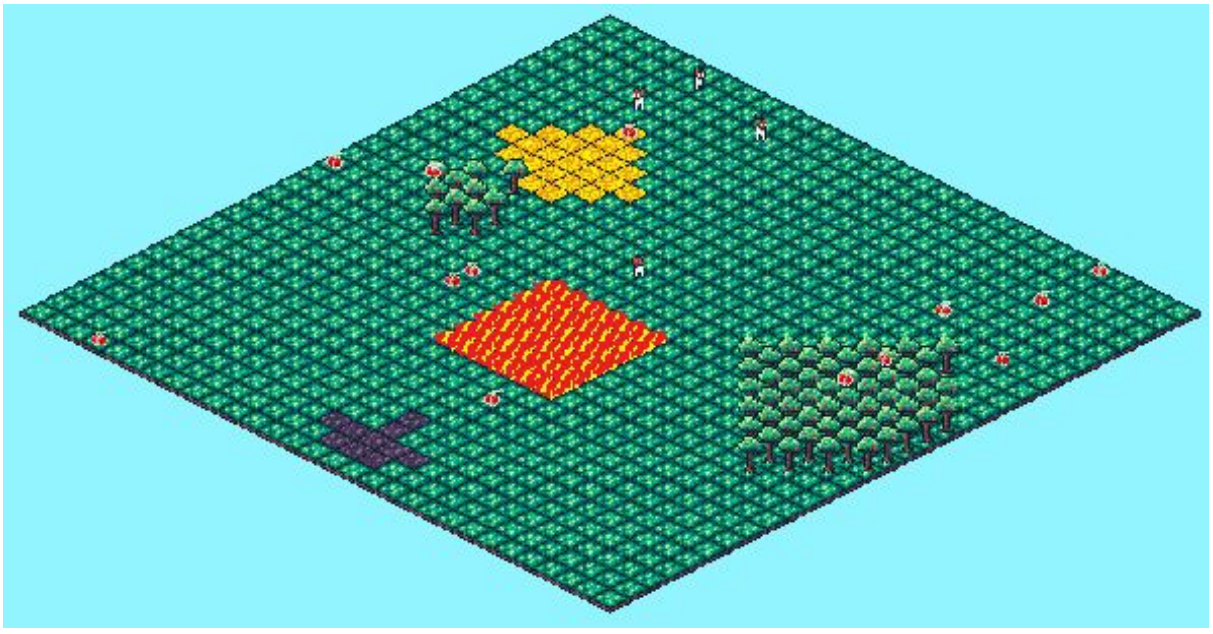
Rapport Projet Programmation

Groupe 3 :
Alexandre Giard | Antoine Moutonnet | Pablo Fevrier | Colin Poubel | Léo Vansimay | Van Tuan Bui



Introduction

L’objectif de ce projet est de réaliser une simulation d’un monde, contenant une espèce (Les Bobs) qui mute au fur et à mesure du jeu en plusieurs sous-espèces à travers la reproduction. Dans ce rapport nous allons présenter nos choix de modélisation, les difficultés que nous avons rencontrées, et comment nous nous sommes organisés. Si dans ce rapport nous n’abordons pas une certaine caractéristique d’un Bob c’est parce que nous l’avons implémenté comme dans l’énoncé du sujet et que nous n’avons pas rencontré de difficultés particulières.

Plan

I) Organisation

Au départ nous nous sommes concentrés sur une version simplifiée du projet, avec juste des bobs et de la nourriture (sans addons). Nous avons commencé par faire un “brainstorming” de toutes les classes/fonctions dont nous pensions avoir besoin lors de la 1ère version du projet.
Nous avons séparé notre groupe de projet en 2 groupes. Le groupe IA programme le comportement des Bob (Alexandre, Antoine, Leo et Colin). Le groupe Interface programme les graphismes et l’interface avec l’utilisateur (Tuan et Pablo).
Une fois la première version terminée, dans le groupe IA nous avons codé la masse et la vélocité en parallèle, un groupe s’occupait de la masse, et l’autre de la vélocité. Puis nous avons eu besoin de fusionner nos deux codes. C’est là que nous avons commencé à utiliser GIT, nous avons créé 2 branches avec les codes des 2 groupes, puis fait un merge. GIT nous a beaucoup aidés à organiser notre projet, mais nous a aussi causé quelques soucis, notamment en ce qui concerne les merge et la synchronisation entre branches, souvent quelqu’un push sur sa branche et ne met pas à jour les branches concernées qui elles évoluent aussi, du coup on se retrouve avec des grands écarts de versions entre nos branches.

II) Choix de modélisation et difficultés rencontrés

Le jeu est composés de 6 différents fichiers :

- class_spore rassemblant les classes et les fonctions utilisées lors de l'exécution en soi de la simulation
- class_graphics qui implémente le rendu graphique au sein de pygame, ainsi que les fonctionnalités de zoom, de déplacement de la map ou de sortie de l’application.
- class_parameters contient la classe Parameters, elle sera instanciée au lancement de notre programme et sera un objet qu’on pourrait considérer comme global car utile à toute partie de notre jeu.
- graphs est le fichier implémentant le calcul et le rendu des courbes et graphiques permettant de visualiser l’évolution des populations et des caractéristiques des bobs en fonction du temps.
- main implémente la classe Game, c’est la classe contenant la boucle d’exécution de la simulation.
- spore_UI est le fichier réalisant l’interface utilisateur, c’est à partir d’elle que le jeu s’exécute. C’est aussi ici que se réalise la ramification des threads au sein du jeu.

Pour l'implémentation du **monde**, nous avons créé un tableau de tableau de Tiles pour simuler une matrice carrée. Les Tiles représentent les cases du monde, elles contiennent la liste de Bob sur la case et la quantité de nourriture sur la case. Pour distribuer la nourriture on incrémente simplement la nourriture sur des cases choisies au hasard. Pour supprimer les nourriture des cases à la fin d'un jour, on parcourt toutes les cases et on met à zéro leur nourriture. Nous avons choisi d'ajouter une liste de Bob globale pour gérer les actions des Bob, en plus de la liste par case pour gérer les interactions entre les Bob sur la même case comme pour les combats par exemple.

Pour le **mouvement des Bob**, nous avons commencé par l'implémenter avec une série de 'if' ce qui fonctionnait mais occupait beaucoup de place dans le code et était peu lisible. Nous avons donc eu l'idée de faire une version récursive de cette fonction en créant des sous-fonctions move_[left, right, up, down] appelées au hasard selon la position du Bob. Le défaut de cette nouvelle version est qu'elle est assez gourmande en calcul lorsque le Bob se trouve dans un coin, car quand il ne peut pas aller dans une direction il appelle au hasard une des 3 autres directions, dans le cas où il tombe sur un autre mur/case visitée il rappelle une des 3 autres directions et ainsi de suite, pouvant dans de rares cas provoquer un débordement de la pile d'exécution.

Pseudo code simplifié de la méthode move:

```
Move(self, world, n=-1, tile_to_avoid = [] )
    Aiguillage n :
        cas -1 :
            n = randint(0, 3)
        cas 0 :
            Si on peut se déplacer à droite:
                self.move_right
            Sinon:
                self.move(n=randint(1, 3))
            break
        cas 1 :
            déplacement à gauche si possible...
            break
        cas 2 :
            déplacement en haut si possible...
            break
        cas 3 :
            déplacement en bas si possible...
            break
```

Afin d'implémenter la **perception des Bob**, nous avons commencé par implémenter une fonction globale qui servait à diviser en plusieurs catégories ce qu'un Bob pouvait voir : la nourriture, les proies potentielles et les prédateurs. Pour cela, nous avons dû implémenter une fonction dist() afin de calculer la distance de Manhattan entre deux coordonnées. cela nous a permis de remplir les listes correspondant à la nourriture, aux proies et aux prédateurs avec une expression en compréhension et de façon optimisée. Pour cela, on

commence par définir un carré centré autour du Bob, puis on parcourt toutes les Tiles de ce carré, en gardant celles qui sont à une distance inférieure où égale à la perception du Bob, ce qui nous donne un “cercle”. On obtient ainsi une liste des Tiles que le Bob peut voir. On applique ensuite sur cette liste trois filtres pour obtenir la liste des Tiles sur lesquelles il y a de la nourriture, une proie ou un prédateur. Nous avons ensuite défini deux méthodes, fly_you_bob() et there_is_always_a_bigger_bob(), qui correspondent respectivement aux comportements de fuite et de prédateur. Nous les avons ensuite regroupées dans une méthode hunt_or_be_hunted(), qui détermine si le Bob doit fuir un prédateur, se diriger vers de la nourriture ou chasser une proie (dans cet ordre de priorité).
Nous avons rencontré plusieurs problèmes lors de l’implémentation de cette caractéristique, car les Bob pouvaient se voir et donc se considérer comme une proie ou un prédateur potentiel et cherchait tout le temps à se fuir. Après avoir réglé ce problème, nous nous sommes rendus compte que nous avions été trop inclusifs dans notre condition déterminant si une case possédait de la nourriture, car les Bob possédant de la perception ne se déplaçaient plus, considérant que la case où ils se trouvaient contenait de la nourriture.

Dans le cas de la **mémoire des Bob**, pour avoir les cases vues par le Bob au tour précédent nous avons dû créer une fonction look() qui remplit des attributs (tableaux de Tiles) de la class Bob de tout ce qu’il voit, et ce qu’il a vu au dernier tour. Cela a beaucoup alourdi la class Bob, mais nous a permis de grandement simplifier la gestion de la mémoire. Nous avons aussi dû changer la fonction move pour prendre en compte les cases visitées qu’il faut éviter, la version récursive de la fonction move nous a beaucoup simplifiée la tâche. Dans le cas où le Bob poursuit une cible il ne doit pas prendre en compte les cases visitées, nous avons donc mis les cases visitées en argument optionnel de la fonction move. Nous avons pu réutiliser la fonction there_is_always_a_bigger_bob() pour diriger le Bob vers les cases de nourriture en mémoire. Dans notre implémentation le Bob va toujours prioriser la mise en mémoire de la nourriture qui sort de son champ de vision, quitte à retirer de sa mémoire les cases visitées les plus anciennes.

Pseudo code de la mémoire :

```
memorise(self, world)
    Si une case de nourriture est en mémoire et dans son champ de vision :
        self.memory.remove(case de nourriture)

    Si il y a de la nourriture qui sort de son champ de vision :
        Si il n’a pas assez de points de mémoire il oublie des cases visitées:
            self.memory.remove(case visitée)
        Si il a assez de points de mémoire:
            self.memory.append(case de nourriture)

    Si il n’a pas assez de points pour mémoriser la dernière case visitée:
        self.memory.remove(case visitée)
    Si il a assez de points de mémoire pour mémoriser la dernière case visitée:
        self.memory.append(case visitée)
```

Pour la **vélocité des Bob** nous avons d'abord implémenté la version avec les actions partielles, mais nous avons eu des difficultés à déplacer le Bob de manière partielle, nous avons donc implémenté la version avec buffer. Comme la vélocité est un nombre à virgule flottante nous avons dû faire attention aux arrondis et aux conditions l'impliquant.

Le comportement des Bob dans un tick est géré par la méthode **action()**, qui commence le tick en déterminant le nombre d'actions du Bob. On commence par déterminer le nombre d'actions du Bob, avec la formule suivante :

$$NbActions = velocity + E(velocity\ buffer)$$

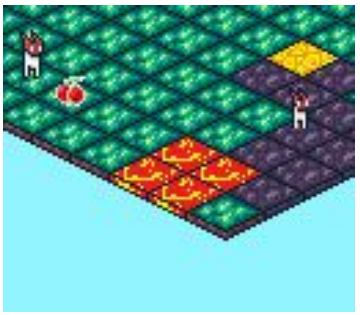
Avec E() la fonction partie entière. On remplit ensuite le buffer avec l'excédent non entier de vélocité pour le prochain tour. Ensuite, si le Bob se trouve dans un marécage, on divise son nombre d'actions par deux (avec un minimum de 1) pour simuler la difficulté de se déplacer dans un marécage. Ensuite, tant que le Bob a des actions disponibles et de l'énergie, il effectue les actions suivantes :

- Regarder autour de lui
- Mettre à jour ce dont il se souvient en fonction de ce qu'il voit
- Se reproduire s'il en est capable
- Se nourrir si la case sur laquelle il se trouve possède de la nourriture
- Fuir s'il voit un prédateur, sinon se déplacer vers la source de nourriture la plus abondante s'il en perçoit une, sinon il se déplace aléatoirement
- Se battre avec les Bob plus petits qui se trouvent sur sa case
- Consommer de l'énergie en fonction de s'il s'est déplacé à ce tick

Pour les deux types de **reproductions**, nous avons mis un booléen afin de déterminer si on se servait de cette méthode au cours d'une exécution, chaque type de reproduction est une méthode de la classe Bob, et teste si le bob a assez d'énergie pour se reproduire. Les erreurs rencontrées venaient principalement de fautes de frappes dans les inégalités dans les tests, ce qui causait des morts en couches pour le Bob parent, ou bien pire encore pour la reproduction sexuée, où si le bob qui agit possède assez d'énergie, et est sur la même case qu'un autre bob, même si le deuxième bob n'avait pas assez d'énergie, la reproduction se lançait, et le parent 2 mourrait donc dans le processus. Mis à part ces erreurs, l'implémentation de ces deux méthodes s'est déroulée sans problèmes majeurs.

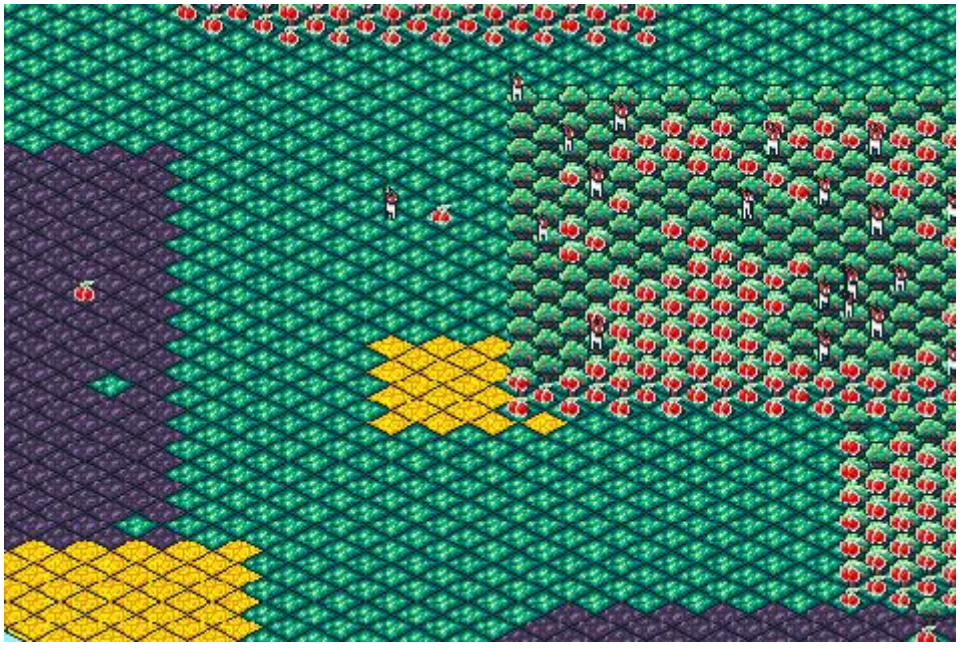
En ce qui concerne **the_floor_is_lava**, une fois que nous avions terminé les caractéristiques de base du projet, nous avons décidé de modéliser d'autres phénomènes qui nous paraissaient intéressants. Ainsi, avec the_floor_is_lava, nous simulons une coulée de lave qui tue tous les bobs et enlève la nourriture sur son passage. On a donc un type de catastrophe naturelle qui peut être modélisé pour étudier l'évolution des bobs avec ce phénomène supplémentaire - phénomène que l'utilisateur peut modifier à sa guise en intervenant sur la probabilité de coulée de lave par tick ou en choisissant le pourcentage de la map qui va être impacté. L'emplacement de la coulée de lave est choisi aléatoirement et les bobs ne peuvent ainsi pas l'anticiper et s'écarter de cette zone.

capture d’écran d'événement lava



Pour l’implémentation des **biomes**, nous avons tout d’abords réfléchi à différents biomes possibles, et les modifications des règles qu’ils induisent. Une fois la liste des biomes dressés, nous avons créé un dictionnaire, pour associer chaque type de biome à un chiffre, afin de rendre la compréhension du code plus facile. Nous avons donc ajouté un attribut aux classes Tiles, afin de prendre en compte l’environnement présent sur la Tile. Ensuite, pour la génération de la map, nous choisissons de prendre un nombre de cases au hasard, et, à l’aide d’une loi de probabilité, déterminons la taille de la “tâche” qu’occupe le biome. Nous avons ensuite adapté l’affichage graphique pour changer la couleur des Tiles en fonctions du biome à représenter.

capture d’écran du monde avec les biomes :

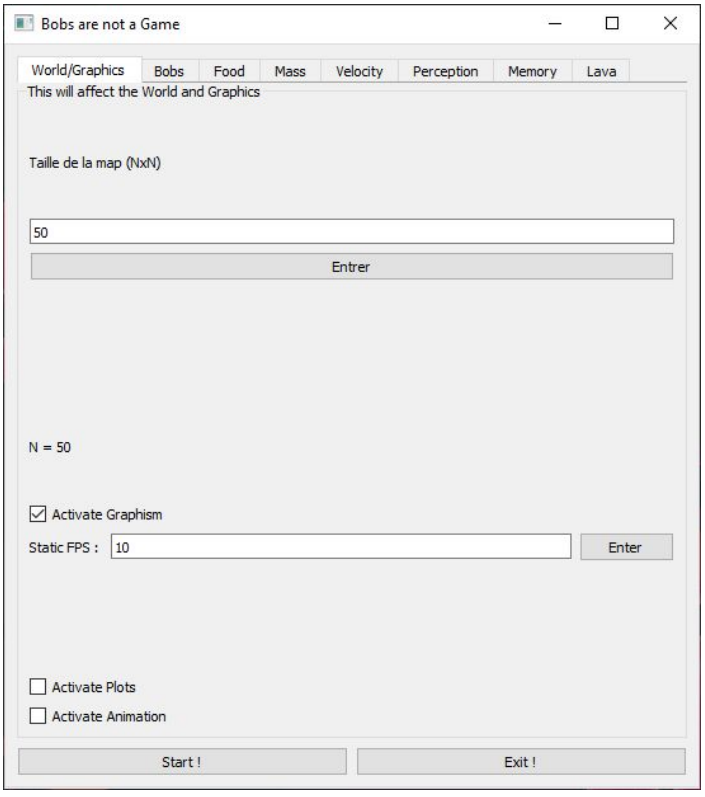


Pour les **graphismes**: nous utilisons la bibliothèque Pygame, qui nous a permis d’implémenter le monde en 2.5D. Deux versions des graphismes sont implémentées: la version avec animations et la version sans animations.
La version sans animations affiche les bobs après qu’ils ont fait toutes leurs actions dans un tic-tac(tick). Cette version est limitée, car les déplacements d’un bob lorsque sa vélocité est plus grande que 2 font qu’il est difficile d’observer ses actions.

Pour la version avec animations, on a animé les déplacements de bob et l'explosion de la lave. Le principe est qu'on change l'image de bob et de la lave dans chaque boucle pour créer l'effet d'animation.
Pour rendre les graphismes indépendants de la simulation, nous avons créé une classe Bob_graphic qui contient toutes les caractères de bob par rapport aux graphismes(par exemple: la taille du bob, l'image,...)
Pour adapter la taille du monde, nous avons créé deux surfaces(display et screen), screen est la fenêtre d'interface graphique de taille fixe, et display a une taille variable en fonction de la taille du monde. Le principe est qu'on dessine les images de bob, de nourritures, de case et les textes(tick et day) sur la surface display et après on transforme la surface display en surface screen.

Pour **intégrer l'interface** à la simulation, nous avons séparé les paramètres modifiables ou globaux dans un autre fichier nommé class_parameter, et on ajoute la variable params (qui est une instance de la classe Parameters) à toutes les fonctions pour appliquer les changements quand on modifie les variables sur l'interface. Cette instance et cette classe est vitale et centrale pour notre Projet. C'est cet objet qui instaure le socle commun entre les Threads, en effet, c'est notre instance params qui implémente les modifications réalisées dans l'interface. Cependant, afin d'y arriver, un long cheminement fut nécessaire.

L'idée première fut d'utiliser classe_spore comme fichier de « stockage » des paramètres globaux, cependant, en tant que fichier contenant les différentes classes et fonctions de notre projet, ceci n'était pas son rôle et l'instancier aurait eu une logique faible pour ne pas dire nulle.

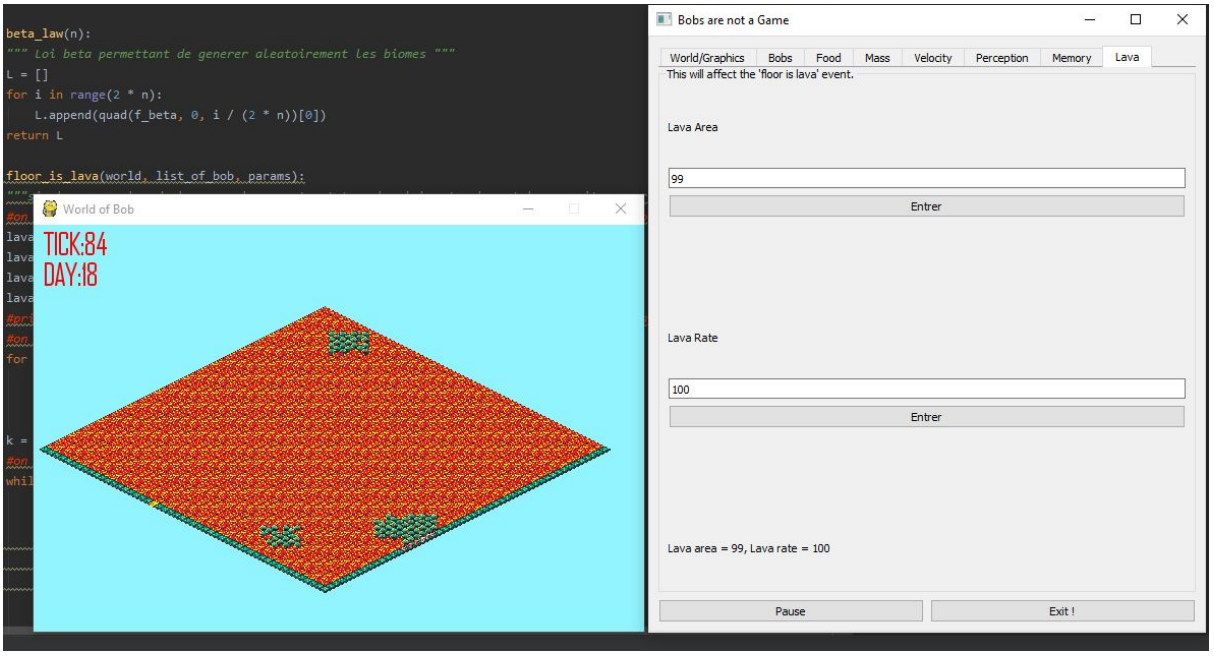


Vue de l'interface graphique et de son onglet d'accueil

L'interface utilisateur est réalisé à partir de PyQt5, du fait de sa rapidité, efficacité et esthétique. On l'implémente à travers une classe héritant de l'objet QMainWindow propre à PyQt5. L'interface est composée d'onglets propre à chaque caractéristiques ou familles de paramètres que l'on souhaite modifier. L'implémentation de ces onglets ne s'est pas avérée être très compliquée, si ce n'est longue et pénible. Il a fallu tout d'abord comprendre comment PyQt5 fonctionnait et quels objets propres à la bibliothèque il était cohérent d'utiliser. La suite se résume par un code dont la complexité ne se trouve que par son gabarit. Toutefois, l'interface met en place des éléments clé pour le bon fonctionnement du jeu. Le multithreading de l'interface et de l'exécution se réalise avec l'implémentation de l'interface. PyQt5 réalise le multithreading en instanciant un QThread depuis le processus d'origine, ce faisant, on instancie aussi un objet contenant le long calcul ou dans notre cas, notre boucle de simulation de Bobs. Cette méthode est d'ailleurs « décorée » par *@pyqtSlot()* indiquant que cette méthode ne doit être lancé qu'avec le démarrage d'un QThread. L'objet instancié va ensuite être enveloppé par le QThread, permettant son exécution dans un fil d'exécution à part. QThread n'est ainsi pas construit pour contenir le processus à exécuter en parallèle, c'est un wrapper.

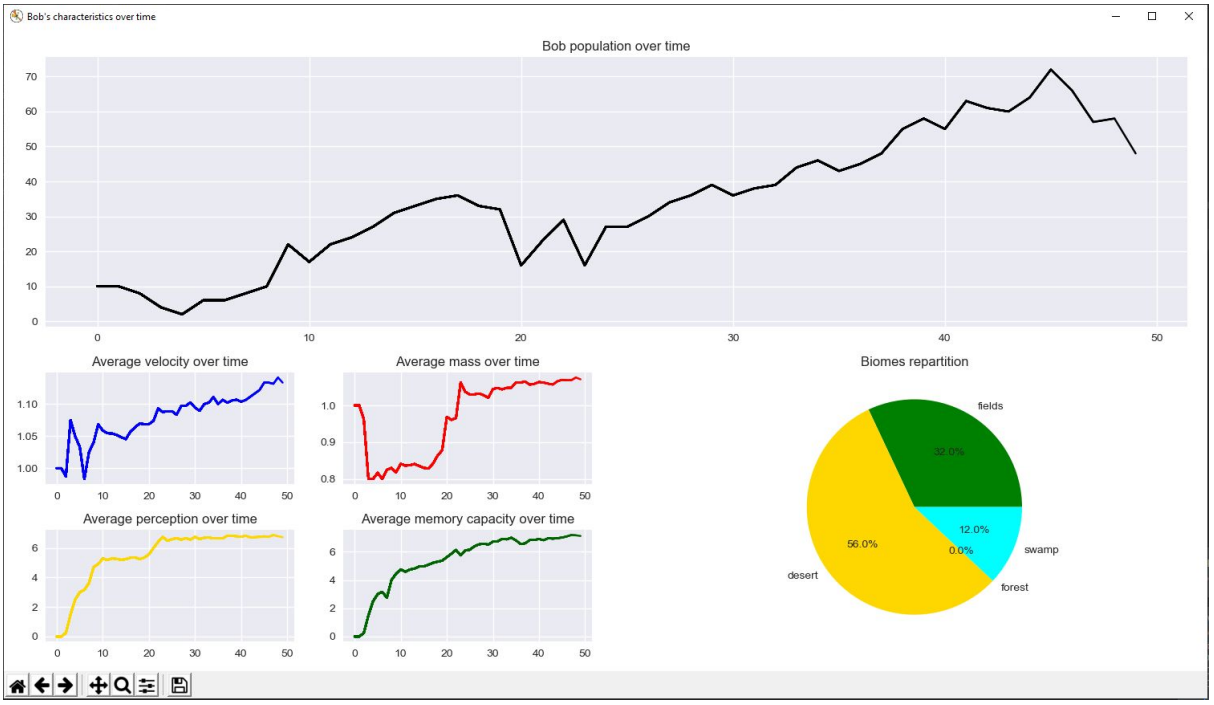
L'implémentation des multiples fils d'exécution fut la tâche la plus ardue. Pygame n'est pas fait et généralement empêche toute sorte de multithreading, du fait qu'il se repose sur SDL et possède sa propre boucle main.

Elle a malgré tout bien été possible, grâce au fait que l'on lance pygame depuis une instance de classe, elle-même contenue dans l'objet qui est entourée du QThread. Un exemple montrant la difficulté de les associés fut au moment de programmer la fin de l'exécution via un bouton exit permettant de quitter le programme correctement. Ainsi, ni PyQt5 ni pygame ne permettait à l'un ou l'autre de lui dire de terminer son exécution. Le bouton exit met donc à vrai un booléen indiquant la fin de l'exécution afin que la boucle main construite dans pygame mette fin à l'application pygame, l'interface en même temps appelle sa fonction close et se termine correctement.



Pygame et l'interface côte à côte.

Afin de pouvoir donner du sens à la simulation, nous avons fait en sorte de pouvoir afficher des **graphiques** représentant l'évolution des différentes caractéristiques des Bobs. L'objectif est de pouvoir analyser les effets des modifications des différents paramètres de la simulation sur la population, et ainsi d'étudier l'évolution de cette population. La gestion des graphiques se fait en utilisant matplotlib, librairie qui a un fonctionnement similaire à Matlab. Au lancement de la simulation, une figure est créée avec une grille de 4 * 4 cases, puis on crée les différents graphiques en leur affectant des cases. Chaque graphique est formé à partir de deux listes, une commune à tous qui représente les différents jours, et une autre représentant la valeur de la caractéristique à chaque jour. A la fin de chaque jour, les listes sont mises à jour en ajoutant la valeur de la caractéristique courante, puis les graphiques sont recalculés et affichés. La seule exception est le diagramme "camembert" représentant la répartition des biomes, qui n'est calculé qu'une seule fois à la création car il n'est pas modifié ensuite. On obtient donc la fenêtre suivante :



On peut donc avoir une bonne idée de l'évolution des caractéristiques des Bob en fonction du temps, afin de tirer une interprétation de la simulation. Cependant, cette méthode pour réaliser des graphiques n'est pas optimale, car les listes ne font que s'accroître au fur et à mesure, et le graphiques doivent être recalculés entièrement à chaque jour. Cela peut donc avoir un impact sur la mémoire si l'on souhaite laisser la simulation tourner pendant un nombre important de jours.

Un des problèmes que l'on a rencontré est la **suppression des Bob** a un moment donné on supprimait les Bob à plusieurs endroits dans le code ce qui causait des erreurs. De plus, un Bob qui pouvait faire 2 actions (avec 2 de vélocité) pouvait mourir lors de sa première action et faire sa deuxième action quand même. Ce bug était facilement remarquable lors de l'exécution de la **fonction de combat**, car le bob se battait avec tous ses adversaires potentiels, il pouvait donc se faire tuer lors de son premier combat, et ensuite se battre avec les autres, le bug se remarquait car le bob mourrait deux fois, ce qui faisait planter le programme. Ceci a amené à modifier la fonction duel_of_fate afin que le combat ne soit lancé que si le bob initiant le combat est certain de gagner.

Un autre problème que nous avons rencontré est la **gestion de l'énergie** des Bob, à partir de l'ajout de la vélocité, on commençait à avoir des formules de consommation d'énergie un peu partout et dans différentes méthodes. Nous avons aussi remarqué que le Bob dépensait de l'énergie à chaque fois qu'il se déplaçait lors d'un même tick. Or, par exemple lorsqu'un Bob a 2 de vélocité il peut se déplacer 2 fois mais il ne dépense qu'une fois la formule d'énergie du mouvement, le fait qu'il dépense 2 fois de l'énergie dans notre simulation rendait la vélocité extrêmement désavantageuse. Pour résoudre ce problème nous avons centralisé au maximum l'énergie dans une formule dans action, et ajouté un attribut 'has_moved' au Bob pour savoir si le Bob consomme 0,5 d'énergie, ou la formule.

III) Conclusion

Ce projet nous a permi de nous familiariser avec la gestion de projet et avec le langage python. Nous avons remarqué qu'il est très important de s'organiser dès le début du projet. Nous nous sommes beaucoup habitués à l'utilisation de GIT, même si nous avons encore des difficultés avec notamment les merge et la synchronisation entre branches.
Au niveau de la simulation, ce projet nous a permis de mieux comprendre la sélection naturelle. On a notamment pu directement constater l'influence de la vision, la mémoire, la force, et la vitesse sur la survie (ou pas) d'une espèce. On a remarqué que les Bob arrivaient à un équilibre au bout d'une dizaine de jours, les Bob favorisent différentes caractéristiques en fonctions du nombre de Bob, de la quantité de nourriture distribuée, et de la carte (taille et biomes). L'implémentation de la lave, nous a aussi permis de voir la réaction d'une espèce face à une catastrophe naturelle.
Nous avons beaucoup aimé ce projet, car le thème est plutôt 'fun' et nous avons pu mettre en pratique nos connaissances en python. Nous avons aussi beaucoup aimé découvrir différents modules python, notamment pour faire les graphiques, les graphismes, et l'interface.
Si nous avions eu plus de temps nous aurions ajouté plus de caractéristiques aux Bobs, et plus d'add-ons (ex: pouvoir contrôler un Bob pour rendre le jeu plus interactif).

Annexe: Répartition du travail (nombre de points à distribuer: 108) :

Alexandre (20 points):

Mouvement
Fonction action
Perception
fly_you_bob (fuite d'un bob)
Vélocité (version sans buffer)
Vélocité (version avec buffer)
Mémoire
Débogage
Adaptation des fonctions aux nouveaux add-ons
Script mémoire + perception
Script fuite + chasse + combat
Docstrings
User guide
Ajout de la modification de la formule d'énergie + paramètres lave dans l'interface

Tuan (19 points):

Graphismes(Pygame): deux versions(animation et sans animation)
Aider à déboguer dans IA : découverte de bugs liés à la modélisation
Intégrer l'interface d'utilisateur à la simulation

Antoine (16 points):

Affichage console
Regroupement des variables globales
Mass
Mémoire (premières fonctions)
Duel of fate
The floor is lava

Leo (17 points):

Biomes
Duel of fate
Manage_energy
Création d'une branche pour corriger l'ensemble du code #PROPRE
reproduction (parthénogénèse et sexuée)
Remaniement de la fonction action
Réflexion sur les différents scripts de test à utiliser

Colin (18 points):

Fonction action
Vélocité (version sans buffer)
Perception
fly_you_bob

Débogage
Reproduction sexuée
Biomes
Graphiques : affichage et gestion des différentes statistiques sur la simulation
Remaniement de la fonction action
Réflexion sur la fonction move

Pablo (18 points):

Prototype de food
Threading
Interface avec l'utilisateur (PyQt5)