Compte Rendu du Projet n°3 :

Le Jeu de la Vie

Leguey Valentin

Guêton Valentin

Leroy Victor

Touraine Cyprien

1G

1. Choix du projet

A) Présentation du thème

Ce programme informatique a été inventé par John Conway, dans les années 70. Ce n'est pas réellement un jeu, mais plutôt un automate cellulaire. En effet, avec des règles très simples, il est possible de faire évoluer des cellules vivantes dans une grille en deux dimensions.

Voici les règles du Jeu de la Vie :

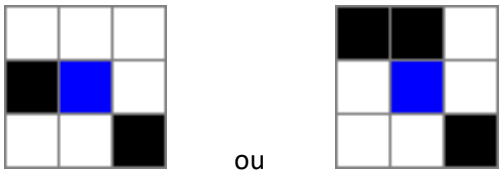
1./ Naissance :

Une cellule peut naître si elle est voisine d'exactement trois cellules vivantes.

Schéma représentant
la naissance d'une cellule.

2./ Survie :

Une cellule survie à la prochaine génération si elle est voisine de deux ou trois cellules vivantes.

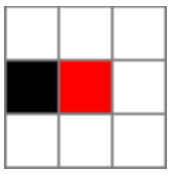


ou

Schémas représentants la survie d'une cellule.

3./ Décès :

Une cellule disparaît en cas d'isolement, si elle est bordée par moins de deux cellules vivantes, ou en cas de surpopulation, si elle compte plus de trois voisines vivantes autour d'elle.



ou

Mort d'isolement Mort de surpopulation

B) Raison de ce choix

Nous avons choisis de prendre ce thème parce qu'il reprenait de nombreuses notions vues en cours. De plus, ce choix nous offrait de bonnes perspectives de bonus comme la gestion de fichiers, ou encore l'interface graphique. Enfin, la complexité du jeu a été un point fort. Des règles très simples en apparence génèrent un tissu de complexité imprévisible, c'est ce qui nous a séduits.

C) Fonctionnalités

Dans cette partie nous allons énumérer les fonctionnalités de notre programme.

- Interaction avec la grille : clic gauche sur une cellule pour la définir vivante, clic droit pour la définir morte.
- Génération aléatoire de cellules vivantes dans la grille (avec un pourcentage définit par l'utilisateur).
- Importation d'un fichier texte comme grille (dans le menu déroulant ou directement en le cherchant dans les dossiers).
- Projet fourni avec une base de 56 structures préfabriquées (dans le dossier */prefabs*).
- Possibilité de sauvegarder la grille, dans l'endroit de votre choix (dirigé vers */saves* de base).
- Nettoyage de la grille à l'aide d'une icône.
- Nettoyage de la grille et arrêt de la simulation si la grille est complètement vide (bordures invisibles comprises)
- Vous pouvez lancer la simulation et la mettre en pause à tout moment, puis la faire repartir.
- Passer à la génération suivante à l'aide d'une icône.
- Nombre de générations affichées dans le coin supérieur gauche.
- Gestion de la vitesse d'affichage entre deux générations.
- Quitter le programme grâce à un bouton prévu à cet effet.
- Jeu de test fonctionnel avec affichage inspiré du premier projet.

2. Organisation du projet

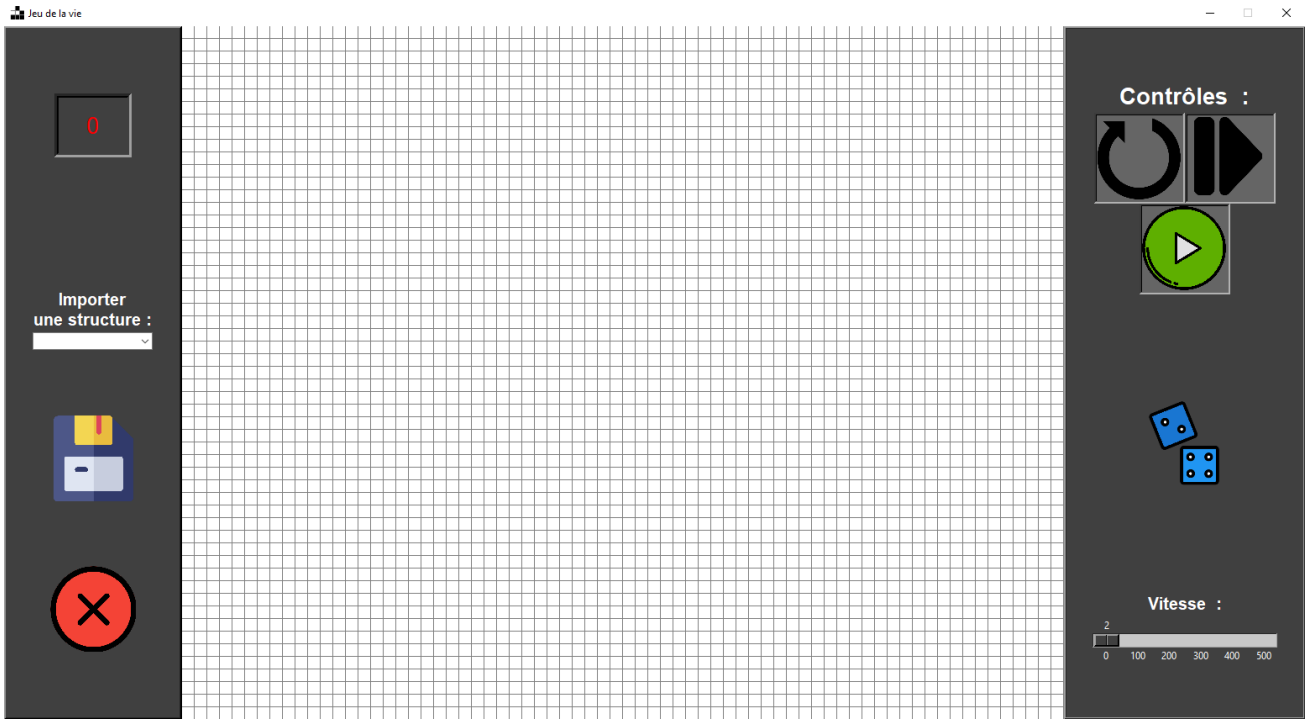
Pour faciliter le bon déroulement du projet, nous devions répartir le code en différents fichiers et dossiers, laissant au passage une meilleure opportunité de nous répartir le travail, au sein du groupe. De ce fait, trois fichiers *python* font s'exécuter le projet.

A) Le programme principal

Le fichier *jeu_de_la_vie.py* est le cœur de notre programme. En effet, c'est lui qui contient toutes les fonctions qui font évoluer notre grille. Il affiche également la grille de manière très rudimentaire, dans la console.

B) L'interface graphique

Pour palier à cet affichage dans la console qui est loin d'être attractif pour nos yeux, nous nous sommes plongés dans la bibliothèque *Tkinter* afin de pouvoir rendre notre programme plus abouti, mais surtout plus esthétique. Voici le rendu lorsque le fichier *affichage_jeu_de_la_vie.*py est lancé :



C) Les tests

Les tests ont été une partie importante de notre projet. En effet, bien que le début de ce fichier ait été très peu lisible, il a été amélioré pour afficher les tests passés d'une manière plus propre, tant pour celui qui ouvre le fichier *test.py*, mais aussi pour celui qui l'exécute.

Pour tester les structures, on enregistre un dictionnaire dont la clef est le nom de la structure ; et la valeur, le nombre d'itérations nécessaire à effectuer. Puis, avec une boucle on teste chaque structure en chargeant celle de base et celle qui devrait être la grille finale au bout de *n itérations*.

Voici un rendu de l'affichage du test en console :

```
1/29    regles_de_base.txt      OK
2/29    bateau.txt              OK
3/29    bigs.txt                OK
4/29    bloc.txt                OK
5/29    mare.txt                OK
6/29    pain.txt                OK
7/29    cthulhu.txt             OK
8/29    balise.txt              OK
9/29    clignotant.txt          OK
10/29   grenouille.txt          OK
11/29   Lunettes.txt            OK
12/29   machine à laver.txt     OK
13/29   Why not.txt             OK
14/29   Grenouille 2.txt        OK
15/29   pulsar.txt              OK
16/29   Croix.txt               OK
17/29   Wavefront.txt           OK
18/29   Fontaine.txt            OK
19/29   101.txt                 OK
20/29   coeur.txt               OK
21/29   volcan.txt              OK
22/29   Fiole.txt               OK
23/29   pentadecathlon.txt      OK
24/29   test1.txt               OK
25/29   test2.txt               OK
26/29   test3.txt               OK
27/29   test4.txt               OK
28/29   test5.txt               OK
29/29   test6.txt               OK
----------------------------------------
Synthèse : 29/29 tests passés (100.0%)
```

3. Les difficultés rencontrées

A) Compter le nombre de voisines vivantes

La première difficulté que nous avons rencontrée a été celle de compter les voisines vivantes autour de chaque cellule de la grille. C'est une étape qui nous paraissait relativement aisée, mais elle s'est avérée plus complexe que nous le pensions. En effet, il fallait à tout prix éviter l'*IndexError*, ainsi que ne pas comptabiliser la cellule que l'on regarde dans le total de cellules vivantes autour d'elle. Nous sommes donc arrivés, avec de l'aide, à trouver une solution qui fontionne :

```python
def compte_voisines_vivantes(grille, x, y):
    """Compte le nombre de cellules vivantes autour d'une cellule de la grille.

    :param grille: Grille des cellules.
    :type grille: list.
    :param x: colonne où se trouve la cellule autour de laquelle on souhaite compter les cellules vivantes.
    :type x: int.
    :param y: ligne où se trouve la cellule autour de laquelle on souhaite compter les cellules vivantes.
    :type y: int.

    :return: Le nombre de cellules vivantes autour de la cellule observée.
    :rtype: int.
    """
    voisines = 0

    # on parcourt toutes les cases voisines de la case donnée :
    for i in range(x-1, x+2):
        for j in range(y-1, y+2):
            # si l'emplacement qu'on regarde n'est pas celui de la cellule et qu'on ne sort pas du tableau :
            if not (i == x and j == y) and 0 <= i < len(grille) and 0 <= j < len(grille[0]):
                voisines += grille[i][j]

    return voisines
```

-> Cette fonction se trouve dans le fichier *jeu_de_la_vie.py*

B) Programme trop gourmand

En demandant de l'aide, il nous a été fait remarquer que notre programme était trop gourmand. Et en effet il l'était. À chaque fois que l'on affichait la grille à l'aide de *Tkinter*, nous créions de nouveaux rectangles, ceux qui représentent les cellules. Le problème c'est que les anciens restaient en mémoires, et donc, la saturait. La raison pour laquelle nous ne nous en étions pas aperçu c'est que l'on travaillait sur de petites grilles, et donc, que le nombre de rectangles à stocker était faible.

Pour remédier à ce problème, nous avons décidé de créer tous les rectangles au début de l'exécution du programme et de les modifier par la suite :

Création :

```
# on dessine la grille et on stocke les cases (pour les modifier plus tard)
cellules = [[canvas.create_rectangle(idx_to_coord(l, c), outline="gray")
             for l in range(nlignes)] for c in range(ncolonnes)]
```

Exemple de modification :

```
canvas.itemconfig(cellules[c][l], fill=couleur)
```

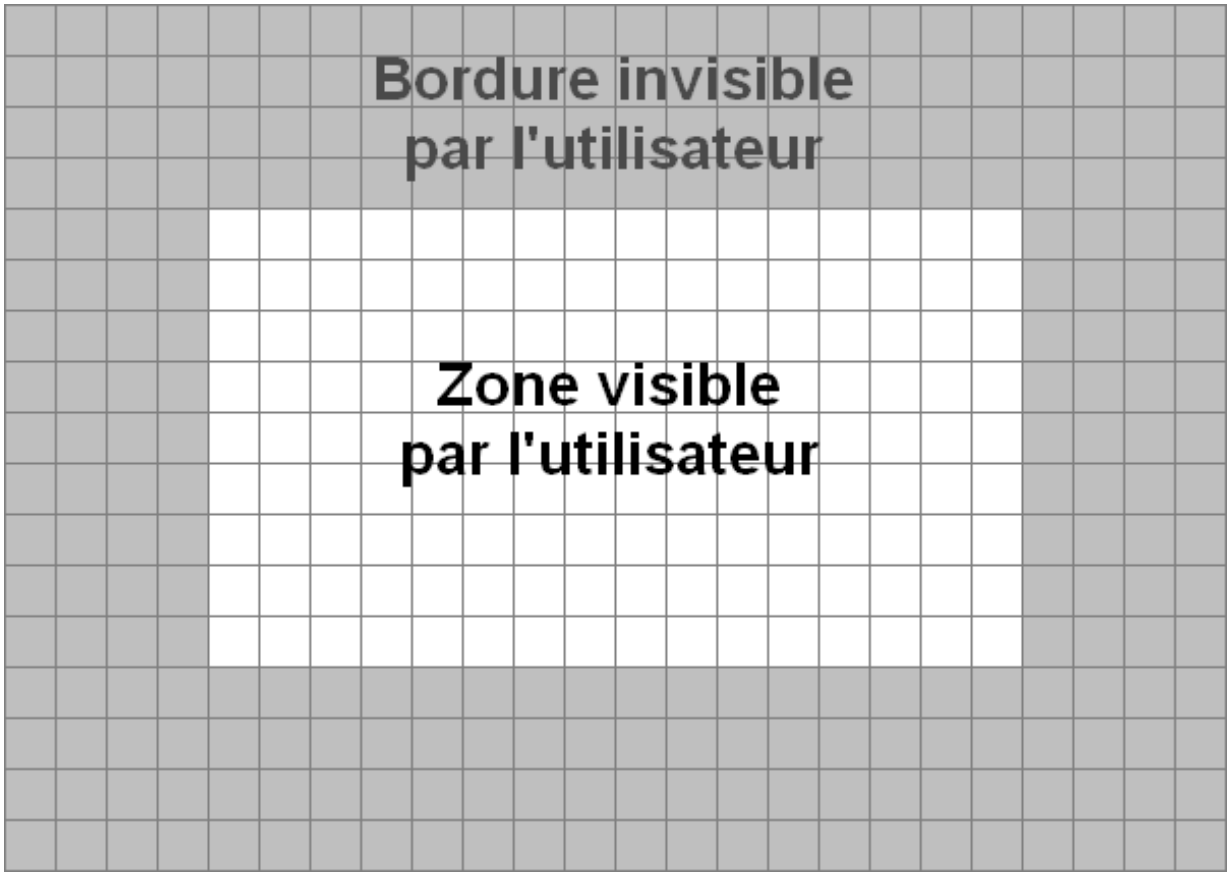
C) Gestion des bords

La gestion de bord a sans doute été la partie la plus délicate du projet. Le problème vient en réalité de notre manière de procéder. En effet, le jeu de la vie est censé simuler une grille de taille infinie. Cependant, avec la méthode que nous avons implémentée, on stocke une grille de taille définie : c'est ce qui cause le problème des bords.

Pour mieux illustrer ce problème, voici une simulation de ce qui se passait avec un planeur, une structure qui se déplace : planeur.gif. (gif présent dans le dossier du compte rendu si un problème se présente)

Cette figure qui se forme au bord, et qu'on appelle le bloc ne devrait pas se former. Alors, pour remédier à ce problème, nous avons essayé de modifier la fonction *compte_voisines_vivantes()*, mais sans succès. Nous avons donc pris la décision de non pas régler le problème mais de le décaler. Nous avons créé une bordure qui entoure notre grille et qui fait effet de tampon entre la bordure visible par l'utilisateur et la bordure réelle et limité de notre grille.

De ce fait, une grille que l'on définirait de taille 16x9 ressemblerait à cela en mémoire :



Avec cette nouvelle version, voici ce qui se passe avec un planeur : planeur fixé.gif. (gif présent dans le dossier du compte rendu si un problème se présente)

Le problème existe toujours, mais il n'est plus visible par l'utilisateur. Nous sommes conscient que laisser des cellules de cette manière dans la bordure pourrait perturber l'évolution de la grille, mais à ce jour c'est la seule solution que nous ayons trouvé qui soit fonctionnelle.

D) Affichage sans les bords

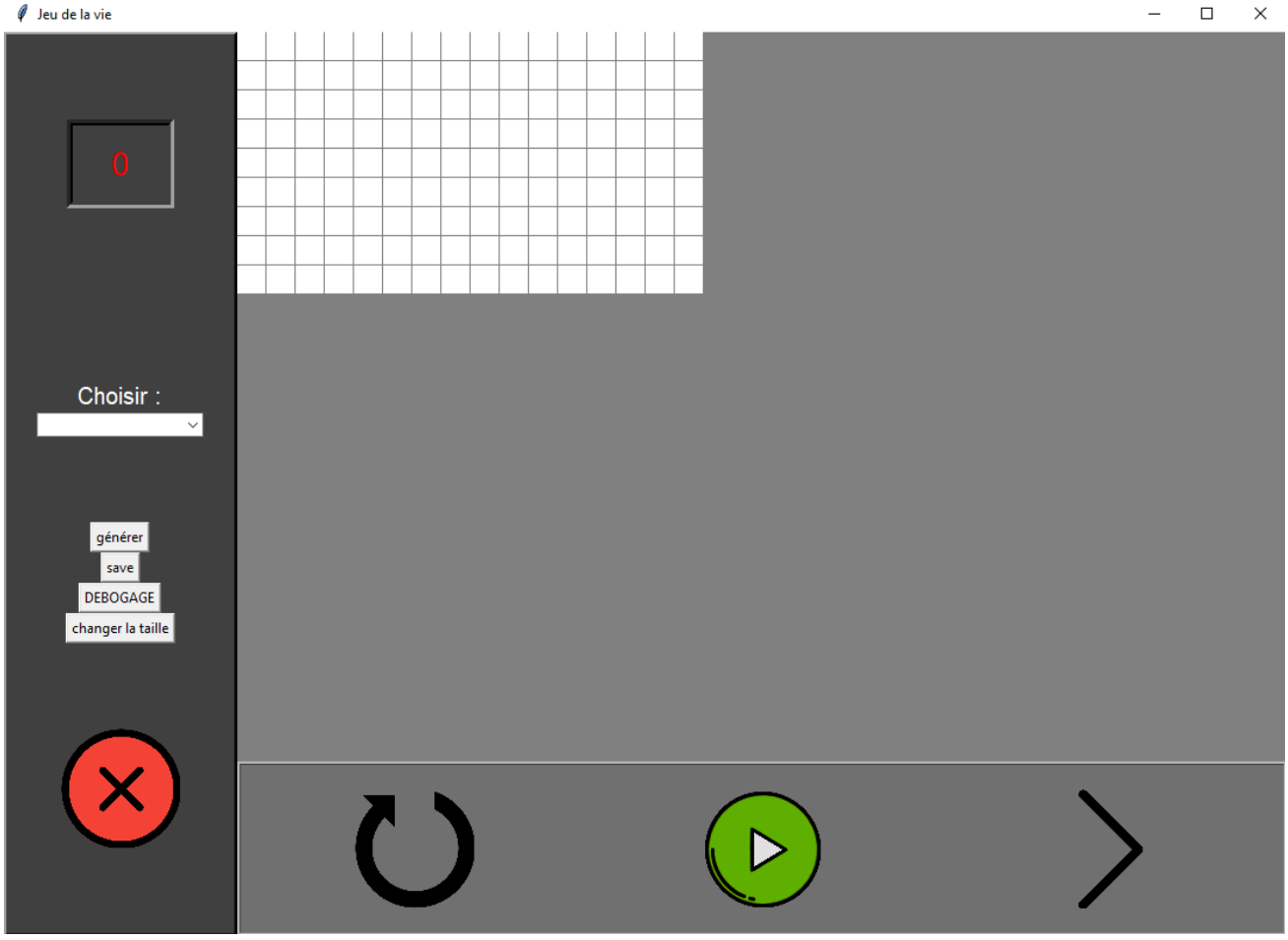
En résolvant le problème de la gestion des bords, un autre s'est créé. Il s'agissait de l'affichage de la grille sans les bords. Pour le régler, nous avons eu a adapter notre code avec la nouvelle variable *rab_bordure*. Les fonctions impactées ont donc été *idx_to_coord*, *coord_to_idx*, *clic_canvas* et *display_grid*. Désormais, le schéma présenté ci-dessus était celui affiché dans la fenêtre *Tkinter*, sans les bords.

E) Compatibilité avec Linux

Lors d'un point sur le projet, on nous a fait remarquer qu'une ligne de code rendait l'exécution de l'interface graphique impossible sur *Linux*. C'est donc en modifiant la ligne qui chargeait l'icône de notre fenêtre que le problème a été résolu. Le format de l'icône est désormais en *PNG* et non plus en *ICO*.

F) Changement de la taille de la grille

Afin de rendre notre projet plus complet, nous avions souhaité intégrer la fonctionnalité de changer la taille de la grille pendant l'exécution du programme. Cependant, lorsque nous changions les variables *nlignes, ncolonnes* et *grid_size* pour que la grille s'emboite parfaitement avec le canvas, il se produisait ce genre de chose :



Nous aurions probablement pu nous en sortir avec plus de temps, mais puisque la date de rendu du projet approchait et que quelques fonctionnalités plus importantes n'étaient pas abouties, nous avons décidé d'abandonner cette fonctionnalité que nous jugions de second plan.

4. Apprentissage

Ce projet nous a permis d'apprendre plus choses sur le langage Python, la librairie *Tkinter*, ainsi que sur un outil de communication et de partage : *GitHub*. Même s'il nous reste un grand nombre de fonctionnalités à découvrir sur cet outil très puissant, nous en avons découvert les bases, amplement suffisantes à notre niveau.

Au sein de la librairie *Tkinter*, nous avons découvert comment actualiser l'affichage d'un *Canvas* avec la méthode *.after(vitesse, fonction)* ainsi que comment la stopper à l'aider de *.after_cancel(id)*. Nous avons aussi appris à utiliser de nombreux widgets, dont le *Scale*, celui qui nous sert de *slider* pour gérer la vitesse d'affichage de notre fenêtre.

Même au sein de ce compte rendu, nous avons pu expérimenter la création de *gifs* servant à l'illustration du problème de gestion des bords.