Ensta Bretagne
2, rue François Verny
29806 Brest cedex
France
Tel +33 (0)2 98 34 88 00
www.ensta-bretagne.fr

Projet Informatique

UV 2.1

17 mars 2016

Simulation d'un écosystème
Partie 1

Simon Allenic & Pierre Erdeven
simon.allenic@ensta-bretagne.org
pierre.erdeven@ensta-bretagne.org

Table des matières

Introduction

Le programme présenté dans ce rapport simule un écosystème. Les animaux sont divisés en plusieurs espèces telles que les herbivores, les prédateurs et les charognards. Ils évoluent dans un environnement assimilé à une carte. Sur cette dernière se trouve, en plus des animaux, les ressources dont ils ont besoin pour vivre. Toutes les espèces ont besoin d'eau. Les herbivores mangent de l'herbe. Les prédateurs mangent les herbivores. Les charognards mangent les cadavres des animaux morts. Leur vie est régie par jours composés de plusieurs cycles. Chaque jour l'animal vieillit.

L'intérêt d'un tel programme réside dans l'observation de l'évolution d'une espèce en fonction de paramètres définis au préalable. Comme les charognards mangent les animaux morts, seul l'évolution des prédateurs et des herbivores seront liées. Afin de vérifier la cohérence du programme de simulation avec la réalité, il pourra être comparé aux équations de Lotka - Volterra qui sont désignées aussi sous le terme de « modèle proie-prédateur ».

Le présent rapport s'intéresse à la partie algorithmique du programme. Il est divisé en quatre chapitres. Le premier traite des différentes pistes envisagées et des hypothèses réductrices choisies. Le deuxième concerne la description générale du programme. Le troisième montre ses mécanismes fondamentaux. Pour finir, le quatrième présente les différents tests effectués, les perspectives et améliorations à apporter à cette simulation.

1 Génèse du programme

1.1 Pistes envisagées

Pour répondre à la problématique, deux pistes ont été envisagées. La deuxième a rapidement dominé la première par sa clarté et sa facilité de mise en œuvre.

La première piste proposée démarrait d'une classe abstraite nommée *Animal* dans laquelle toutes les propriétés communes à tous les animaux étaient définies. Les classes *Herbivore*, *Predateur* et *Charognard* héritaient alors de la classe abstraite *Animal*. Le séquencement de ces sous-classes a posé problème. En effet, les ressources ont été traitées au cas par cas dans les classes représentantes les espèces, et la méthode permettant de changer leur comportement faisait appel à d'autres classes elles aussi définies au cas par cas. Par exemple, une classe nommée *Normal_Herbivore* héritait de la classe *Animal*, ce qui déjà n'était pas très cohérent, et permettait de définir le comportement normal d'un herbivore. Une autre classe nommée *Faim_Herbivore* qui héritait aussi de la classe abstraite définissait le comportement d'un herbivore affamé. On se rend alors très vite compte du nombre de classes nécessaires pour coder les différents comportements de tous les animaux. Cette piste n'était pas envisageable du point de vue de l'optimisation du programme.

La deuxième piste a tout de même été inspirée de la première dans ses premières lignes. En d'autres termes, la super-classe *Animal* a été conservée ainsi que ses trois sous-classes. Le comportement de l'espèce a quand à lui été défini dans une seule méthode ce qui en simplifie grandement son exécution.

1.2 Hypothèses réductrices

Pour simuler l'écosystème, nous avons émis les hypothèses réductrices suivantes :

- L'eau est une ressource illimitée.
- Tous les végétaux comestibles sont réduits à de l'herbe.
- L'animal ne dort pas.
- La reproduction n'est pas prise en compte pour l'instant.
- Seuls deux facteurs agissent sur la vie de l'animal : la vieillesse et le manque d'eau ou de nourriture. La maladie n'intervient pas.
- Lors de l'attaque d'un prédateur l'herbivore tentera de fuir. S'il n'y parvient pas, il sera forcément tué. La blessure n'existe pas.
- Les prédateurs ne s'entretuent pas pour déterminer le dominant. Il n'y pas de hiérarchie dans les espèces.

2 Description générale

Avant de décrire chacune des classes du programme, attardons nous sur son fonctionnement général. Pour une meilleure lisibilité le programme est divisé en sept fichiers. La classe qui pilote l'écosystème est *Ecosys*. C'est elle qui contient le *main* et qui permet de gérer le nombre des animaux présents sur la carte. Elle commande le nombre de cycles et de jours. En fin de journée, elle vérifie si les quotas en nourriture et en eau des animaux sont atteints. Elle fait appel à la méthode *calcVie* de la classe *Animal* qui calcule la perte de vie de l'animal. L'action de ce dernier (déplacements, boire, manger,...) sera effectué à chaque cycle. Les animaux sont définis par héritage de la classe *Animal*. La carte est créée dans un fichier texte puis chargée dans la classe *ResMap*. Chaque ressource possède un identifiant permettant de faire le lien entre ce fichier texte et la classe abstraite *Ressource* d'où elles sont gérées. Pour finir, la représentation géométrique des animaux et de leur champ de vision est générée dans le fichier `object.py`. La connaissance de ces paramètres permet de développer la gestion des collisions avec les autres animaux, les bords de la carte ou encore les différentes ressources.

Le diagramme de classe de la Figure 1 entre un peu plus dans le détail de l'héritage et du polymorphisme des différentes classes du programme. Par la suite, nous ne détaillerons pas toutes les classes et méthodes le composant mais seulement les fondamentales.

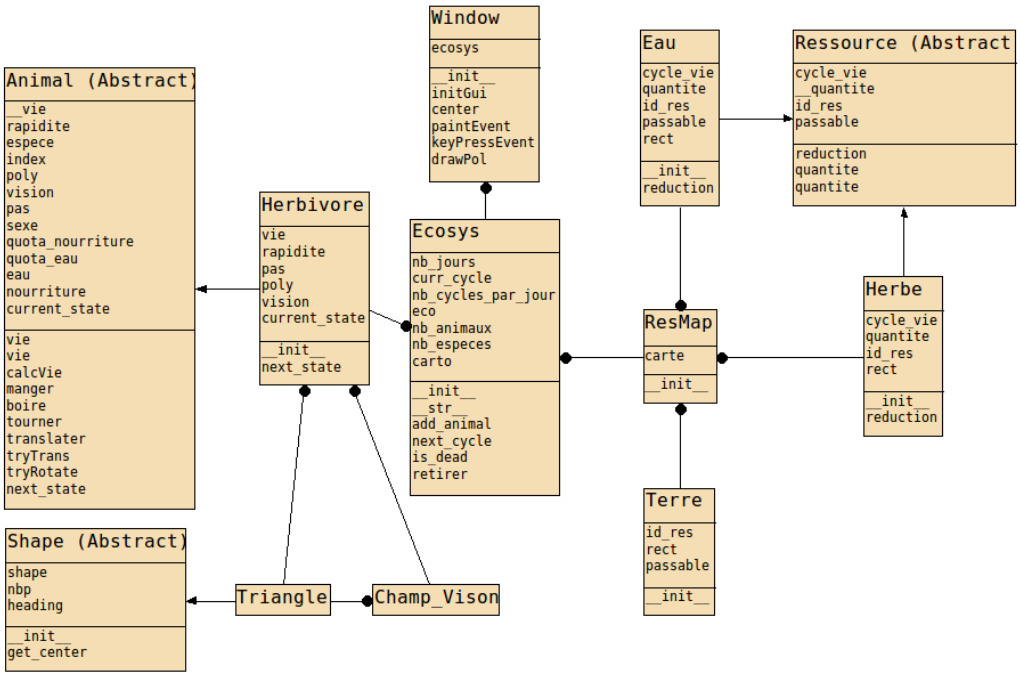

FIGURE 1 – Diagramme de classe

3 Mécanismes fondamentaux

3.1 Classe *Ecosys*

Cette classe est le noyau du programme. Elle gère les animaux et leur vie. Elle contient des méthodes qui ajoutent ou retirent un animal et vérifient si celui-ci est mort.

3.1.1 Méthode *add_ animal*

La méthode *add_ animal* permet d'ajouter un animal dans l'écosystème. Celui-ci est ajoutée dans une liste correspondant à son espèce. L'écosystème est organisé par un dictionnaire dont les clés sont les espèces et les valeurs sont leurs animaux.

3.1.2 Méthode *next_ cycle*

La méthode *next_ cycle* gère la vie de l'animal. Tous les animaux de l'écosystème évolueront en même temps, rythmés par des cycles. À la fin de la journée, la méthode remet à 0 les réserves en nourriture et en eau de l'animal.

3.2 Classe abstraite *Animal*

Cette classe contient les variables communes à tous les animaux. Celles-ci sont stockées sous forme de variable de classe. Les méthodes communes aux animaux y sont définies. On y trouve par exemple les méthodes *manger*, *boire*, *translater* et *tourner*. Le comportement propre à chaque espèce se retrouvera dans la méthode *next_ state* qui est elle-même définie dans les sous-classes de *Animal*.

3.2.1 Méthode *calcVie*

La méthode *calcVie* gère la réduction de la vie de l'animal. Elle est appelée par la méthode *next_ cycle* de la classe *Ecosys* lorsqu'un jour s'est écoulé. C'est la raison pour laquelle elle prend en paramètre un booléen indiquant si oui ou non le jour est terminé. Elle retire alors une unité de vie à la variable *vie* pour simuler la vieillesse de l'animal. Elle lui retire également autant d'unité de vie qu'il manque entre l'eau qu'il a bu dans la journée et son quota nécessaire à sa survie. Cette opération traduit simplement le fait que si un animal ne boit pas, au bout d'un certain temps qui dépend de son âge, celui-ci meurt. Le même raisonnement est appliqué pour la nourriture.

3.2.2 Méthode *tryTrans*

La méthode *tryTrans* permet de vérifier si l'animal peut avancer ou reculer. Elle utilise pour cela les méthodes programmées dans le fichier `collisions.py` qui véri-

fient les collisions avec les bords de la carte et les autres polygones présents. Elle est appelée dans la méthode *translater*. Le principe est d'utiliser une figure virtuelle de même forme que celle de l'espèce considérée. Il suffit ensuite de vérifier s'il y a collision avec des éléments de la carte lors de la translation de ce polygone. La méthode *tryRotate* est basée sur la même idée.

3.3 Classe *Herbivore*

La classe *Herbivore* est encore en cours d'élaboration. Dans le constructeur, on y définit les caractéristiques de l'espèce. La méthode *next_ state* a pour objectif de suivre le comportement défini par l'automate de la figure 2.

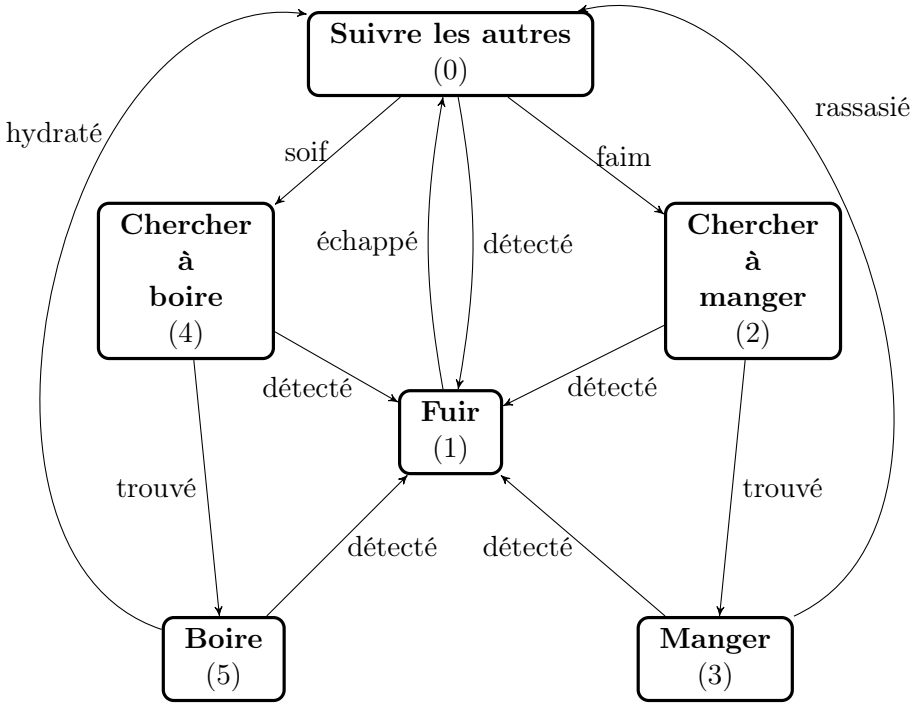

FIGURE 2 – Diagramme d'état d'un herbivore

3.4 Ressources

Les ressources sont gérées avec quatre classes. De la même manière que pour la gestion des espèces, les ressources ont à leur tête une classe abstraite appelée *Ressource*. On y définit leurs caractéristiques communes comme leur quantité, leur cycle de vie et un identifiant. On indique également si l'animal peut marcher sur cette ressource ou non par un booléen. Cette variable se révèle importante pour l'eau qui, dans la plupart des cas, est infranchissable. En ce qui concerne leur quantité, l'eau

est par hypothèse illimitée. L'herbe diminue en fonction du nombre de bouchées de l'animal.

Une remarque importante concerne la gestion des cadavres des animaux. Elle sera traitée dans cette partie car assimilée à une source de nourriture pour les charognards. La contrainte de la figure imposée sera remplie par la mise en place d'une liste simplement chaînée. Une case initialement herbe ou terre sera remplacée par une qui représentera le cadavre de l'animal en question. Et, redeviendra terre ou herbe après qu'un charognard l'ait mangé. La figure 3 permet de mieux visualiser ce concept.

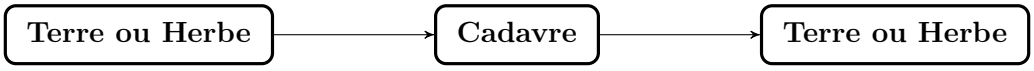

FIGURE 3 – Liste simplement chaînée pour la gestion des cadavres

3.5 La représentation en mémoire de l'écosystème

Deux grands types de représentation sont utilisés dans le programme. Les ressources sont représentées par une case au sein d'une matrice tandis que les animaux sont représentés par un polygone, c'est à dire une liste de coordonnées.

Les ressources (*Eau* et *Herbe*) ainsi que *Terre* sont toutes appelées et stockées au sein de la classe *ResMap*, dans un tableau à deux dimensions. La carte des ressources est chargée depuis un fichier dans le constructeur de la classe *ResMap*. Dans ce fichier, on a les équivalences suivantes:

— 0 = Terre
— 1 = Herbe
— 2 = Eau

Lors de la création d'un Herbivore, 2 polygones sont créés. Un premier qui représente l'animal et un deuxième qui représente son champ de vision. Ces polygones sont créés en faisant appel à la classe *Triangle* qui hérite de la classe abstraite *Shape*. Le champ de vision est attaché au polygone représentant *Herbivore*, il se déplace de la même manière que lui. La mise à jour de la position du champ de vision est réalisée par la méthode *update* de la classe *Champ_ Vision*.

Un polygone n'est qu'une simple liste de coordonnées, il suffit donc d'effectuer le produit de chaque vecteur de coordonnées et d'une matrice de translation ou de rotation pour faire bouger le triangle. La principale difficulté se situe au niveau de la gestion des collisions. Deux polygones ne peuvent pas se chevaucher et un herbivore ne peut pas marcher dans l'eau par exemple. Pour éviter de lourds calculs inutiles plusieurs niveaux de gestion de collisions ont été implémentés. Les fonctions détaillées ci-dessous sont toutes issues du fichier `collisions.py`.

Pour savoir sur quelle ressource se situe un point aux coordonnées quelconques, il suffit de connaître la place occupée par la ressource dans la matrice *carte* de la classe *Resmap*.

Listing 1 – test

```
def current_case(x_h, y_h):
        '''
        Retourne la place (i,j) qu'occupe la ressource sur
            lequel est le point (x_h, y_h) dans la carte.
        Il suffit de diviser les coordonnées (x_h,y_h) par la
            longueur ou la largeur du carré qu'occupe la
            ressource sur la carte.
        '''
    return (int(y_h//constantes.carre_res[1]), int(x_h//
        constantes.carre_res[0]))
```

La gestion de collisions entre polygones (donc entre animaux) est plus complexe et se fait en deux temps. D'abord il y a une première vérification avec la fonction *check_rect_coll* qui prend en paramètre le polygone que l'on cherche à déplacer et la liste des autres polygones présents sur la carte. Pour chacun des polygones, la fonction extrait le meilleur carré qui l'englobe puis teste les collisions entre les carrés. En effet, tester des collisions entre carrés est très peu calculatoire.

S'il y a collision entre deux carrés, et seulement dans ce cas, la fonction *check_glob_coll* est appelée pour tester très précisément la collision entre les deux triangles. La méthode est simple. Il faut placer un point I aléatoirement en dehors de l'écran. Ensuite on définit un rôle pour chacun des deux polygones. L'un des deux joue le rôle du polygone passif et l'autre le rôle du polygone actif. Pour chaque point P du polygone actif, on regarde ensuite combien de segments $[AB]$ du polygone passif sont traversés par le segment $[IP]$. Si ce nombre est impair, il y a collision, s'il est pair il n'y pas collision. On répète ensuite l'opération en inversant les rôles des polygones actif et passif.

La fonction *intersectSegment* se charge de tester s'il y a intersection entre deux segments $[AB]$ et $[IP]$. Pour cela, la fonction vérifie dans un premier temps si l'on ne se situe pas dans un cas limite, en calculant $D = det(\overrightarrow{AB}, \overrightarrow{IP})$. Si le déterminant vaut 0, alors les deux droites peuvent être parallèles ou confondues, on relance donc la fonction avec un autre point I. Sinon, on écrit les équations paramétriques des deux segments:

$$J = A + u\overrightarrow{AB}$$

$$K = I + t\overrightarrow{IP}$$

On résout le système pour obtenir t et u. Si les deux sont compris entre 0 inclus et 1 exclu, alors il y a bien intersection entre les deux segments.

4 Tests, perspectives et améliorations

Le test est réalisé dans le fichier `ecosys.py`. Il ajoute deux herbivores et effectue un cycle. Son exécution retourne :

```
jour 0
espèce mouton (2)
         80 10
         80 10
```

Le nombre de gauche représente sa vie et celui de droite sa réserve de nourriture. On remarque que, au bout d'un cycle, les deux moutons ont bien mangé 10 unités d'herbe correspondant à une bouchée.

La partie algorithmique n'est pas encore terminée à l'heure actuelle. Les perspectives et améliorations seraient donc nombreuses à énumérer. Néanmoins, à court terme, l'objectif est d'étoffer la classe *Herbivore* en créant les comportements plus avancés tels que la fuite ou encore la détection des prédateurs. À moyen terme, la création des deux autres classes *Predateur* et *Charognard* deviendra importante. La gestion de la ressource *Cadavre*, la reproduction et la maturité des animaux sera développée à ce moment là également. Pour finir, sur le long terme, le développement de l'interface graphique afin de visualiser l'évolution de l'écosystème.

Dans notre cas, il est difficile de décorréler complètement la partie graphique de la partie algorithmique. En effet, gérer un système de collisions sans développer une interface graphique pour tester le bon déroulement du programme n'est pas une chose aisée. Pour accéder à un test graphique du système de collisions, il suffit d'exécuter le fichier `window.py`. Il est alors possible de contrôler un herbivore au clavier et de le faire se déplacer sur la carte.

Conclusion