeu**, **Automate2DG** et **Automate2DBrian** représentent les cinq automates que nous avons implémentés. Elles héritent toutes d'une classe abstraite **AutomateMere**. La classe **AutomateMere** utilise le design pattern template pour sa méthode appliquerTransition qui prend en paramètres deux états qui peuvent être 1D ou 2D.
Les classes **EtatMere** et **AutomateMere** utilisent aussi le design pattern template method qui permet de définir le squelette de méthodes qui sont implémentées dans leurs classes filles (par exemple appliquerTransition).

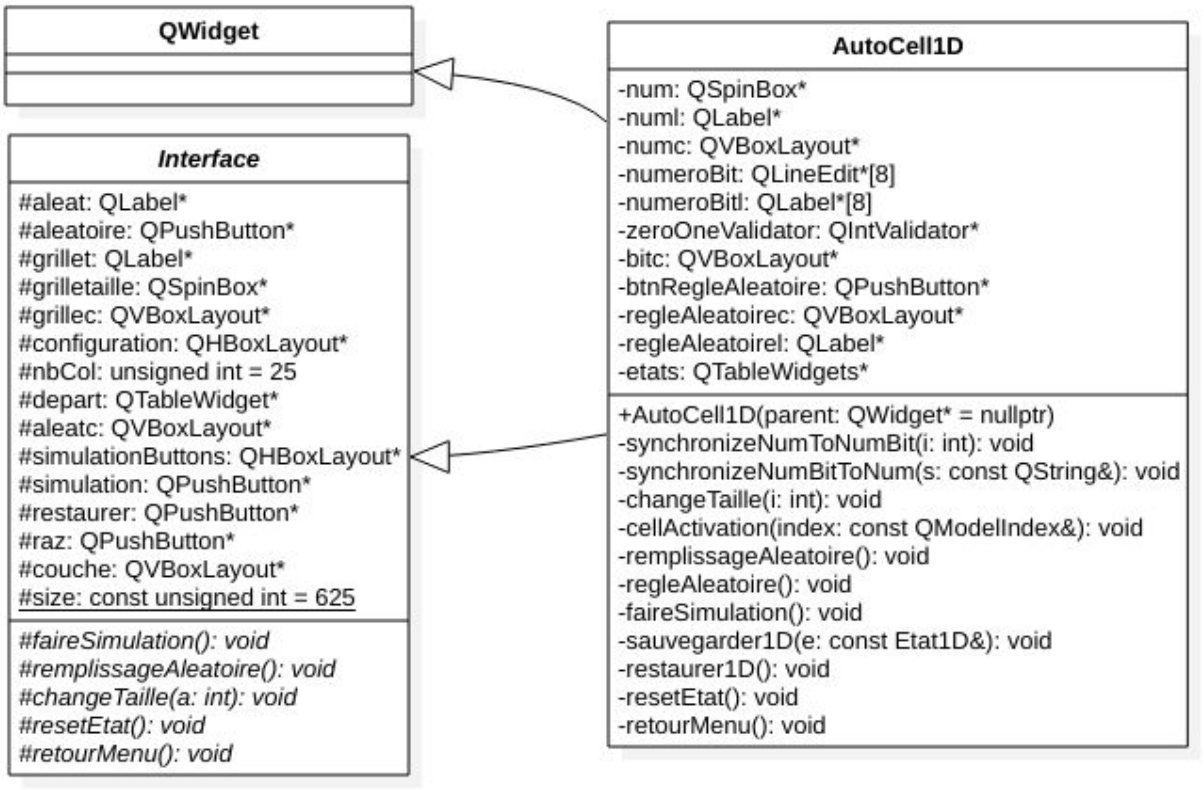
Le simulateur est l'instance qui va se charger de calculer les états suivants à partir d'un état de départ et de la règle de transition d'un automate. C'est pourquoi un simulateur a en attributs un **Etat** `depart`, un tableau d'**Etats** successifs `etats` et un **Automate** `automate`. C'est la méthode `next()` du simulateur qui applique la transition sur l'état de courant et ajoute l'état généré au tableau `etats`. Les états et les automates que manipule le simulateur peuvent être en une ou en deux dimensions, c'est pourquoi on a utilisé le design pattern template sur cette classe.

UML présentant les interactions entre Simulateur, Automate et Etat et l'héritage des états et des automates

Architecture des interfaces homme-machine

AutoCell1D

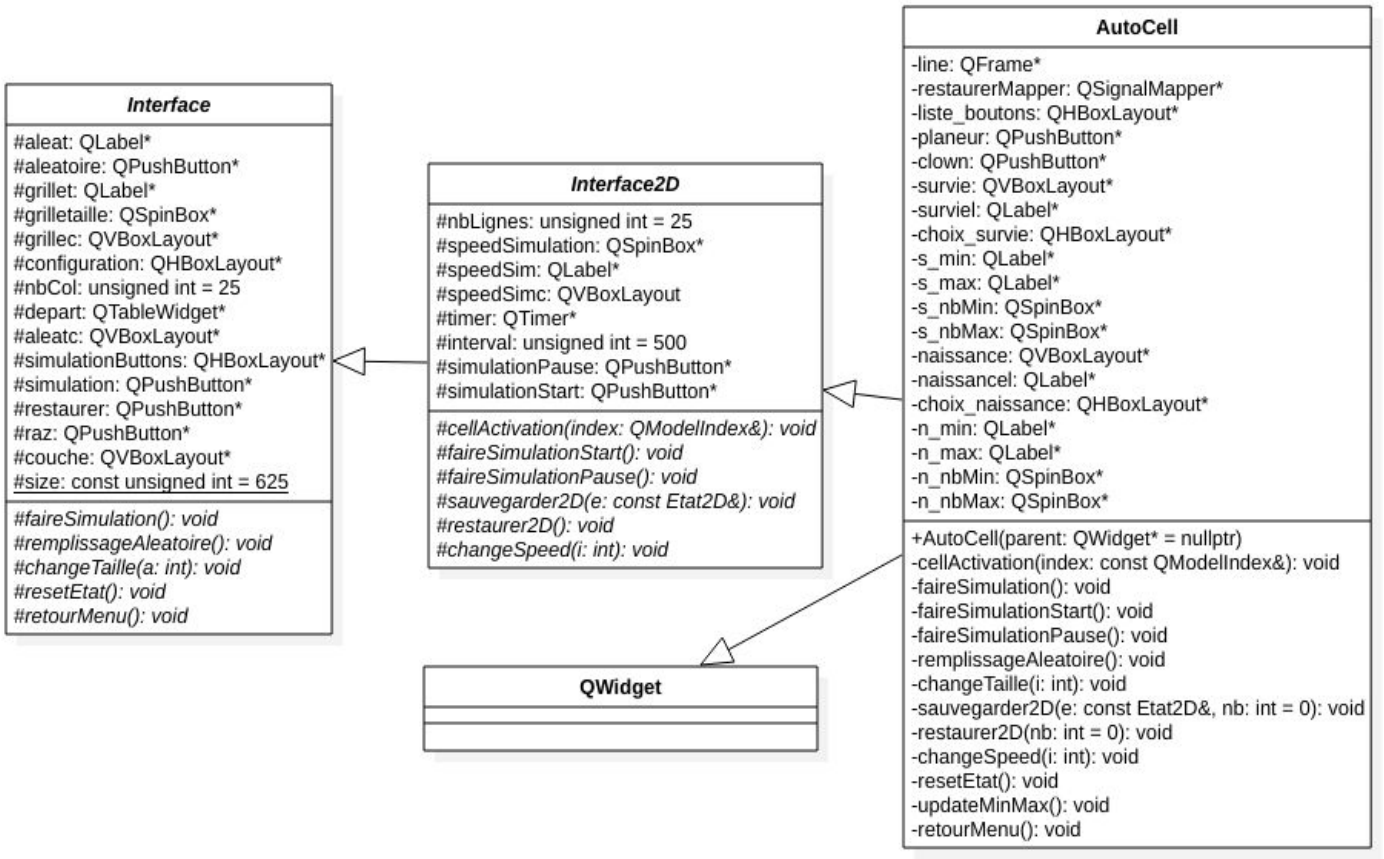
Pour interagir avec l'utilisateur, nous avons créé des composants graphiques, les **AutoCells**. Ils héritent de la classe **QWidget** (pour profiter des fonctionnalités d'affichage de Qt) et de la classe abstraite ***Interface***, qui factorise les éléments qu'il y a dans tous les **AutoCell** (ex : bouton pour démarrer la simulation). Ici l'interface graphique pour l'automate 1D hérite directement d'***Interface***. Etant donné qu'il n'est pas possible de sous-classer une classe dérivant de QWidget, chaque implémentation d'une interface doit dériver de QWidget également.



UML partiel : classe AutoCell1D (interface pour l'automate à une dimension).

AutoCell2D

Pour les interfaces utilisateurs des automates 2D, nous avons ajouté un niveau de factorisation de code supplémentaire, avec la classe abstraite ***Interface2D*** qui hérite d'***Interface*** et qui y ajoute les éléments communs à toutes les interfaces 2D (ex: bouton pour la vitesse de simulation). Ici l'interface pour le Jeu de la Vie hérite de la classe ***Interface2D***.



UML partiel : classe AutoCell (interface du Jeu de la Vie).

Vous pourrez trouver un diagramme UML plus complet en annexe montrant les implémentations des interfaces pour les automates Feu de forêt, Griffeath et Brian's Brain.

Implémentation d’un nouveau type d’automate

Lors de ce projet, nous avons favorisé l’héritage. Ainsi, celui-ci permet de donner à une classe toutes les caractéristiques d’une ou de plusieurs autres classes. Les classes mères **EtatMere** et **AutomateMere** sont des bases pour l’implémentation de futurs types d’état ou d’automates. A titre d’exemple, on pourrait ajouter facilement un automate utilisant d’autres règles de voisinage (Margolus, etc..). Donc, comment se passe l’implémentation d’un autre type d’automate ?

+ Optionnel, on peut créer un type d’état (par exemple **Etat3D** ou un différent de ceux déjà présents, bien que **Etat2D** soit la base de beaucoup d’automates).
+ On créer une nouvelle classe d’automate héritant de AutomateMere :

class **AutomateNew** : public **AutomateMere**<*EtatChoisi*> {...}

On a plus qu'à définir notre automate dans cette classe. A l’aide des template, on a une souplesse accrue ! On a simplement à définir un template <class T>.

+ Une fois fini, on ajoute dans la partie du simulateur.cpp un “template class **Simulateur**<*AutomateNew*,*EtatChoisi*>;”
+ Pour finir, il ne nous reste plus qu'à implémenter dans la partie Autocell, la composante “graphique” de notre automate. Celle-ci est encore une fois très simple puisqu’il existe une classe mère Interface et **Interface2D**. On créer donc une classe héritant de **Interface2D** ET de QWidget.

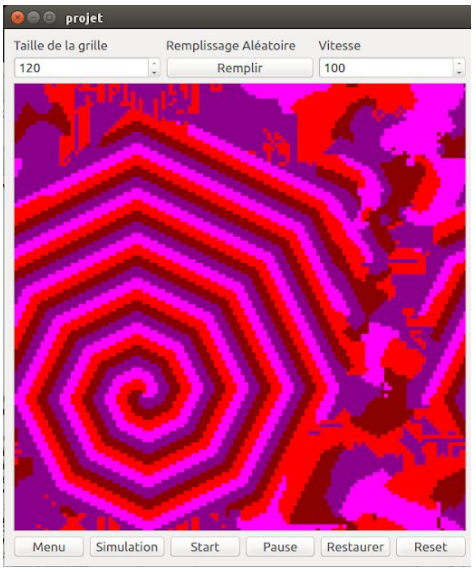
De plus, on a une grosse souplesse et flexibilité dans l’utilisation de divers automates à l’aide des templates (patron de fonctions). C’est un modèle d’une fonction dont le type de retour et le type des arguments n’est pas fixé. Leur utilisation permet donc de gagner en performances, en temps de codage, et surtout en clarté.

Annexes

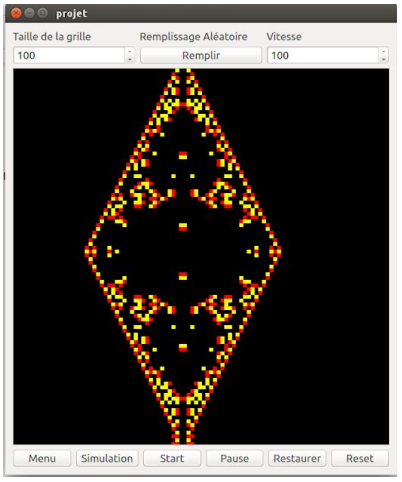
Instructions de compilation

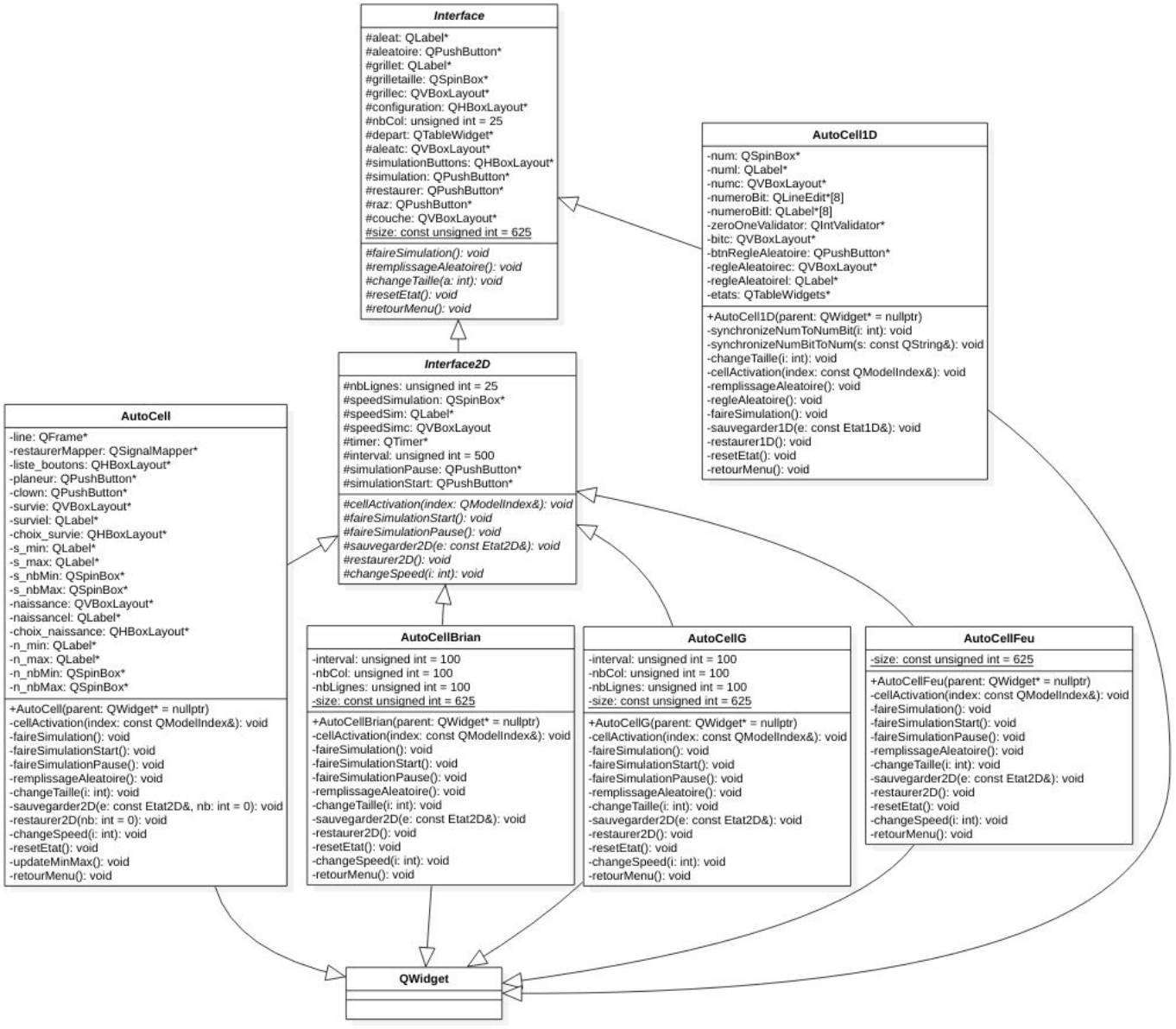
Le répertoire d'exécution doit être placé au même niveau que le dossier “Code”, ou alors que l'exécutable soit dans ce même dossier “Code”, afin que les sauvegardes et restaurations se fassent correctement (notamment pour les fichiers de configuration intégrés au programme).

Concernant les automates, l’automate 2D Griffeath, avec une bonne configuration (et de la chance, ou un ordinateur très puissant [grille de 200x200]), on est censé observer un état cyclique permanent :



Pour ce qui est de l’automate Brian’s Brain, la manière la plus intéressante de le faire tourner est de placer deux blocs rouges côte à côte :





UML Partiel : interfaces des automates implémentés

Présentation des livrables

- Vidéo de présentation des fonctionnalités du programme (à la racine du zip)
- Rapport composé de 3 parties et d'annexes
- Code source (dans le répertoire Code)
- Documentation en html avec *Doxygen* (index.html dans le répertoire doc)