Alban Delespaul et Bruno Monasson

Projet - Jeu de la vie

Objectif du projet :

Le Jeu de la vie est un jeu de simulation inventé par John Horton Conway en 1970. Les règles de ce jeu sont simples : une grille dans laquelle chaque case représente une cellule, soit morte soit vivante, qui évolue en fonction du nombre de cellules voisines vivantes. Par exemple, si une cellule morte possède exactement trois voisines vivantes elle évoluera au tour d'après, et deviendra vivante. Au contraire, une cellule vivante n'ayant ni deux voisines vivantes ni trois voisines vivantes meurt au tour suivant. Le Jeu de la vie est donc un jeu dans lequel l'utilisateur n'intervient pas en cours de route car le déroulement de la partie suit des règles bien définies. L'objectif de notre projet est donc de mettre au point une interface avec laquelle l'utilisateur peut suivre l'évolution de la partie tout en modifiant les paramètres initiaux pour explorer les différentes combinaisons.

Description de l'algorithme et de ses fonctions :

Classe : **Cellule**

Tout d'abord, nous avons décidé de créer une classe Cellule qui représente un des cases de la grille. Chaque cellule possède trois attributs, sa position, son statut (si la cellule est vivante ou morte) et son nombre de voisins vivants, et deux méthodes, statut() et voisins().

Attributs :

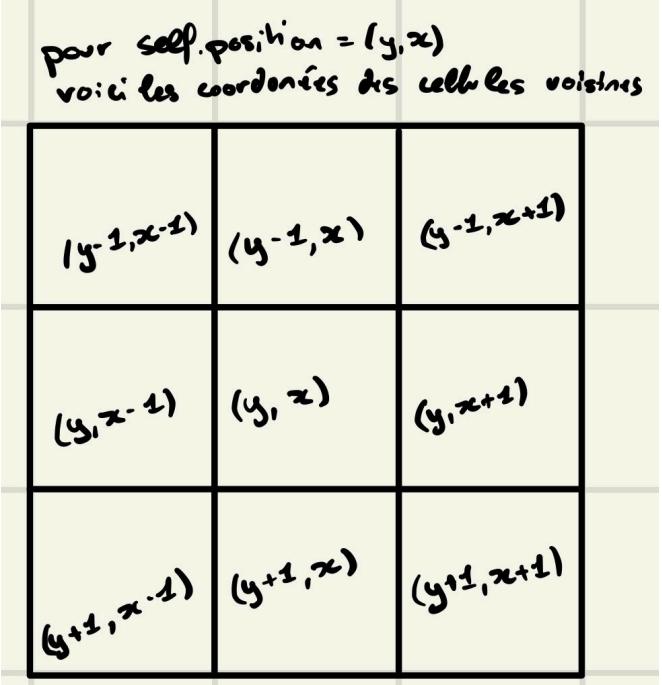
- **self.position** est un tuple de deux valeurs qui indique la position de cette cellule dans la grille. La grille étant représentée par une liste de liste, la première valeur du tuple indique la ligne de la cellule, ce qui équivaut à l'indice de la sous liste, et le deuxième élément renvoie à la colonne de la cellule, l'indice de cette cellule dans la sous-liste. Par exemple, pour une cellule c ayant des coordonnées telles que c.positions = (a, b) alors GRILLE[a][b] = c . *(a et b sont bien sur des entiers positifs)*
- **self.en_vie** est un booléen qui vaut soit TRUE si la cellule est vivante, soit FALSE si cette cellule est morte. Le statut est ensuite mis à jour à chaque étape grâce à la méthode statut() comme nous allons le voir.
- **self.nbvoisins** est une variable, dont la valeur est toujours un entier positif, qui comptabilise le nombre de voisins vivants que possède une cellule, valeur qui est calculée par la méthode voisins().

Méthodes :

- **self.statut()** est la méthode qui modifie le statut d'une cellule selon les règles du jeu. En effet, si la cellule est vivante mais n'a pas deux ou trois voisins vivants elle meurt (self.en_vie = False) alors que si elle est morte mais qu'elle possède trois voisins

vivants elle naît (self.en_vie = True). Dans tous les autres cas la cellule ne change pas de statut et la valeur de self.en_vie n'est donc pas modifiée.

- **self.voisins()** est la méthode qui calcule le nombre de voisins vivants d'une cellule. Pour cela on parcourt les huit voisins de la cellule et on incrémente de 1 la valeur de self.nbvoisins à chaque fois qu'une de ces cellules est vivante. Pour parcourir les voisins il suffit d'utiliser self.position comme nous pouvons le voir sur le schéma suivant :



Fonctions :

- **affichage_grille()** est la fonction qui affiche une grille de TAILLE x TAILLE cases. Cette fonction est notamment utilisée au tout début du programme pour afficher la grille sur la fenêtre ouverte. De plus, il est important de réafficher la grille à la fin de chaque étape car sinon elle disparaît sous les cases blanches
- **creation.grille()** est la fonction qui crée la variable grille qui est une liste de liste. Ce tableau représente donc la grille de jeu et chaque case du tableau est une des cellules. Il est aussi important de noter que la grille présente à l'écran n'est pas une transposition directe de la variable grille car une marge, non visible à l'écran, est présente dans le tableau pour remédier à des problèmes d'indices comme nous allons l'expliquer ensuite. C'est pour cette raison qu'on ne parcourt pas l'entièreté de la grille dans la fonction etape(), ou qu'on enlève 1 aux indices de la case qu'on veut colorier dans la fonction coloriage().
- **mise_a_jour_marges()** est la fonction utilisée pour s'assurer que les marges continuent de jouer leur rôle. En effet, nous avons créé cette marge pour ne pas avoir d'erreur d'indice dans la méthode voisins() de la classe Cellule, en copiant la valeur des cellules des bordures dans la marge. Toutefois si le statut d'une cellule qui se situe dans un coin est modifié sa copie, qui se situe dans la marge, ne va pas être automatiquement modifiée et c'est pour cela qu'il faut le faire manuellement à la fin de chaque étape.
- **coloriage()** est la fonction qui crée un carré de la taille de la case aux coordonnées de la case pour représenter le statut de la cellule (carré noir si elle est en vie ou carré

blanc si elle est morte). Les nouveaux carrés superposent ce qui a déjà été dessiné, ce qui pose problème car la grille disparaît sous les nouvelles cases. C'est pour cela qu'on appelle la fonction affichage_grille() à la fin de chaque étape.

- **etape()** est essentielle au fonctionnement de notre projet car c'est elle qui simule le déroulement d'une étape. Tout d'abord, on calcule le nombre de voisins de chaque cellule en parcourant la grille de jeu (sans les marges) et en utilisant la méthode voisins() pour chaque cellule. Ensuite on parcourt encore une fois la grille de jeu pour calculer et modifier le statut de chaque cellule grâce à la méthode statut() de la classe Cellule. Les cellules qui changent de statut changent par la même occasion de couleur car la fonction coloriage() est appelée à l'intérieur de statut(). Enfin, on met à jour les marges et on réaffiche la grille.
- **initialisation_case()** est une fonction très importante. En effet, elle permet à l'utilisateur de choisir les cellules qui seront vivantes au début de la partie ou alors elles seront choisies aléatoirement. Elle commence par une boucle while puis récupère l'endroit de clic de l'utilisateur. Ensuite, la fonction regarde si le clic apparaît sur la grille. Il va donc, dans un second temps, diviser les coordonnées par la longueur d'une case pour voir dans quelle cellule l'utilisateur a cliqué. Nous avons aussi fait en sorte que nous pouvons faire naître une cellule morte ainsi que d'en tuer une vivante, en cas de mauvais clic. Après cela, une fonction parcourt la grille entière pour regarder s'il y a des cellules vivantes. Ensuite, si il n'y en a pas, la fonctione reparcourt toute la grille et met en vie, avec une chance sur trois, les cases. Ce qui a comme conséquence la création d'une grille de départ aléatoire. On finit par rafraîchir l'affichage.

<u>Principales difficultées rencontrées :</u>

La première difficulté que nous avons rencontrée est que le Jeu de la vie se joue sur une grille à deux dimensions qui est théoriquement infinie. Il est donc impossible de simuler une partie du Jeu de la vie de manière complètement exhaustive, sur une grille d'une taille donnée, car certaines formations de cellules extérieures à la grille visible sur l'écran pourraient influencer le comportement des cellules présentes dans la grille visible sur l'écran.

Pour remédier à ce problème, nous avons décidé de simuler cette notion de grille infinie en utilisant la forme géométrique d'un tore. En effet, ce solide possède une surface définie mais dans laquelle on peut se déplacer dans n'importe quelle direction sur une distance infinie sans jamais buter sur une arête (bord de la grille). A l'image d'un cylindre dont le patron est un rectangle, avec deux cercles sur lesquels on enroule ce rectangle, la représentation 2D du tore est tout simplement un rectangle qui s'enroule sur lui-même soit horizontalement que verticalement. De manière concrète, pour en revenir à notre projet, cela signifie qu'une cellule qui se situe sur la ligne du bas est voisine avec une cellule qui elle se trouve sur la ligne du haut (uniquement si elles sont aussi voisines horizontalement bien sur). Cela signifie que des formation qui sortent de la grille apparaissent à l'opposé et donc

que c'est formations ne sont pas "perdues", contrairement à ce qui se serait passé sur une grille avec des dimensions bien définies.

Pour implémenter cela nous aurions pu utiliser modulo mais nous avons préféré mettre en place un système de marge qui n'était pas si difficile à créer ou à gérer et bien plus facile à se visualiser. L'idée était de rajouter une colonne de cases à droite et à gauche, ainsi qu'une ligne en haut et en bas, dans lesquelles les cellules correspondantes seraient copiées. Il nous a donc suffit de créer un tableau plus grand, une fonction qui mettait à jour les marges et de décaler quelques indices de 1.

Une autre difficulté rencontrée est la complexité de redemander tant que l'utilisateur n'entre pas une valeur correcte. En effet, nous avions commencé avec uniquement un assert, qui présentait le désavantage de créer une erreur d'assertion. Ensuite, à force de faire des fausses touches qui nous obligeaient à relancer le programme à chaque fois, nous avons décidé de changer cela. Nous avions commencé par reprendre quelque chose fait dans les projets d'avant: la boucle *while*. Cette boucle demande à l'utilisateur de rentrer un nombre et que, tant que le nombre n'était pas positif, l'ordinateur redemandait. Seul problème étant que, ça n'est tout simplement pas possible. Cela n'est pas possible pour une simple raison: pour que l'utilisateur rentre un nombre, il faut la formulation *int(input(* devant. Le problème étant que, avec cette formulation, si l'utilisateur fait une fausse touche et tape une lettre, l'ordinateur nous renverrait une erreur. En effet, l'ordinateur demande un entier et nous lui renvoyons une chaîne de caractères! Nous avons donc ensuite essayé de le faire en deux étapes. La première serait de demander à l'utilisateur une valeur mais, sans mettre *int(input*(, mais avec seulement *input(*. Nous associons ce résultat à la variable *TAILLE*. Ensuite, nous avons créé une boucle *while*. Cette dernière ne s'arrêterait que si *TAILLE* devenait un entier. Avec cette idée là, l'ordinateur changerait *TAILLE* en entier, si un entier avait été choisi. Mais non! Puisque nous n'avions que fait *input(*, l'ordinateur associait cela à une chaîne de caractère. Pour que le code marchait, il fallait donc que l'utilisateur écrive la valeur deux fois; une fois pour le premier *input(*, et une seconde fois pour le *int(input(*. Ceci était assez dérangeant et aucune solution ne se présentait à nous. Après de longues recherches, nous ne trouvions toujours pas. Une idée nous vient donc, la méthode *try*. Cette solution de miracle a finalement été celle qui a été choisie car c'était le seule qui faisait que l'on attendait.

<u>Notions apprises durant ce projet :</u>

Comment utiliser la librairie pygame:

Avant de commencer ce projet, nous ne connaissions même pas cette librairie. Nous avons donc appris comment l'utiliser au fil du temps. Nous avons notamment utilisé les rectangles (pour colorier les cases mais aussi pour les zones de textes), les lignes (pour dessiner la grille) et la fonction mouse (pour avoir la position des clicks notamment)

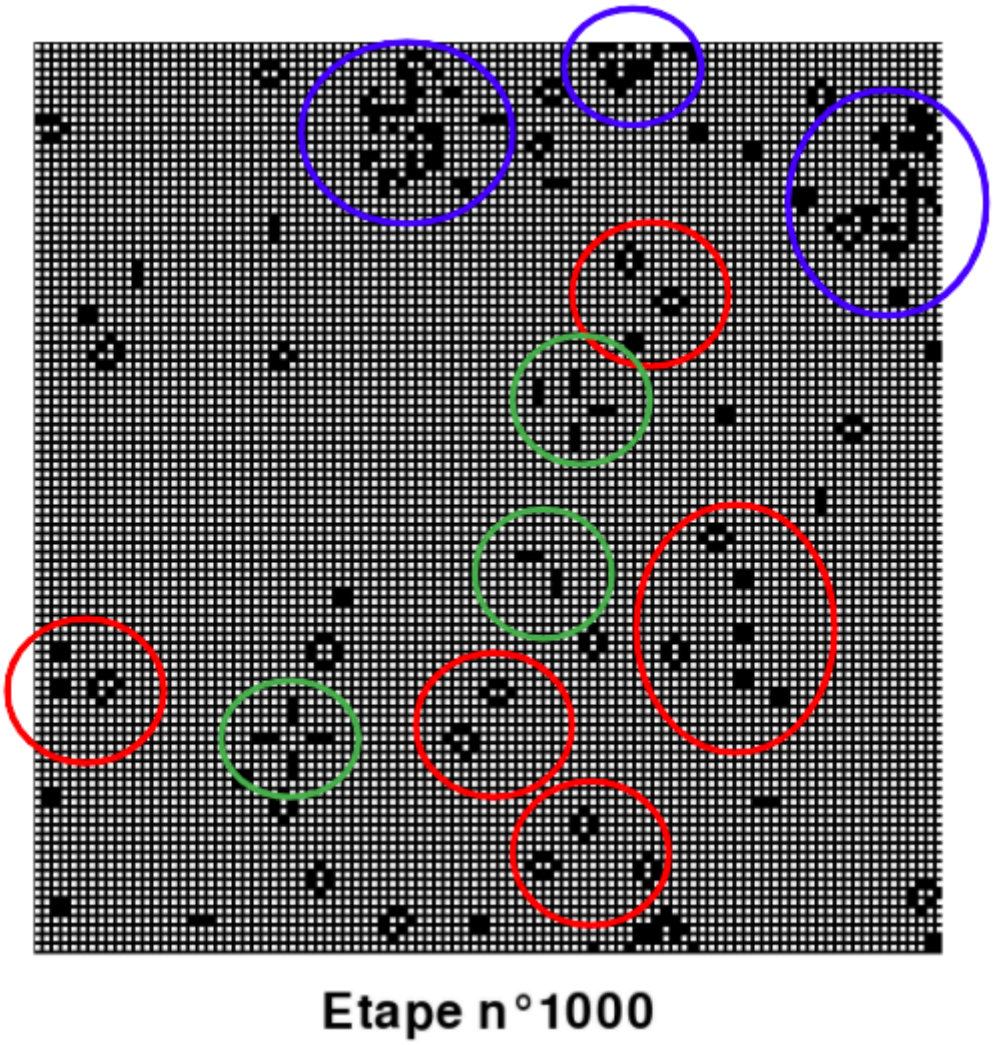
La fonction *try*:

C'était une fonction que nous ne connaissions pas jusque là. En effet, nous ne l'avions pas vraiment utilisé en cours pendant ces dernières années. Nous savons donc désormais le fonctionnement de *try*, *except*, et *else* (lorsqu'il suit un *try*). Pour l'utiliser, nous

avons, en premier lieu, fait un *input(*. Celui-ci n'est pas un *int(input(* car, si l'utilisateur mettrais une lettre ou un symbole, l'ordinateur renverrais une erreur. Nous avons donc, ensuite, créé une boucle *while*. Cette boucle se termine à une seule solution, lorsqu'essai est *False*. Ce dernier avait été initié précédemment et donné la valeur *True*. La seule manière qu'essai devienne *False* est de remplir toutes les conditions: ce qui est entré doit être un nombre positif plus grand que 0. Ensuite, dans la boucle, nous utilisons *try* pour essayer de transformer l'*input* de l'utilisateur en *int*. Si cette condition n'est pas remplie, l'ordinateur redemande à l'utilisateur une valeur. Si cette condition est remplie, l'ordinateur va donc comparer cette valeur à 0. Si elle est strictement plus grande, essai devient False et la boucle se finit donc. Autrement, l'utilisateur doit entrer une nouvelle valeur.

Finalement nous avons découvert plus en détail le fonctionnement du jeu de la vie, dont nous connaissions la notion mais pas vraiment son déroulement. Au contraire, une fois ce projet fini, nous avons pu tester le développement de nombreuses grilles initiales, notamment grâce à la configuration aléatoire, sur des grilles avec un grand nombre de cases et sur un grand nombre d'étapes aussi. Nous nous sommes rendu compte que, contrairement à ce que nous pensions originellement, "la vie" ne disparaît jamais totalement car de nombreuses petites structures formées substituent peu importe le nombre d'étapes (du moment qu'aucun élément extérieur ne les dérange), ces structures étant soit stables (entourées de rouge dans la grille ci-dessous), soit oscillantes (entourées de vert ci-dessous).

Nous avons aussi pu comprendre, dans les grandes lignes, le déroulement d'une partie (sur une grande grille et avec la configuration aléatoire) qui se découpe en plusieurs étapes. Tout d'abord, la "population" de la grille va chuter très rapidement d'environ ⅓ des cellules étant vivantes à moins de des cellules étant vivantes après une cinquantaine d'étapes. Ces cellules se regroupent en structures (une quinzaine environ au début) qui ressemblent à des explosions (entourées de bleu ci-dessous), les débris de ces explosions entraînant d'autres explosions etc. Il est aussi très intéressant qu'assez souvent des petits planeurs (structures qui se déplacent en diagonale et qui gardent la même formation périodiquement) sont créés aléatoirement, alors que nous pensions qu'ils étaient très rares et qu'en réalité ils ne pouvaient se former que de manière artificielle. Petit à petit, la taille et le nombre de ces explosions jusqu'à arriver à une configuration stable où toutes les cellules vivantes appartiennent à des structures stables ou oscillantes, et où aucune nouvelle formation ne verra le jour. Le nombre d'étapes nécessaires pour atteindre ce stade semble beaucoup plus aléatoire que celui où le nombre de cellules arrête de chuter par exemple. En effet dans certain cas il ne fallait que 900 étapes alors que dans d'autres pas moins de 4500.



Exemple de grille (100x100 cases) au bout de 1000 étapes avec de formations stables (en rouge), telles que des carrés de côté 2, de formations oscillantes (en vert), telles que des barres de longueur 3, et de formations qui sont en cours “d’explosion” (en bleu)

Ressources utilisées :
https://fr.wikipedia.org/wiki/Jeu_de_la_vie
https://www.w3schools.com/python/python_try_except.asp