GPA675

Laboratoire 1 – Automate cellulaire

Objectif principal

Par ce projet, on vise d'abord une reprise de contact avec la programmation C++ pour ensuite réaliser une application implémentant un automate cellulaire utilisant quelques structures de données de type tableau et des algorithmes dédiés.

Objectifs spécifiques

Plus spécifiquement, ce projet vise les objectifs suivants :

- reprise de contact avec :
  - programmation C++
  - programmation orientée objet
- utilisation potentielle des structures de données suivantes de la librairie standard :
  - std::vector
  - std::array
- développement de la classe **Grid** réalisant une gestion d'une grille 2d s'appuyant sur une gestion manuelle de la mémoire
- développement d'une classe réalisant un engin complet de l'automate cellulaire 2d
- développement de quelques algorithmes simples permettant:
  - la résolution des problématiques exposées
  - l'exploitation de techniques effectuant les meilleurs compromis entre :
    - fonctionnalité
    - modularité
    - performance

Automate cellulaire 2d

Définition générique

> *Un automate cellulaire consiste en une grille régulière de « cellules » contenant chacune un « état » choisi parmi un ensemble fini et qui peut évoluer au cours du temps. L'état d'une cellule au temps t+1 est fonction de l'état au temps t d'un nombre fini de cellules appelé son « voisinage ». À chaque nouvelle unité de temps, les mêmes règles sont appliquées simultanément à toutes les cellules de la grille, produisant une nouvelle « génération » de cellules dépendant entièrement de la génération précédente.*[1]

Définition précise pour ce projet

Plus spécifiquement, nous utiliserons pour ce laboratoire le jeu de la vie développé par John Conway.

[1] Wikipedia

> *Le « jeu de la vie » est un automate cellulaire bidimensionnel où chaque cellule peut prendre deux valeurs (« 0 » ou « 1 », mais on parle plutôt de « vivante » ou « morte ») et où son état futur est déterminé par son état actuel et par le nombre de cellules vivantes parmi les huit qui l'entourent.*
>
> *Si la cellule est vivante et entourée par deux ou trois cellules vivantes, elle reste en vie à la génération suivante, sinon elle meurt.*
>
> *Si la cellule est morte et entourée par exactement trois cellules vivantes, elle naît à la génération suivante.*
>
> *En apparence simple, ces règles font émerger une forte complexité.*[2]

Malgré tout, on précisera les éléments suivants pour ce projet :

- Le monde est constitué d'une grille à deux dimensions où se retrouve un nombre fini de cellules. La taille du monde est fixée au début de la simulation et immuable pour la suite de la simulation.
- Chaque cellule possède 2 états possibles (vivante ou morte).
- On généralisera le concept d'évolution ainsi :
    - à chaque étape de simulation, on applique la même logique de calcul pour chacune des cellules de façon à faire évoluer ces dernières;
    - l'état futur d'une cellule dépend de son état courant et de l'état de ses 8 voisins;
    - pour les voisins, on ne considère que le voisinage immédiat à 8 éléments (orthogonaux et diagonaux) et on ne retiendra que le nombre de cellules vivantes;
    - au début de la simulation, on détermine la règle, c'est-à-dire pour quelle quantité de voisins vivants une cellule est vivante à l'itération suivante (on détermine ces conditions 2 fois, une fois pour les cellules qui sont mortes et une autre pour les cellules qui sont vivantes);

Règles

Le Jeu de la vie applique la règle B3/S23. Cette représentation indique que :

- une cellule morte naît (***B**orn*) si seulement 3 voisins sont vivants;
- une cellule vivante survie (***S**urvive*) si seulement 2 ou 3 voisins sont vivants.

La règle Day & Night applique la règle B3678/S34678, qui indique que :

- une cellule morte naît (***B**orn*) si seulement 3, 6, 7 ou 8 voisins sont vivants;
- une cellule vivante survie (***S**urvive*) si seulement 3, 4, 6, 7 ou 8 voisins sont vivants.

En fait, on comprend qu'il existe plusieurs règles possibles et que ce projet doit offrir une flexibilité à cet égard. Plus précisément, il faut offrir la possibilité de réaliser les 725 760 combinaisons possibles.

[2] Wikipedia

Les règles sont encodées selon ce patron :

B*/S@

- **B** : un premier caractère obligatoire (minuscule ou majuscule)
- * : une suite *n* caractères différents compris dans l'intervalle des caractères '0' à '8' (indiquant le nombre de voisins pour lesquels la cellule naît). *n* est de 0 à 9.
- / : un caractère obligatoire
- **S** : un caractère obligatoire (minuscule ou majuscule)
- @ : une suite *n* caractères différents compris dans l'intervalle des caractères '0' au '8' (indiquant le nombre de voisins pour lesquels la cellule survie). *n* est de 0 à 9.
- La règle ne supporte pas d'autres caractères (ni même l'espace) ou toute autre disposition.

Il doit être possible de déterminer la règle par une simple chaîne de caractères représentant ce patron. Aussi, si le patron n'est pas conforme, on ne modifie pas la règle en cours.

Patron

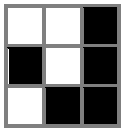
On désire pouvoir insérer un patron dans la simulation. Ces patrons seront définis selon le standard suivant :

[*x@]010101...

- [: un caractère obligatoire
- * : une suite *n* caractères compris dans l'intervalle '0' à '9' indiquant la taille horizontale du patron, la largeur. De plus, n > 0 et largeur > 0.
- **x** : un caractère obligatoire (minuscule ou majuscule)
- @ : une suite *n* caractères compris dans l'intervalle '0' à '9' indiquant la taille verticale du patron, la hauteur. De plus, n > 0 et hauteur > 0.
-] : un caractère obligatoire
- **01...** : une suite de 'largeur x hauteur' caractères compris dans l'intervalle '0' à '1' représentant l'état de chaque cellule du patron (répartis sur une seule ligne) '0' correspondant à mort et '1' à vivant. Toutes les lignes de la matrice 2d sont disposées de façon contiguë les unes après les autres.

Par exemple, le patron suivant représente un *glider*.

[3x3]001101011



Le centre du patron doit se retrouver sur la coordonnée donnée. Si le patron est plus grand que le monde, la partie invisible n'est simplement pas utilisée. Finalement, si le patron n'est pas conforme, la fonction quitte sans rien faire.

Mandat

On vous demande de créer un engin permettant de réaliser une telle simulation.

Outils mis à votre disposition et infrastructure imposée

Classes GOL et Grid

Vous avez à votre disposition une classe abstraite nommée **GOL**. Cette classe constitue une interface de programmation encapsulant toutes les tâches nécessaires à simuler un automate cellulaire. L'objectif est de réaliser une implémentation, par polymorphisme, d'une classe qui réalise cet engin de simulation.

En bref, cette classe permet :

- configurer l'engin
- réaliser les calculs de la simulation
- produire une image issue de la représentation interne.

Le fichier **GOL.h** où se trouve la classe **GOL** présente aussi, sous forme de commentaires, la classe **Grid** (voir la fin du fichier). Cette classe vise à réaliser une structure de données de type *tableau 2d* redimensionnable. Vous avez aussi l'obligation de réaliser cette classe.

Autrement dit, la classe **GOL** utilise, par composition, la classe **Grid** pour stocker l'information de la simulation.

Le projet consiste donc à développer deux classes :

- Une classe nommée **GOLTeam##** héritant de **GOL** correspondant au moteur principal du projet. L'utilisation de la classe **GOL** n'est pas optionnelle et vous ne devez en aucun cas modifier son fichier.
- Une classe utilitaire nommée **GridTeam##** représentant un tableau 2d. Cette classe sert d'appui à votre implémentation de **GOLTeam##** et vous aidera à réaliser la gestion 2d de la grille. Obligatoirement, cette classe doit gérer manuellement la mémoire.

Vous devez lire attentivement les nombreux points décris dans le fichier **GOL.h** pour bien comprendre le rôle et les contraintes de chaque fonction. Un petit site web documentaire est aussi disponible dans le dossier doc de la librairie (voir le fichier de démarrage index.html)

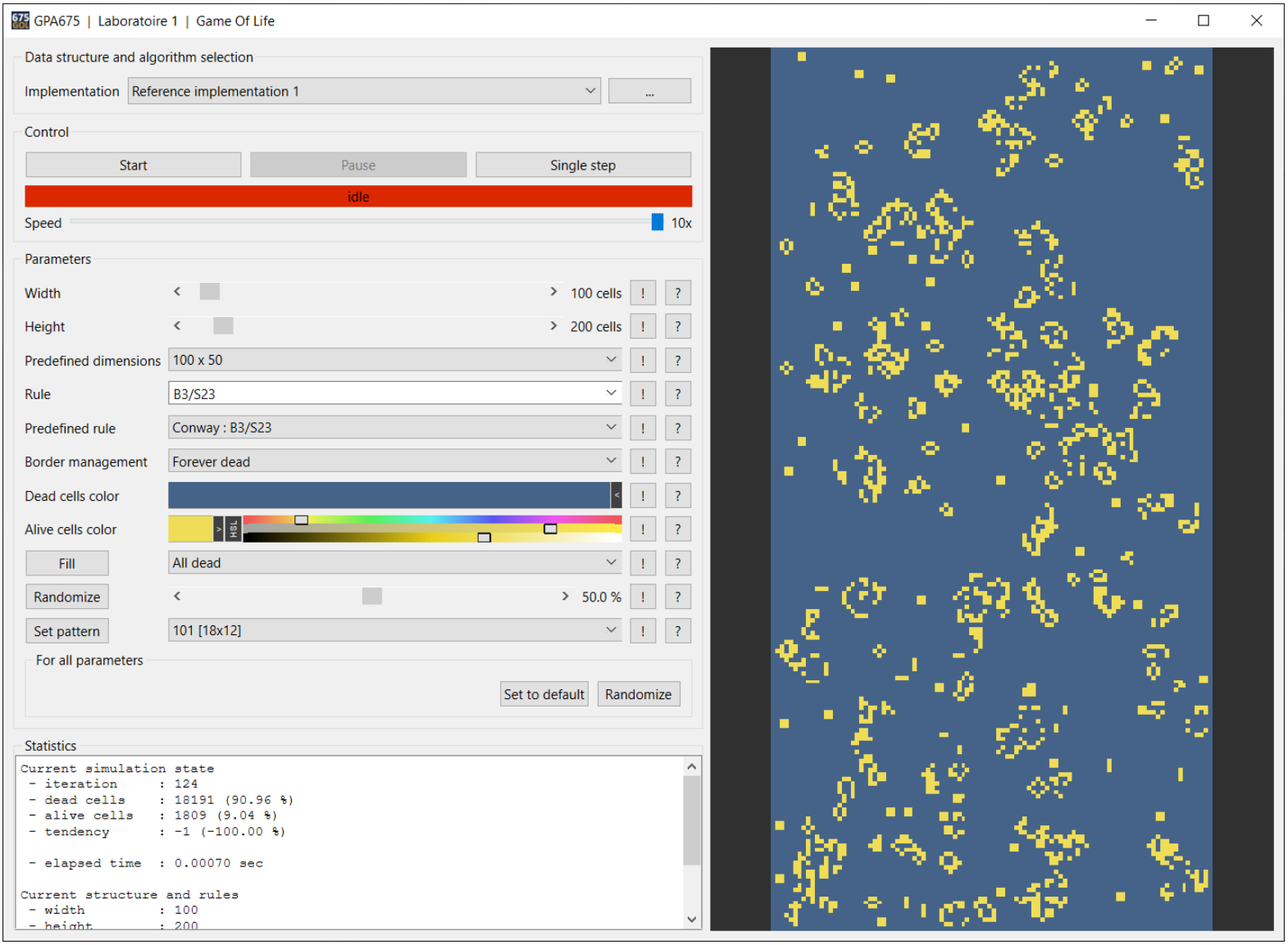
Logiciel de simulation

Vous avez à votre disposition un logiciel offrant une interface utilisateur graphique. Ce logiciel est réalisé en **C++** avec la librairie **Qt**.

Il permet l'exploitation de toutes les fonctionnalités attendues par la classe **GOL**. Il sera ainsi facile et agréable de tester votre code.

Ce logiciel est disponible sous forme de bibliothèque déjà montée dans une solution **Visual Studio**. La solution contient un seul projet dans lequel se trouve un seul fichier : **main.cpp**. Sans surprise, on y retrouve le **main** de l'application qui réalise le code classique d'une application **Qt**.

Après avoir créé votre implémentation de **GOL**, il suffit de l'intégrer dans l'application à même le **main** par la fonction **GOLApp::addEngine**. Voir l'exemple à même le code source du fichier **main.cpp**.



L'application se divise en trois parties principales :

- *Data structure and algorithm selection*
  - Cette section permet de choisir l'implémentation à utiliser parmi celles disponibles. Choisir une nouvelle implémentation réinitialise la grille avec les paramètres courants.
  - Nominalement, les implémentations de référence sont présentes et la vôtre s'y retrouvera après l'avoir ajoutée.
  - Si vous faites plusieurs implémentations (vraiment optionnel), vous pouvez toutes les ajouter. Seulement vous assurer que la bonne, celle qui sera évaluée, possède le nom attendu. Les autres ne seront pas évaluées.
- *Control*
  - Permets le démarrage et l'exécution de la simulation soit en continu ou pas-à-pas.
- *Parameters*
  - Permets la définition des paramètres de l'engin de simulation.

Robustesse

L'un des points les plus importants de votre développement est le fait que votre implémentation doit être robuste en tout temps. Ceci implique que votre classe doit garantir l'intégrité du modèle **en tout temps**! De sa création jusqu'à sa destruction en passant par n'importe quel appel de fonction à n'importe quel moment et dans n'importe quel ordre.

La robustesse est plus importante que le fait de réaliser tous les détails de l'implémentation.

Nommage

Vous devez utiliser le suffixe suivant : **Team##**

- Les caractères **##** représentent le numéro de votre équipe sur Moodle. Vous devez utiliser 2 chiffres. Donc l'équipe 1 se nommera : **Team01**

Vous devez utiliser ce suffixe pour :

- la classe réalisant l'engin de **GOL** :
  - le nom de votre classe : **GOLTeam01**
  - le titre des informations est : **GOLTeam01** (voir **ImplementationInformation**)
  - les noms des fichiers où se trouvent la déclaration et la définition de la classe :
    - **GOLTeam01.h**
    - **GOLTeam01.cpp**
- Pour la classe réalisant la grille :
  - le nom de votre classe: **GridTeam01**
  - les noms des fichiers où se trouvent la déclaration et la définition de la classe :
    - **GridTeam01.h**
    - **GridTeam01.cpp**

Information à remplir

La classe **GOL** possède un sous-type correspondant à un mini-rapport à produire. Ce rapport n'est pas technique dans le sens où on valide votre compréhension pour divers aspects. Un quiz servira à confirmer votre compréhension du projet. Il vise plutôt à bien présenter votre travail pour en faciliter son évaluation.

Il est vraiment important de comprendre que l'objectif n'est pas de rédiger un roman présentant tous les détails de votre implémentation. Vous devez obligatoirement utiliser le vocabulaire technique approprié tout en étant très concis et très précis. Vos réponses doivent être les plus brèves possible en présentant clairement les concepts demandés.

La structure **GOL::ImplementationInformation** possède les informations suivantes :

1. **title** : le titre de votre implémentation, par exemple **GOLTeam01**
2. **author** : les auteurs du projet
   - pour chaque étudiant de l'équipe :
     - **firstName** : prénom
     - **lastName** : nom de famille
     - **studentEmail** : courriel étudiant

3. **answers** : les réponses aux questions suivantes
    1. décrivez votre stratégie de gestion de la mémoire dynamique dans la classe **Grid**
    2. décrivez l'implémentation algorithmique de la fonction **GOL::processOneStep**
    3. décrivez l'implémentation algorithmique de la fonction **GOL::updateImage**
    4. décrivez comment vous avez géré l'application de la règle (pas sa validation, mais comment vous l'appliquer lors des calculs pour chaque pas de simulation)
    5. décrivez comment vous avez géré les cinq stratégies de gestion des bords.
4. **optionnalComments** : tous commentaires, suggestions ou coquilles reliées au projet

Suggestions quant à la réalisation de ce projet

Voici une suggestion de l'ordre de développement du projet :

1. Formation des équipes.
2. Mise en place des outils de développement :
    - Installation de **Visual Studio 2022** (faire la mise à jour si nécessaire).
    - Installer **Qt** (la version la plus récente).
    - Installer l'extension pour **Visual Studio** de **Qt**.
    - Faire un test créant un projet de type **Qt Widgets Application** sans rien de plus où une simple fenêtre apparaît lors de son exécution. On valide ainsi que l'infrastructure est fonctionnelle et qu'il est possible de compiler, de « *linker* » et d'exécuter le programme.
3. Vous assurer que la solution de **Visual Studio** qui vous est donnée est fonctionnelle. Vous devriez être capable de la compiler, de la « *linker* » et de l'exécuter.
4. Développer l'essentiel de la classe **Grid** :
    - Assurez-vous de bien comprendre son rôle et les parties importantes.
    - Faites une conception des éléments importants.
    - Appliquez le concept **CALTAL**.
    - Développer les parties essentielles pour vous permettre d'amorcer le développement de **GOL**.
    - Vous pouvez réaliser des tests de développement à même le **main** de l'application.
5. Mise en place de la classe GOL :
    - Créer votre classe héritant de la classe GOL.
    - Rédiger **toutes** les déclarations des substitutions de fonction (**override**).
    - Rédiger **toutes** les implémentations de ces fonctions en laissant les fonctions vides et, pour celles qui retournent de l'information, retourner l'équivalent d'un **false**.
    - Ajouter votre classe **GOL** dans l'application principale dans le **main** de l'application avec la fonction **GOLApp ::addEngine**.
    - Pour l'accesseur **GOL ::information() const**, assurez-vous de retourner quelques informations de base (au moins le titre) dans **GOL ::ImplementationInformation**.
    - Validez que tout compile et que vous avez une entrée vide dans la liste des implémentations.
6. Développement incrémental :
    - Ajouter des fonctionnalités les unes à la fois. Commencer avec celle qui vous semble plus simple et, incrémentalement, passez à celles qui vous semblent essentielles et, finalement, celles qui semblent plus optionnelles (par exemple **GOL::setFromPattern**).

- Assurez-vous, à chaque petite étape, de valider votre implémentation en utilisant le débogueur en mode **Debug**. Lorsque vous avez terminé une partie importante, assurez-vous de faire une validation exhaustive de tous les cas limites de cette partie. Avant de passer à une autre partie, valider que tout fonctionne bien même en mode **Release**.

7. Il est tout à fait raisonnable de faire une première version nommée **GOL_1_Team##** visant une implémentation plus simple et naïve. L'objectif étant de vous exposer à toutes les parties du projet sans trop mettre d'emphase sur la partie plus complexe d'optimisation. Souvent, on visera des algorithmes simples, mais fonctionnels qui permettent de couvrir entièrement le projet. Par la suite, on fait une deuxième version nommée **GOLTeam##** qui reprend en grande partie le code réalisé auparavant, mais auquel on ajoute les optimisations envisagées. Une telle approche est excellente d'un point de vue académique et vous assure plus facilement de livrer. De plus, pendant que vous faites la première version, vous voyez clairement où se trouvent les goulots d'étranglement et quelles sont les parties à améliorer.
8. Avant la remise, assurez-vous que :
   - La classe **GOL::ImplementationInformation** est bien remplie.
   - Le projet compile sans erreurs (évidemment), mais aussi sans avertissements (« *warnings* ») autant en **Debug** qu'en **Release**.
   - Faire un nettoyage de votre projet avant de le compresser (format ZIP) pour la remise. En bref, vous devez :
     - **ne pas effacer** le dossier :
       - **.../GPA675Lab1GOL/GOLAppLib/x64** de la *librairie*
     - effacer les dossiers :
       - **.../GPA675Lab1GOL/.vs** de la solution
         *attention, il est invisible par défaut sous **Windows***
       - **.../GPA675Lab1GOL/x64** de la solution
       - **.../GPA675Lab1GOL/GPA675Lab1GOL/x64** du projet

Références

Voici plusieurs références pertinentes, utiles et intéressantes :

- Automate cellulaire, Wikipedia (français et anglais)
- Cellular Automaton, Wolfram MathWorld
- LifeWiki
- Golly

Évaluation

Vous trouverez sur Moodle la grille générale de correction. Toutefois, quelques points particuliers sont à considérer :

- plusieurs pénalités habituelles peuvent s'appliquer :
  - remise en retard
  - remise problématique ou dysfonctionnelle
  - présence de « *warnings* » lors de la compilation ou de l'édition de lien (« *linker* »)
  - qualité du français
- une pénalité supplémentaire s'applique aussi :
  - si votre application plante dans certaines situations, vous aurez des pénalités
  - cette pénalité est très forte, alors il vaut mieux un code robuste qui ne fait pas tout ce qui est demandé plutôt que l'inverse
  - évidemment, il y a un minimum à livrer : minimalement un pas de simulation avec la visualisation.
- 20% de la note est attribuée à la performance de votre application. La performance de votre implémentation sera comparée aux trois implémentations données et votre note sera linéairement attribuée selon cette grille :

Comparaison de votre implémentation	**Pointage sur 20%**
< impl. 1 ou non fonctionnel	0
>= impl. 1 et < impl. 2	linéaire entre 0% et 10%
>= impl. 2 et < impl. 3	linéaire entre 10% et 20%
>= impl. 3	20% + maximum de 10% points

- Vous trouverez, dans la section *Statistics* de l'application, la métrique *elapsed time* qui indique le temps qui s'écoule à chaque pas de simulation. Cette métrique évalue exclusivement le temps que prend les fonctions **`GOL::processOneStep`** et **`GOL::updateImage`**. Autrement dit, cette métrique ne tient pas compte de l'affichage à l'écran, de la gestion des signaux ou tout autre traitement. C'est cette métrique qui est utilisée pour l'attribution des points. De plus, vous devez savoir que, lors de l'évaluation, l'ordinateur est mis dans un état optimal pour avoir des comparaisons fiables et répétables. De plus, l'attribution des points est réalisée avec la configuration **`Release`** seulement.
- Il est important de comprendre que l'objectif des optimisations attendues est lié à l'utilisation intensive de pointeurs pour se déplacer en mémoire. Il existe des algorithmes spectaculairement performants comme *Hashlife* qui permettent des performances incomparables avec les méthodes dites plus traditionnelles. L'usage d'une telle approche est interdit.
- Sachez aussi que les implémentations auxquelles vous êtes comparées utilisent ce qui est attendu dans le cours. Vous ne vous comparez donc pas à des implémentations de type *Hashlife*, ou utilisant du parallélisme (*multithreading* ou *GPGPU*) ou des instructions de bas niveau qui ne sont pas couvertes dans le cours.