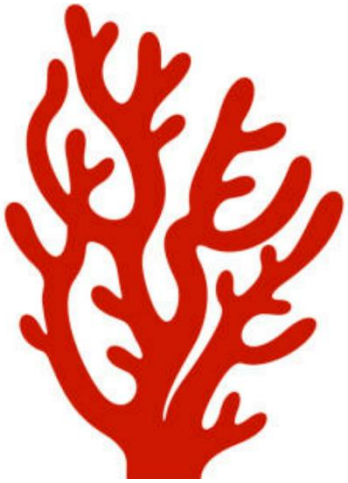

Simulateur corallien

Rapport de projet informatique

Adam Goux—Gateau, Titouan Maréchal

Mai 2022

Table des matières

1 – Introduction 3
2 – Pistes envisagées et hypothèses 3
3 – Fonctionnement général du programme 4
4 – La superclasse Organisme 5
4.1 – Principes généraux 5
4.2 – La sous-classe Corail 6
4.2.1 – Description 6
4.2.2 – Les méthodes principales de la classe Corail 6
4.3 – La classe plancton 7
4.3.1 – Description 7
4.3.2 – Méthode principale 7
5 – Les autres classes 7
5.1- La classe Rocher 7
5.2 – La classe Bouton 8
6 – Fonctions et paramètres de simulation 9
6.1 – Les fonctions de création, d'initialisation et de simulation 9
6.2 – Les paramètres de simulation 9
7 – Tests et figures imposées 10
7.1 – Tests 10
7.2 – Table des figures imposées choisies 10
8 – Premiers résultats avec Matplotlib 11
9 – Interface graphique et application finale 11
9.1 – Affichage des simulations avec Pygame 11
9.2 – Tracés de graphiques 12
10 – Conclusion 13

1 – Introduction

Notre programme Python tente de simuler l'évolution d'une colonie de coraux formant progressivement une gangue de calcaire dans un massif rocheux au sein d'un aquarium. L'objectif est de créer un décor initial constitué de rochers dans lequel vont se déplacer et interagir différents organismes : des organismes coralliens plus ou moins âgés et du plancton qui constitue leur source de nourriture. Au fil des tours, la colonie va se développer et occuper de plus en plus d'espace.

L'évolution de la colonie se fait sur un modèle de temps discret et un modèle spatial continu à deux dimensions. L'utilisateur choisit un jeu de paramètres, lance la simulation et observe graphiquement la croissance de la colonie. Le programme lui permet alors d'évaluer l'influence des différents paramètres sur le développement de la colonie.

Nous avons choisi un modèle spatial continu pour des raisons esthétiques et pour visualiser l'avancement du projet avec le module pyplot de la bibliothèque Matplotlib dès les premières phases de programmation. Nous nous sommes ensuite servis de la bibliothèque *pygame* pour construire notre IHM et visualiser en temps réel l'avancement de notre colonie.

Notre objectif était d'obtenir une grande variété de résultats en modélisant un grand nombre de paramètres physiques et biologiques. Nous voulions également faire interagir plusieurs colonies dans le même aquarium pour enrichir la diversité des scénarii possibles.

2 – Pistes envisagées et hypothèses

Nous avons fait plusieurs hypothèses pour modéliser le développement d'une colonie corallienne. La première a été d'assimiler les différentes entités à des disques. En effet, cette solution permet de simplifier grandement les calculs de distance entre entités. En ce qui concerne les rochers, on peut créer des formes complexes en superposant un grand nombre de disques de tailles différentes.

La seconde hypothèse concerne le milieu simulé. Nous avons décidé de travailler dans un espace fini, ce qui modélise le cas d'un aquarium : les organismes vivants ne peuvent pas sortir. Les entités simulées évoluant dans le plan, le point de vue de l'observateur de la simulation se fait du dessus (les rochers sont vus du dessus).

Le plancton est rajouté aléatoirement à chaque étape de simulation. Dans l'eau de mer, cela pourrait correspondre à un courant porteur. Dans notre cas, modélisant un aquarium fermé, on peut imaginer un courant généré artificiellement par une machine au sein de l'aquarium ou un ajout de nourriture par un observateur extérieur par le dessus de l'aquarium.

3 – Fonctionnement général du programme

L'espace de simulation est un aquarium modélisé par un rectangle dont les dimensions *dimx* et *dimy* sont choisies par l'utilisateur. Le programme est constitué de cinq modules : *organismes*, *rochers*, *tests_unitaires, affichage* et *simulation*. Les deux premiers modules contiennent les classes *Organismes* et *Rochers* ainsi que leurs méthodes. Elles permettent de créer et de contrôler tous les membres de l'écosystème évoluant dans l'aquarium.

Les règles qui régissent le développement de la colonie dans les différents modes de simulation sont implémentées dans le module *simulation*. C'est également à l'aide des fonctions de ce module que sont initialisées les simulations. Les objets représentant les entités simulées, que nous détaillons dans les parties suivantes, sont stockés dans des listes. Ces dernières peuvent être données en argument à des fonctions de type *un_tour()* qui, lorsqu'elles sont appelées, réalisent une étape de la simulation et font interagir les différents organismes.

Un menu permet d'accéder aux différents modes de simulation qui offrent la possibilité de faire évoluer une ou deux colonies. Nous avons également créé un mode jeu pour mettre en valeur l'aspect ludique de nos simulations. Toutes ces fonctionnalités sont implémentées dans le module *affichage* qui permet de suivre graphiquement l'avancement de la simulation. Une schématisation de l'application est donnée en Figure 1 :

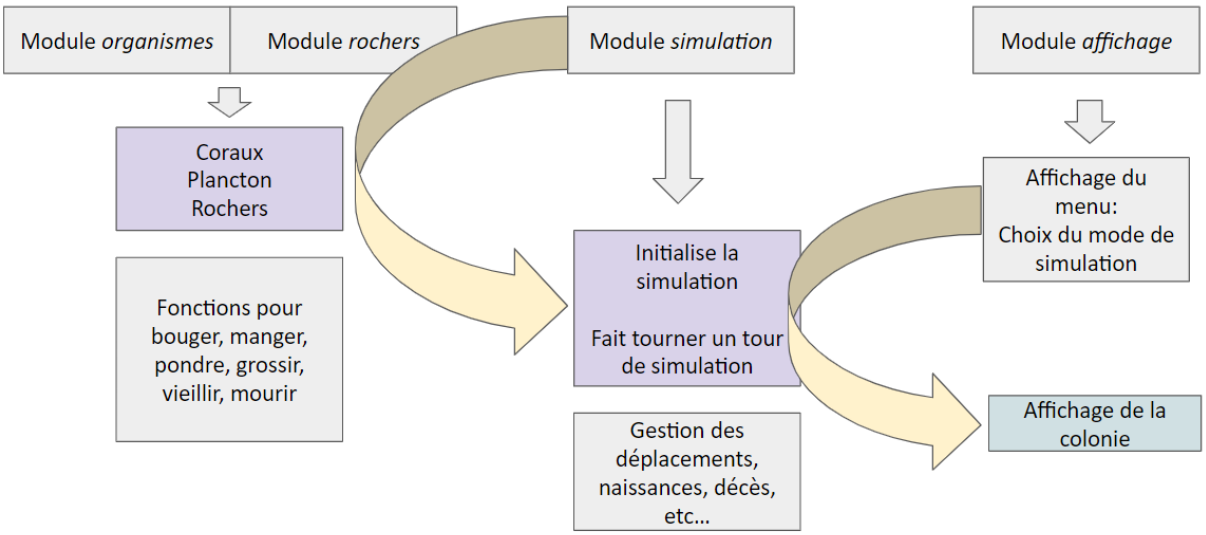

Figure 1 Schématisation du fonctionnement du programme

Notons dès maintenant que les différents objets qui interviennent dans la simulation sont caractérisés par une position (*x*, *y*) au sein de l'aquarium. Nous avons naturellement choisi de mesurer la distance qui sépare ces objets avec la norme euclidienne.

4 – La superclasse Organisme

4.1 – Principes généraux

La superclasse *Organisme*, dont le diagramme de classe est présenté en Figure 2, permet de générer tous les êtres vivants qui interviennent dans la simulation. Ces êtres vivants sont modélisés par des disques de rayons variables et centrés sur un point (*x*, *y*) de l'aquarium. Selon les cas, ils peuvent être fixes ou mobiles (certains coraux sont en effet fixés).

Un setter permet de contrôler la position des organismes pour s'assurer qu'ils ne quittent pas l'aquarium. Pour nos simulations, par exemple, nous avons choisi un aquarium de taille 1000 x 600 (unités arbitraires). Si la position d'un quelconque être vivant doit prendre un jeu de coordonnées extérieur à l'aquarium, le setter lui fait prendre automatiquement des cordonnées situées en bordure. Cela évite aux organismes de sortir du visuel de la simulation et modélise fidèlement le concept d'aquarium étanche.

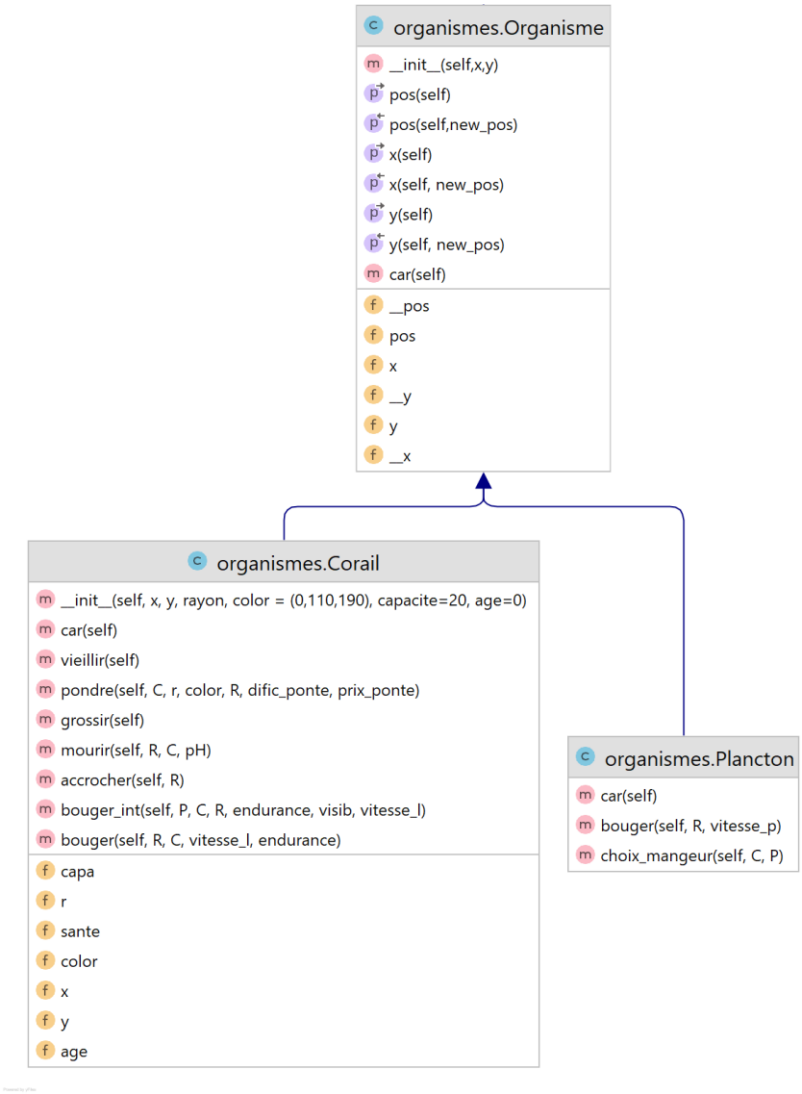

Figure 2 Diagramme de la superclasse Organisme

4.2 – La sous-classe Corail

4.2.1 – Description

La sous-classe *Corail*, qui hérite de la superclasse *Organisme*, permet, lorsqu'elle est instanciée, de générer une larve à un emplacement aléatoire de l'aquarium. Cette larve est susceptible de se déplacer dans l'eau selon des stratégies de déplacement déterminées pour enfin se fixer à un rocher et devenir ainsi un corail adulte. Elle pourra alors vieillir, grossir, manger le plancton qui passe à proximité, pondre une nouvelle larve et finalement mourir pour se transformer en calcaire. Le calcaire en question constituera un nouveau site de fixation pour les autres larves.

Cette classe fait appel à des variables d'instance supplémentaires par rapport à la classe mère comme l'âge, la capacité de stockage en nourriture, la couleur ou encore la santé. L'âge peut varier entre 0 et 5 : 0 correspondant au stade larvaire, les âges 1, 2, 3 et 4 correspondants quant à eux à un âge adulte fixé. L'âge 5 correspond enfin à la mort du corail et marque sa transformation en calcaire.

Nous expliciterons dans la section suivante les méthodes d'instance de la classe *Corail* qui figurent parmi les plus importantes.

4.2.2 – Les méthodes principales de la classe Corail

La méthode *bouger()* permet de changer la position d'une larve en modélisant un mouvement brownien. Elle s'assure que la larve ne pénètre pas dans un rocher et ne se superpose pas avec d'autres coraux vivants. La vitesse de la larve est paramétrable.

La méthode *accrocher()* permet à une larve de se fixer à un rocher ou du calcaire lorsqu'elle passe à proximité.

La méthode *pondre()* permet à un corail adulte de pondre, sous certaines conditions, une larve dans l'eau. Comme les larves initialement présentes, elle peut suivre sa route et se fixer sur un autre rocher ou un corail mort.

La méthode *mourir()* sert à éliminer les coraux trop âgés ou privés de nourriture par leurs voisins. Ces derniers sont alors remplacés par des rochers jaunes de mêmes rayons qui représentent le calcaire.

La méthode *bouger_int()* est une nouvelle méthode de déplacement de la classe *Corail* que nous avons développée dans la seconde partie du projet. Elle est plus évoluée que son ancêtre : la méthode *bouger()*. Au lieu de se déplacer suivant un mouvement brownien, la larve évalue son environnement et se déplace dans la direction la plus densément peuplée en plancton. Cela lui permet de survivre plus longtemps dans une zone pauvre en nourriture ou inhomogène. Si toutes les directions sont équivalentes, alors la larve se déplace avec la méthode *bouger()* classique.

4.3 – La classe plancton

4.3.1 – Description

La classe *Plancton*, qui hérite également de la superclasse *Organisme*, permet de générer un objet plancton dont la seule fonction sera de se déplacer aléatoirement dans l'aquarium pour nourrir les coraux.

Cette classe contient les données nécessaires à l'apparition et au mouvement du plancton dans notre simulation. Son constructeur n'est pas explicité, et reprend donc par défaut les variables d'instances de position de la classe mère.

4.3.2 – Méthode principale

Comme pour les larves, nous avons animé les planctons d'un mouvement brownien via la méthode *bouger().* Le fonctionnement est similaire à celui de la méthode *bouger()* de la classe *Corail* mais moins contraignant en termes de zones accessibles : la méthode interdit simplement au plancton de pénétrer dans un rocher. Lors d'un appel de la méthode, le plancton va voir ses deux coordonnées augmenter ou diminuer selon une loi aléatoire, et pourra aller plus loin selon sa vitesse, qui est paramétrable.

La méthode *choix_mangeur()* mesure la distance entre le plancton et les coraux de l'aquarium. Elle permet de déterminer le corail le plus proche en mesure de manger le plancton : ce dernier est désigné comme mangeur s'il est suffisamment près.

5 – Les autres classes

5.1- La classe Rocher

Les rochers constituent l'environnement initial de la simulation. Ils sont voués à être recouverts par le corail au fil des tours et à augmenter en volume au fil de la création du calcaire.

Les rochers sont instanciés comme étant des disques de position (*x, y),* de rayon *r* et de couleur grise par défaut. Les coraux morts sont remplacés par des rochers jaunes pour représenter le phénomène de calcification.

Cette classe ne possède pas de méthode d'instance particulièrement intéressante (la méthode *car* () sert simplement à renvoyer la chaîne de caractère *'R').* Le diagramme de la classe Rocher est présenté sur la figure 3.

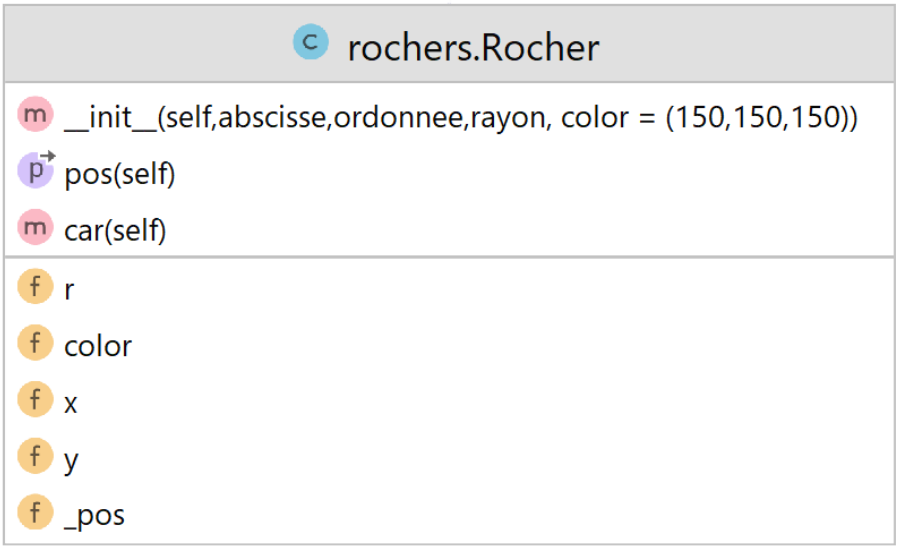

Figure 3 Diagramme de la classe Rocher

5.2 – La classe Bouton

La classe *Bouton* a été créée spécialement pour le module *affichage*. Elle permet de créer, comme son nom l'indique, les différents boutons utilisés dans notre IHM. Son diagramme de classe est présenté en figure 4. Grâce à cette classe, les images des boutons peuvent s'afficher.

Ses variables d'instances sont une image, ses coordonnées, et un rectangle contenant l'image. Est également définie la variable d'instance click, servant à éviter de cliquer en continu.

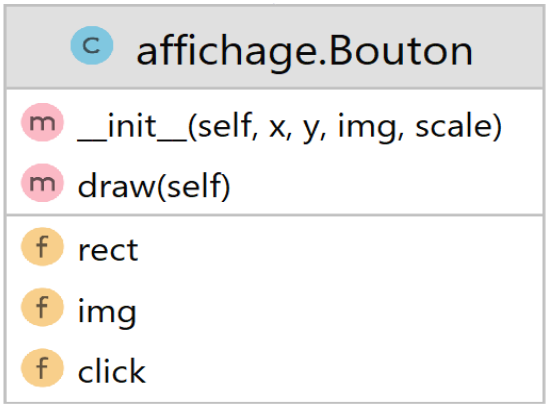

Figure 4 Diagramme de la classe Bouton

6 – Fonctions et paramètres de simulation

6.1 – Les fonctions de création, d'initialisation et de simulation

Plusieurs fonctions servent à générer aléatoirement les différents objets dans l'espace de la simulation. C'est le cas des fonctions *creation_corail()*, *creation_rochers()*, *creation_plancton()*, *apparition_plancton()* ou encore *apparition_corail()* implémentées dans les modules *organismes* et *rochers*. Ayant toutes des fonctionnements similaires, nous n'allons en décrire qu'une seule.

La fonction *creation_rochers()*, par exemple, prend comme paramètres un nombre entier *n*, les dimensions *L x h* du rectangle sur lequel se déroule la simulation, un rayon maximal *R_max* ainsi qu'un rayon minimal *R_min*. Cette fonction crée *n* objets de type *Rocher*, de rayons aléatoires et uniformément répartis dans l'intervalle [*R_min, R_max]* et de coordonnées aléatoires $(x, y) \leq (L, h)$ pour l'ordre lexicographique.

Les fonctions *initialisation_solo()*, *initialisation_multi()* et *initialisation_jeu2()* du module *simulation* dont appel à ces fonctions de création pour générer les situations de départ dans les différents modes de simulation.

Les fonctions de simulation constituent le cœur de notre algorithme. Elles permettent de faire évoluer la colonie de l'étape n à l'étape n+1 en ajoutant de la nourriture dans l'aquarium, en faisant manger, bouger, pondre, vieillir et mourir les différents organismes. Il existe différentes fonctions pour effectuer un tour de simulation : *un_tour_solo()*, *un_tour_multi()* et *un_tour_jeu2()*.

6.2 – Les paramètres de simulation

Nous avons défini de nombreux paramètres de simulation afin d'obtenir une grande variété de scénarii. Tous ces paramètres sont stockés dans un fichier texte nommé *parametres*, modifiable par l'utilisateur.

Parmi ces variables, on peut évoquer celles qui concernent l'environnement comme le *pH* de l'eau (qui peut entraver la calcification des coraux s'il est trop éloigné de 7), la taille et le nombre des rochers, l'apport en nouvelles larves…

Nous avons également défini des variables modélisant les propriétés physiologiques des larves comme l'endurance qui modélise la capacité des coraux à survivre en absence de nourriture, la visibilité qui modélise la capacité des larves à estimer la meilleure direction à suivre dans la méthode *bouger_int* ou encore la difficulté de ponte *dific_ponte* modélisant la capacité des coraux à pondre de nouvelles larves

Nous avons défini d'autres paramètres de simulation dont certains ont été dédoublés afin de rendre le mode affrontement plus riche. On peut ainsi voir s'affronter deux colonies avec des visibilités ou des endurances différentes.

7 – Tests et figures imposées

7.1 – Tests

Nous avons testé toutes les méthodes d'instance pertinentes et détecté les exceptions via notre module *test_unitaires*. Grâce au module unittest.py intégré à Python, nous créons une « classe test » pour chaque classe de notre simulation, et vérifions chaque paramètre utilisé par chaque méthode. Que ce soit l'appartenance d'une coordonnée à un certain intervalle, ou l'égalité entre variables d'instance, tous les cas sont passés au crible afin d'éviter toute erreur de code dans les fondations de notre projet.

Prenons l'exemple de la méthode *test_bouger()* de la classe *TestCorail* : cette méthode a pour but de vérifier le bon fonctionnement de la méthode *bouger()* pour les coraux : il y a plusieurs paramètres à tester. Le premier *self.assertTrue()* sert à valider la vitesse de déplacement. En effet, le test est considéré comme valide seulement si la larve reste, après mouvement, dans l'intervalle maximal de déplacement.

Le deuxième *self.assertEqual()* permet de s'assurer que les larves perdent bien des points de santé si elles se déplacent sans plancton autour d'elles. En effet, nous avions au préalable stocké la santé des larves avant et après mouvement dans deux listes distinctes afin de pouvoir les comparer.

Toutes les méthodes de test fonctionnent de la même manière, il faut uniquement adapter les paramètres à vérifier selon chaque cas.

7.2 – Choix des figures imposées

Les figures imposées que nous avons choisies sont recensées dans la table suivante :

Figures imposées	**Zone d'implémentation**
Factorisation du code	-
Documentation et commentaire du code	-
Tests unitaires	Module *tests_unitaires*
Création d'un type d'objet	Modules *organismes*, *rochers* et *affichage*
Héritage entre deux type créés	Superclasse Organisme
Lecture/écriture de fichiers	Module *simulation*
Modes supplémentaires	*mode_duo*, *mode_jeu_deux_joueurs*

8 – Premiers résultats avec Matplotlib

Au cours du développement de notre programme et avant la création de l'IHM avec le module pygame, nous avons visualisé les résultats de nos simulations avec Matplotlib. Nous avons effectué mille tours de notre prototype de simulation, voici les résultats obtenus (figures 5 et 6). Les larves sont représentées par des disques bleus, le plancton par des disques verts, les rochers par des disques gris, le calcaire par des disques jaunes et les coraux fixés par des disques marrons. Les larves pondues sont quant à elles représentées par des disques violets.

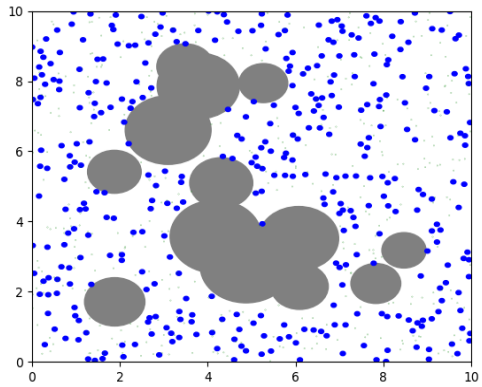

Figure 5 Situation initiale

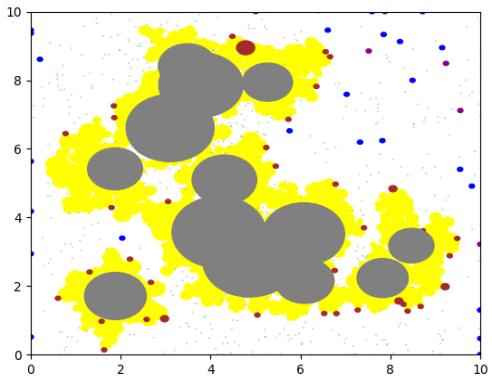

Figure 6 Résultats après 1000 étapes de simulation

Cette première approche nous a permis de valider notre stratégie mais ne permettait pas de de visualiser en temps réel la simulation. Nous ne pouvions donc pas apprécier correctement l'influence des différents paramètres sur le déroulement de la simulation.

9 – Interface graphique et application finale

9.1 – Affichage des simulations avec Pygame

Dans la seconde partie du projet, nous avons développé une interface graphique avec le module pygame. Celle-ci lance un menu permettant de choisir le mode de simulation désiré.

L'aquarium étant de taille (*dimx*, *dimy*). L'affichage de la colonie se fait en temps réel dans une fenêtre de dimension *dimx* pixels par *dimy* pixels. Un compteur affiche alors le nombre de tours en bas à gauche de la fenêtre.

Selon les paramètres choisis, la colonie peut se développer suivant des géométries différentes. Si la ponte est coûteuse en points de vie ou si l'endurance des coraux est faible : les coraux mourront tôt et donneront des structures calcifiées plus fines. Un aperçu de l'IHM est donné en Figure 7, les coraux de première génération sont en bleu tandis que ceux qui ont été pondus par sont représentés en violet :

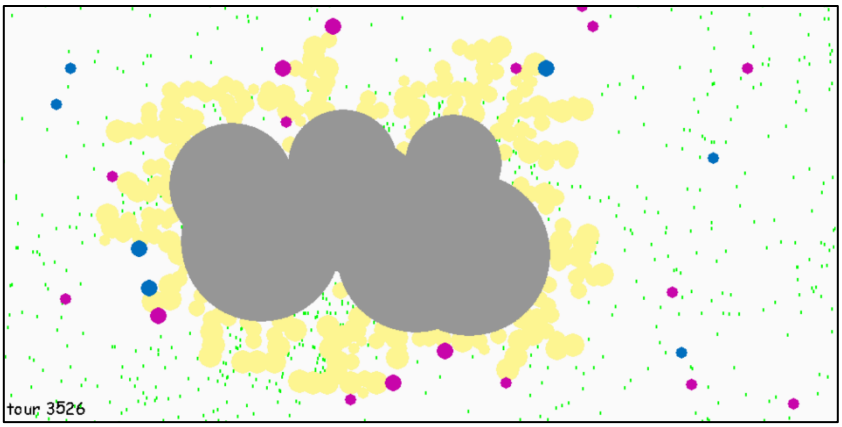

Figure 7 Image d'une colonie simulée en temps réel

L'utilisateur peut stopper la simulation à tout moment ou la mettre en pause. Il peut également interagir avec celle-ci dans certains modes en ajoutant des larves ou du plancton avec les clics droit et gauche de la souris. Il peut également relancer la simulation.

9.2 – Tracés de graphiques

Nous souhaitions visualiser le résultat des simulations sous forme graphique en traçant avec Matplotlib l'évolutions des populations de l'aquarium au cours du temps. Ceci nous a permis de caractériser différents types d'écosystèmes en fonction des paramètres de simulation. Un exemple de colonie stable est donné en Figures 8 :

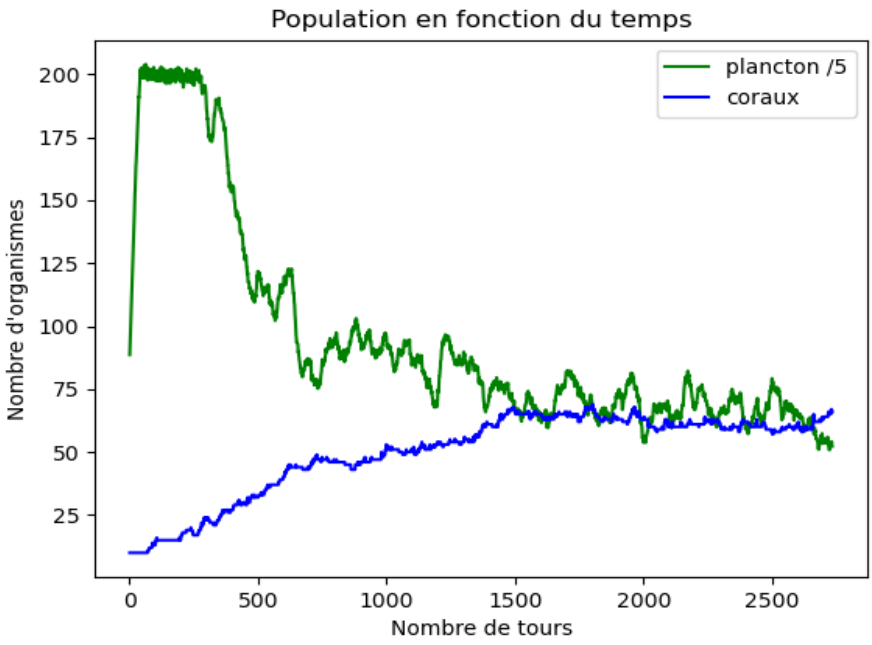

Figure 8 Convergence vers une situation d'équilibre

Dans la première phase, la population corallienne augmente linéairement et la quantité de nourriture chute. Au bout d'un certain temps, on voit apparaître un régime stable dans lequel les populations n'évoluent plus significativement au fil de la simulation. Il y a autant de naissances que de morts.

Les équilibres sont parfois détruits par l'apparition de poches de plancton qui se forment au sein de la gangue de calcaire. Le plancton s'y accumule, ce qui provoque une famine à la surface de la gangue. L'émergence de cette situation est liée à certains paramètres de simulation comme l'endurance et la difficulté de ponte qui influent sur la forme de la structure calcifiée. Cette situation est représentée sur les figures 9 et 10 :

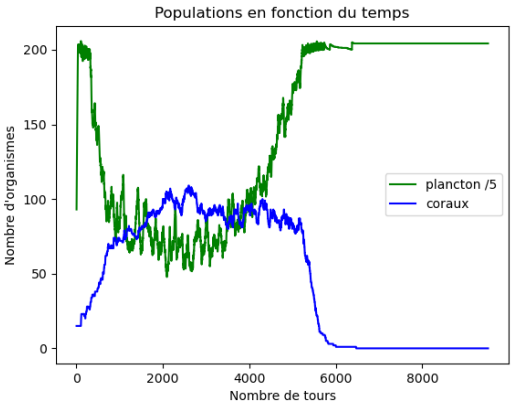

Figure 9 Décès d'une colonie suite à la formation de poches

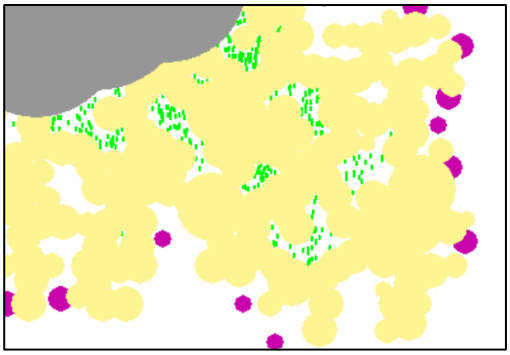

Figure 10 Apparitions de poches de plancton

Nous avons pu jouer avec les paramètres de simulations pour tenter d'observer les différentes possibilités de développement. Les graphes ont constitué un outil particulièrement intéressant pour classer les différents résultats.

10 – Conclusion

Grâce à ce projet, nous avons pu nous initier aux simulations biologiques. Notre objectif était de voir apparaitre des scénarii variés, ce qui a été en partie réussi : nous n'avons pas obtenu de résultats particulièrement surprenant mais l'obtention d'équilibres a été très satisfaisante.

Nous pourrions continuer le projet en introduisant de nouvelles fonctionnalités comme un menu graphique pour régler les paramètres avant le début de chaque simulation ou un algorithme permettant d'effectuer une succession de simulations et d'en faire un bilan par le biais de statistiques et de graphes. Le grand nombre de paramètres et la durée des simulations rendraient cependant cette étude statistique longue et difficile.