USRS46 – TP

Implémentation du jeu de la vie en C#

Sommaire

Présentation

Le jeu de la vie est un jeu de simulation qui permet d'observer l'évolution de cellules sur une grille. A chaque pas de temps, certaines cellules apparaissent et d'autres disparaissent en fonction de règles simples. Il est alors possible de voir émerger des comportements typiques plus ou moins chaotiques ou encore des convergences vers des états stables.

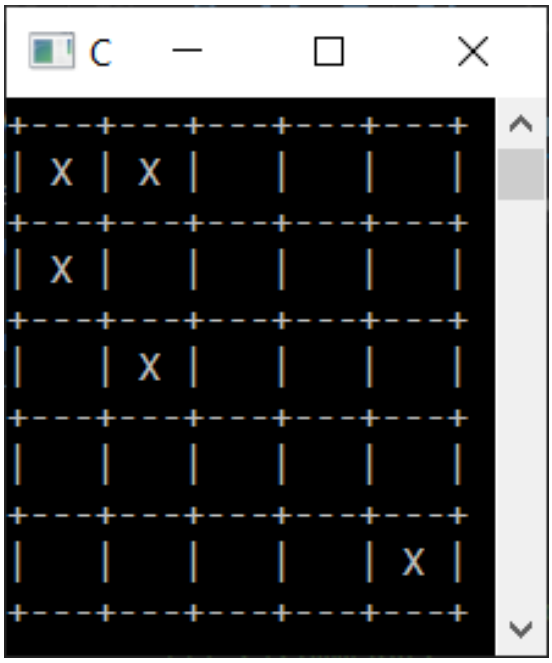

Figure 1 - Représentation de la grille initiale du jeu de la vie en version textuelle

Règles de simulation

Cette simulation ne repose que sur deux règles :

- Si au temps `T`, à une emplacement donné `E` il n'y a aucune cellule mais que sur les emplacements voisins on compte exactement 3 cellules alors au temps `T+1` une cellule apparaît à cet emplacement `E`.
- Si au temps T à un emplacement donné `E`, il y a une cellule et qu'autour d'elle il y a exactement 2 ou 3 cellules vivantes alors la cellule de l'emplacement `E` reste en place au temps `T+1`, sinon elle disparaît.

Tout se joue sur les conditions initiales, c'est-à-dire **le nombre de cellules** et **leurs positions** au lancement de la simulation. Il est en effet possible de voir apparaître des comportements très intéressants en changeant simplement les conditions de départs. C'est un problème qui n'est pas compliqué en soit mais pour autant qui peut être qualifié de **complexe**.

Ce travail va vous permettre d'approfondir vos connaissances en programmation orientée objet ainsi que vous permettre de manipuler l'environnement graphique proposé par le langage C# au travers des Windows Form.

Implémentation textuelle

Dans une premier temps, il est important d'avoir le réflexe de développer vos applications dans des environnements textuels, en console. Une fois que votre code sera propre et fonctionnel, il sera alors possible d'ajouter la couche graphique. Les étapes de développement sont importantes pour assurer le maximum d'indépendance entre le code source qui s'occupe de la mécanique du jeu et le code permettant de gérer l'affichage graphique. Prenez l'habitude de bien séparer chaque composante de votre projet.

En savoir plus sur le modèle MVC

Prenez le réflexe de créer un nouveau dépôt git pour chaque nouveau projet. Vous pouvez travailler à plusieurs si vous le souhaitez et vous entraîner ainsi à utiliser git sur un projet de groupe

Partie 1 - Environnement textuel

Si vous ressentez le besoin d'être guidé dans votre travail, vous trouverez dans cette partie des indications sur les classes à implémenter, les attributs et les méthodes ainsi que les interactions entre les différentes classes. Le but de cette UE est de vous rendre capable de réaliser ce type de travail sans indication. Si vous ne ressentez pas le besoin d'être guidé, vous pouvez simplement consulter le diagramme de classes et commencer votre implémentation.

Dans tous les cas, prenez bien le temps d'observer et de comprendre le diagramme de classes (Fig. 2) pour avoir en tête l'organisation des classes les unes par rapport aux autres et le rôle de chacune d'elle.

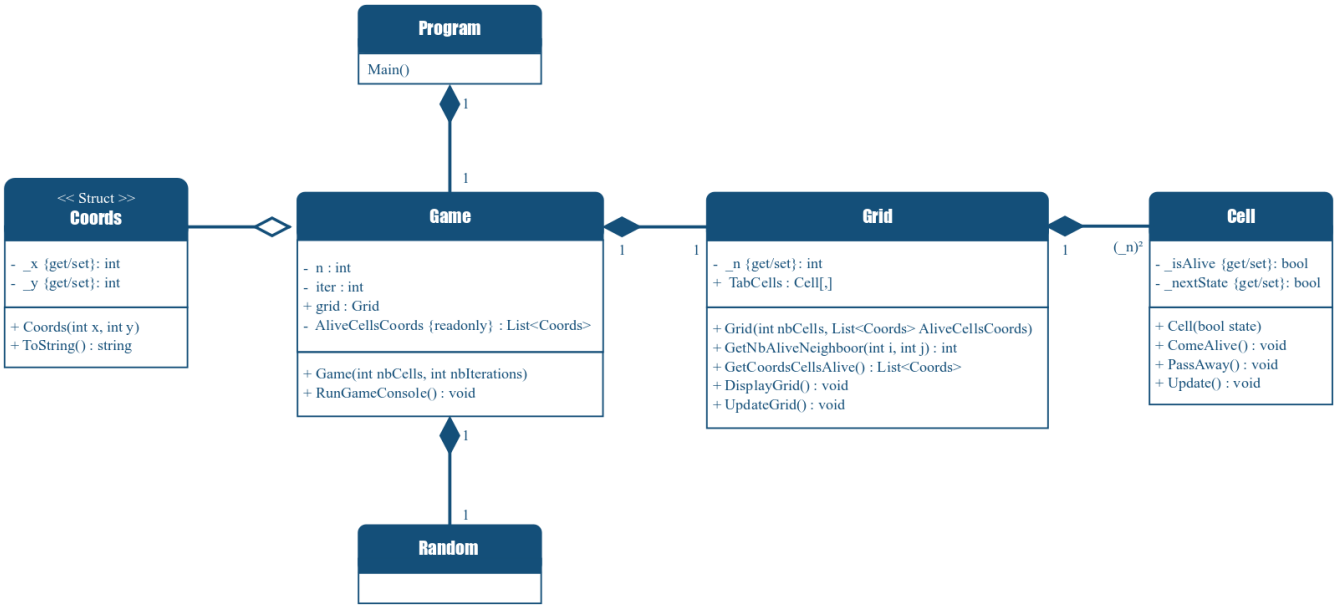

Figure 2 - Représentation UML du projet Jeu de la vie en environnement textuel

Un système d'étoile pour chaque méthode permet de vous informer sur la difficulté de leur implémentation. Une étoile correspond à une méthode très simple ou entièrement guidée. Trois étoiles correspondent à une méthode peu guidée ou demandant de mettre au point un algorithme et de réfléchir aux différents cas d'application (autrement dit il vous faudra plus de temps pour l'implémenter et c'est normal).

Entraînez-vous à utiliser git en effectuant un commit commenté pour chaque nouvelle méthode fonctionnelle.

Ouvrez l'IDE Visual Studio et créer un nouveau projet. Vérifier qu'il est pour le moment configurer pour une sortie de type console en modifiant le fichier de configuration : `Projet > Propriétés de NomDeVotreProjet... > Application > Type de sortie > Application Console`. Sauvegarder vos changement.

Vous pouvez si vous en ressentez le besoin récupérer les squelettes des classes suivantes avant de démarrer vos implémentaitons, les attributs y sont déjà déclarés ainsi que les signatures des méthodes attendues.

La classe Cell

```
using System;

public class Cell
{
    // Attribut privé représentant l'état actuel de la cellule
    private bool _isAlive;

    // Accesseur pour lire/modifier _isAlive
    public bool isAlive { get { return _isAlive; } set { _isAlive = value; } }

    // Attribut privé pour stocker l'état futur de la cellule
    private bool _nextState;

    // Accesseur pour lire/modifier _nextState
    public bool nextState { get { return _nextState; } set { _nextState = value; }
```

```
}

    // ★☆☆ Constructeur : initialise la cellule avec un état donné
    public Cell(bool state) {
    }

    // ★☆☆ Méthode : fait vivre la cellule au prochain pas de simulation
    public void ComeAlive() {
    }

    // ★☆☆ Méthode : tue la cellule au prochain pas de simulation
    public void PassAway() {
    }

    // ★☆☆ Méthode : met à jour _isAlive avec _nextState
    public void Update() {
    }
}
```

Remarque : L'attribut _isAlive est un attribut privé, pour y accéder on définit ici deux moyens, un accès en lecture grâce au mot clé get et un accès en écriture grâce au mot clé set.

Rappel sur les accesseurs - Partie III-C à lire

[Point de vigilance] Assurez-vous de bien comprendre la différence entre `_isAlive` (attribut de classe) et `isAlive` (accesseur qui permet de lire ou modifier l'attribut qui lui est associé)

La classe Grid

```
using System;
using System.Collections.Generic; // Pour List<T>

public class Grid
{
    // Taille de la grille
    private int _n;
    public int n { get { return _n; } set { _n = value; } }

    // Tableau 2D de cellules
    public Cell[,] TabCells;

    // ★★★ Constructeur : initialise la grille et les cellules
    public Grid(int nbCells, List<Coords> AliveCellsCoords)
    {
        // TODO : initialiser n et TabCells
        // TODO : remplissage du tableau avec à chaque emplacement une instance
d'une cellule Cell créée vivante (true) si les coordonnées sont dans la liste
AliveCellsCoords ou absente (false) sinon.
    }

    // ★★☆ Méthode : retourne le nombre de cellules vivantes autour d'une
```

```
cellule
    public int getNbAliveNeighboor(int i, int j) {
        return 0;
    }

    // ★★☆ Méthode : retourne les coordonnées valides autour d'une cellule
    public List<Coords> getCoordsCellsAlive() {
        return new List<Coords>();
    }

    // ★★☆ Méthode : afficher la grille en console (X pour cellule vivante)
    public void DisplayGrid() {
    }

    // ★★★ Méthode : mettre à jour la grille selon les règles du jeu
    public void UpdateGrid()
    {
        // Méthode qui parcourt chaque cellule et qui met à jour leur attribut
_nextStep, via son accesseur en écriture, en fonction des règles de la simulation.
L'attribut est mis à true si la cellule reste en vie ou apparaît et à false si la
cellule à cet emplacement disparaît ou reste absente. Une fois toute la grille
parcourue, une deuxième passe est effectué pour associer la valeur de nexStep à
l'attribut isAlive de chaque cellule.

        // TODO : première passe : calculer nextState pour chaque cellule
        // TODO : deuxième passe : appliquer nextState à _isAlive
    }
}
```

Rappel sur les tableaux

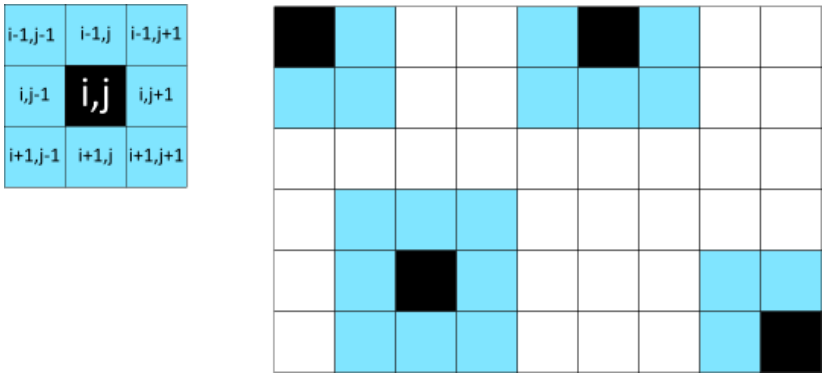

Figure 3 – Rappel des coordonnées des cellules voisines pour une cellule de coordonnées (i,j) et représentation des cellules voisines (bleues) pour les cellules identifiées en noir.

La classe Game

```
using System;
using System.Collections.Generic; // Pour List<T>
using System.Threading;           // Pour Thread.Sleep

public class Game
```

```
{
    // Taille de la grille
    private int n;

    // Nombre d'itérations de la simulation
    private int iter;

    // Grille contenant toutes les cellules
    public Grid grid;

    // Liste des coordonnées des cellules vivantes au départ
    public List<Coords> AliveCellsCoords;

    // ★★★ Constructeur : initialise la simulation
    public Game(int nbCells, int nbIterations)
    {
        // TODO : initialiser n et iter
        // TODO : initialiser AliveCellsCoords avec une configuration initiale
        // TODO : créer une nouvelle grille Grid(n, AliveCellsCoords)
        // (quelques exemples de configuration de départ sont fournis en fin de
sujet)
    }

    // ★★★ Méthode : exécute la simulation dans la console
    public void RunGameConsole()
    {
        // TODO : afficher la grille initiale avec grid.DisplayGrid()

        // Boucle sur le nombre d'itérations
        for (int i = 0; i < iter; i++)
        {
            // TODO : mettre à jour la grille avec grid.UpdateGrid()
            // TODO : afficher la grille après mise à jour avec grid.DisplayGrid()
            // TODO : mettre en pause 1 seconde avec Thread.Sleep(1000)
        }
    }
}
```

La classe Program

```
using System;

namespace JeuDeLaVie
{
    static class Program
    {
        static void Main()
        {
            Game game = new Game(4, 10);
            game.RunGameConsole();
```

```
            }
        }
    }
```

Partie 2 - Environnement graphique

Nous allons maintenant mettre en place un environnement graphique pour pouvoir observer l'évolution de notre simulation dans une fenêtre (voir Fig. 4). Les étapes suivantes ne sont pas entièrement détaillées et vous renvoies aux manipulations faites dans le TD précédent. N'hésitez pas à rechercher des ressources complémentaires pour aller plus loin.

Figure 4 – Représentation graphique attendue

L'UML pour la partie graphique du TP est le suivant :

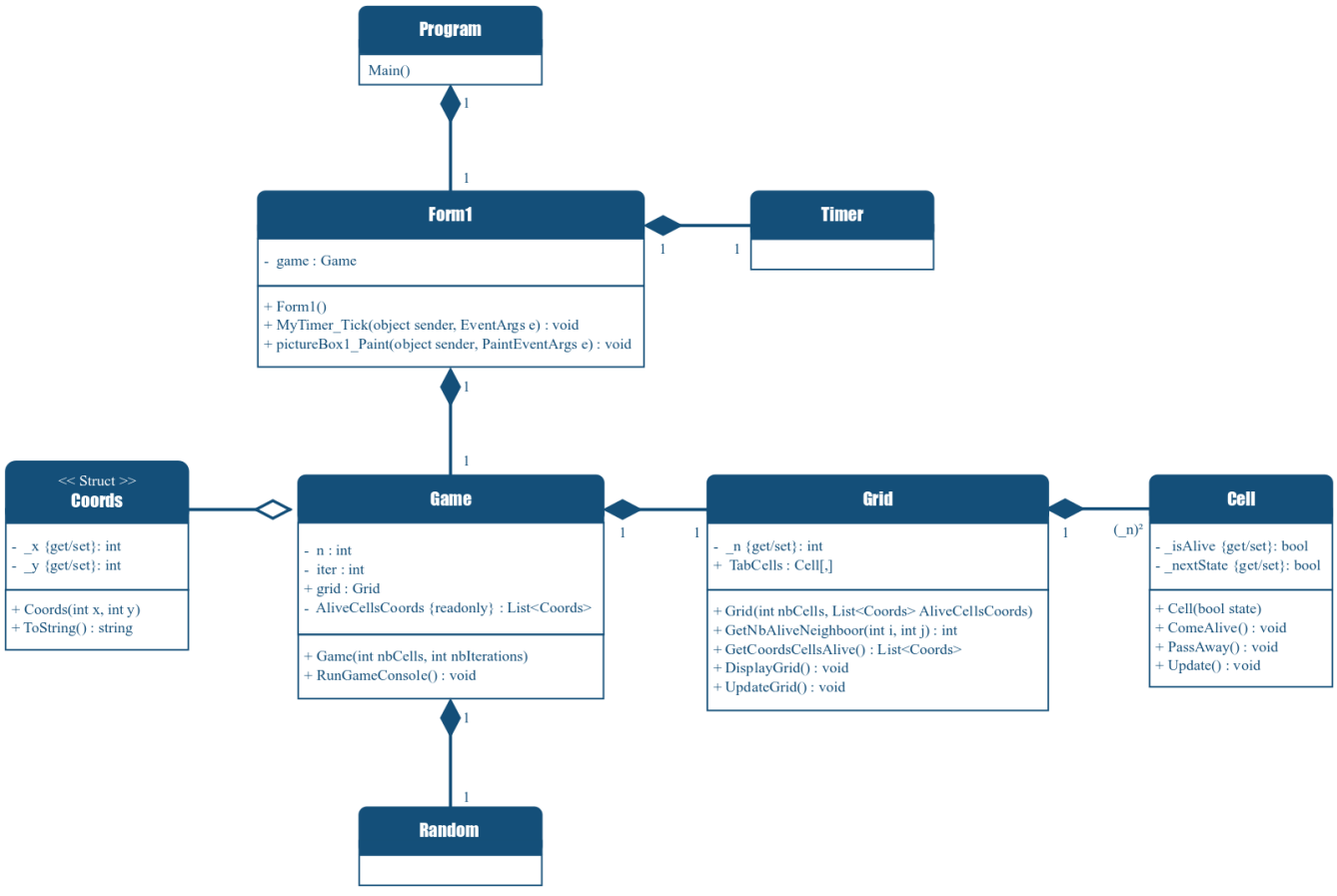

Figure 5 – Représentation UML du projet Jeu de la vie en environnement graphique

Implémentation graphique

1. **Créer un nouveau projet Windows Forms**
    - Choisissez le modèle **Application Windows Forms**. Cela vous permettra de manipuler directement la fenêtre graphique.
2. **Ouvrir le fichier Form1.cs**
    - Vous y trouverez une fenêtre graphique vide prête à être personnalisée.
3. **Afficher la boîte à outils**
    - Si elle n'apparaît pas à gauche, allez dans **Affichage > Boîte à outils**.
    - Faites de même pour tous les outils dont vous aurez besoin.
4. **Créer des classes personnalisées pour PictureBox et Label**
    - Pour les besoins de base de cet exercice vous allez manipuler uniquement une `Picture-Box` et un `Label`. De la même façon que pour le TD précédent, définissez un nouveau dossier Controls dans lequel vous allez écrire deux classes pour définir les propriétés de nouveaux objets qui héritent pour l'un de PictureBox et pour l'autre de Label.

      Classe héritant de PictureBox

        - Le constructeur prend un entier `n` en paramètre.
        - La couleur de fond est modifiée pour une couleur foncée.
        - La taille est un carré dont chaque côté vaut `5 * n`.

      Classe héritant de Label

        - Le constructeur ne prend aucun paramètre.
        - La bordure est visible.
        - Le texte est initialement vide et centré verticalement et horizontalement.
        - La taille est de 100 × 23 pixels.
5. **Déclarer et initialiser les attributs dans Form1**
    - Ajoutez un attribut pour chaque classe (PictureBox et Label) et initialisez-les dans le constructeur.
6. **Positionner les éléments**
    - Centrez les éléments dans la fenêtre graphique, l'un au-dessus de l'autre.
7. **Définir un attribut `n` dans Form1**
    - Initialisez-le à 40 dans le constructeur.
    - Il servira pour la taille de la PictureBox.
8. **Configurer Form1_Load()**
    - Définissez la taille et le titre de la fenêtre dans cette méthode, qui est générée automatiquement lors d'un double-clic sur la fenêtre en mode Design.

9. **Importer les classes de la partie 1**
    - Assurez-vous que vos classes `Cell` et `Grid` soient accessibles.
10. **Récupérer le code source de Program.cs**

```
using System.Windows.Forms;

namespace JeuDeLaVie {
    static class Program {
        static void Main() {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());;
        }
    }
}
```

11. **Ajouter un attribut de type `Game` dans Form1**
    - Initialisez-le dans le constructeur pour gérer la logique du jeu.
12. **Configurer un `Timer`**
    - Intervalle : 200 ms.
    - Associer la méthode `UpdateGrid` à l'événement Tick du Timer.
13. **Ajouter un compteur de générations**
    - Déclarez un entier `generation` initialisé à 0 dans le constructeur.
    - Incrémentez-le à chaque tick du `Timer`.
14. **Définir la méthode UpdateGrid**

```
private void UpdateGrid(object sender, EventArgs e)
{
    // Mettre à jour la grille du jeu
    // Incrémenter la variable 'generation'
    // Mettre à jour le Label avec la valeur de 'generation'
    // Rafraîchir la fenêtre graphique
}
```

15. **Ajouter un PaintEventHandler à la PictureBox**

```
pctBox_main.Paint += new PaintEventHandler(pctBox_main_Paint);
```

16. **Définir la méthode pctBox_main_Paint**

```
private void pctBox_main_Paint(object sender, PaintEventArgs e)
{
    // Définir une brush blanche
    // Boucler sur toutes les cellules de la grille
    // Si la cellule est vivante :
    //     Dessiner un rectangle plein de 5x5 pixels à la position
correspondante
}
```

Améliorations

Une fois que vous aurez pu admirer vos simulations tourner dans une fenêtre graphique, entraînez-vous en implémentant les fonctionnalités suivantes. Aucune consigne ne vous est donnée pour vous guider, c'est un travail en toute autonomie.

1. Ajouter une fin de simulation quand la grille n'évolue plus.
2. Ajouter une génération aléatoire du nombre de cellules initiales et de leurs coordonnées.
3. Améliorer l'interface en ajoutant la possibilité de faire une pause ou de recommencer une simulation au début

Annexe - Exemples de configuration de départ

```
// Disparition de toutes les cellules en quelques iterations sur une grille 4,4
AliveCellsCoords = new List<Coords> {
    new Coords(2, 2),
    new Coords(2, 3),
    new Coords(0, 0),
    new Coords(1,1),
    new Coords(1,2)
};

// Oscillateur de periode 2 sur une grille 4,4
AliveCellsCoords = new List<Coords> {
    new Coords(1, 1),
    new Coords(1, 2),
    new Coords(2, 1),
    new Coords(2,2),
    new Coords(2,0),
    new Coords(1,3)
};


// Emergence d'un cercle sur une grille 5,5
AliveCellsCoords = new List<Coords> {
    new Coords(n/2, n/2),
    new Coords(n/2+1, n/2),
    new Coords(n/2, n/2+1),
    new Coords(1,1)
};

// Evolution d'une croix sur une grille 11,11
AliveCellsCoords = new List<Coords> {
    new Coords(5, 5),
    new Coords(5, 4),
    new Coords(5, 6),
    new Coords(4, 5),
    new Coords(6, 5),
};


// Motif du planeur qui se deplace sur une grille 11,11
AliveCellsCoords = new List<Coords> {
    new Coords(5, 4),
    new Coords(4, 4),
    new Coords(4, 5),
    new Coords(3, 5),
    new Coords(3, 3),
};
```