Modélisation de Biosystèmes dynamiques

Caroline HULOT

Avril 2025

Table des matières

1 Introduction **2**

2 Description de l'algorithme 2D **2**
2.1 Boids 2
2.2 Prédateur 3
2.3 Paramètres modifiables 3

3 Description de l'algorithme 3D **4**
3.1 Boids 4
3.2 Prédateur 5
3.3 Affichage 6

4 Réflexions **7**

5 Propositions d'amélioration **7**

6 Conclusion **8**

1 Introduction

Dans ce TP, j'ai simulé un comportement de boids en utilisant **Python** et **Pygame**. Chaque boid suit trois règles principales : séparation, alignement et cohésion. De plus, deux obstacles fixes et un prédateur mobile sont ajoutés pour complexifier la simulation.

2 Description de l'algorithme 2D

2.1 Boids

Chaque boid applique les comportements suivants à chaque étape :
— **Séparation** : éviter les boids trop proches.
— **Alignement** : s'aligner sur la vitesse moyenne des voisins.
— **Cohésion** : se rapprocher du centre de masse des voisins.
— **Évitement d'obstacles** : éviter deux obstacles fixes.
— **Fuite du prédateur** : si le prédateur est proche, s'en éloigner.

Voici une partie du code correspondant :

```
def apply_behaviors(self, boids, predator, obstacles):
    sep = self.separate(boids)
    ali = self.align(boids)
    coh = self.cohesion(boids)
    avoid_pred = self.avoid_predator(predator)
    avoid_obs = sum([self.avoid_obstacle(pos, radius) for pos,
       radius in obstacles], pygame.Vector2(0, 0))

    self.acceleration += sep * 1.5 + ali + coh + avoid_pred * 2 +
       avoid_obs * 2
```

Voici les algorithmes simplifiés utilisés pour chaque comportement :
— **Séparation** : éviter les voisins trop proches

```
Pour chaque voisin proche:
    Calculer vecteur direction opposee
    Somme des directions
Normaliser et ajuster la vitesse
```

— **Alignement** : s'aligner sur la vitesse moyenne

```
Pour chaque voisin proche:
    Additionner vitesses
Calculer moyenne des vitesses
Adapter sa propre vitesse vers cette moyenne
```

— **Cohésion** : rejoindre le centre du groupe

```
Pour chaque voisin proche:
    Additionner positions
Calculer centre de masse
Se diriger vers le centre
```

2.2 Prédateur

Le prédateur peut :

— Se déplacer automatiquement vers le boid le plus proche.

— Être contrôlé manuellement par l'utilisateur avec les flèches du clavier. La touche `Entrée` permet de basculer entre contrôle manuel et automatique.

Extrait de la gestion du prédateur :

```
if self.manual:
    direction = pygame.Vector2(0, 0)
    if keys[pygame.K_UP]: direction.y -= 1
    if keys[pygame.K_DOWN]: direction.y += 1
    if keys[pygame.K_LEFT]: direction.x -= 1
    if keys[pygame.K_RIGHT]: direction.x += 1
    if direction.length() > 0:
        self.velocity = direction.normalize() * MAX_SPEED
else:
    closest_boid = min(boids, key=lambda b: self.position.
       distance_to(b.position))
    self.velocity = (closest_boid.position - self.position).
       normalize() * MAX_SPEED
```

2.3 Paramètres modifiables

Grâce à l'utilisation de la bibliothèque `argparse`, les paramètres suivants sont configurables en ligne de commande :

— `neighbor_radius` : rayon d'influence des boids.

— `max_speed` : vitesse maximale des boids.

— `num_boids` : nombre initial de boids.

Exemple de commande pour lancer le script avec des paramètres :

```
python TPOiseaux2D.py --neighbor_radius 70 --max_speed 6 --
   num_boids 200
```

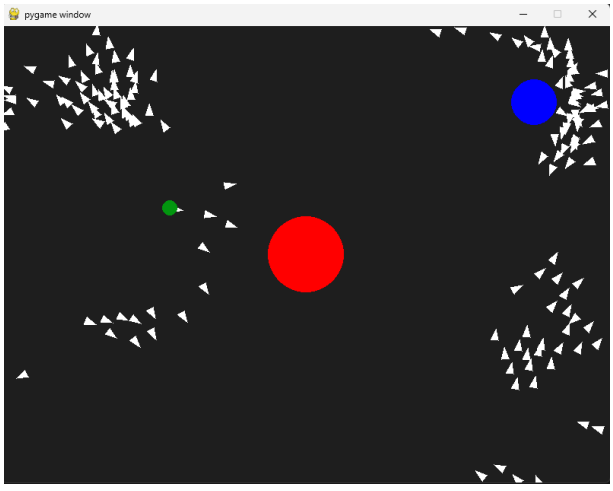

FIGURE 1 – Rendu Simulation 2D

3 Description de l'algorithme 3D

Dans cette version 3D de la simulation, les boids évoluent dans un espace tridimensionnel. Chaque boid applique des comportements similaires à ceux de la simulation 2D, mais dans un environnement à trois dimensions. Le prédateur et les obstacles sont également modélisés en 3D.

3.1 Boids

Les boids suivent les mêmes règles de séparation, d'alignement, de cohésion, d'évitement d'obstacles et de fuite du prédateur, mais dans un espace 3D. Chaque boid a une position et une vitesse dans l'espace tridimensionnel, et les comportements sont ajustés en conséquence.

Voici un extrait de code de l'algorithme pour la mise à jour d'un boid :

```
def update(self, boids, obstacles, predator):
    separation = self.separation(boids)
    alignment = self.alignment(boids)
    cohesion = self.cohesion(boids)
    avoidance = self.avoid_obstacles(obstacles)
    flee = self.flee_predator(predator)

    self.velocity[0] += separation[0] * 1.5 + alignment[0] * 1.0 +
       cohesion[0] * 1.0 + avoidance[0] * 2.0 + flee[0] * 3.0
    self.velocity[1] += separation[1] * 1.5 + alignment[1] * 1.0 +
       cohesion[1] * 1.0 + avoidance[1] * 2.0 + flee[1] * 3.0
    self.velocity[2] += separation[2] * 1.5 + alignment[2] * 1.0 +
       cohesion[2] * 1.0 + avoidance[2] * 2.0 + flee[2] * 3.0

    speed = math.sqrt(sum(v**2 for v in self.velocity))
    max_speed = 1
    if speed > max_speed:
        for i in range(3):
            self.velocity[i] = (self.velocity[i] / speed) *
               max_speed

    for i in range(3):
        self.position[i] += self.velocity[i]

        if self.position[i] > 10:
            self.position[i] = 10
            self.velocity[i] *= -1
        if self.position[i] < -10:
            self.position[i] = -10
            self.velocity[i] *= -1
```

Les comportements du boid sont calculés en fonction de la distance aux voisins et aux obstacles, en utilisant des forces de répulsion et d'attraction. Le boid s'oriente également vers une cible ou s'éloigne du prédateur si nécessaire.

3.2 Prédateur

Le prédateur se déplace automatiquement vers le boid le plus proche dans l'espace 3D. Il peut aussi être contrôlé manuellement via les touches de direction. Le mouvement du prédateur est similaire à celui du boid, avec une mise à jour de sa position en fonction de la distance aux boids et de sa vitesse.

Extrait de la gestion du prédateur :

```
def update(self, boids):
    if not self.manual_control:
        # Trouver le boid le plus proche
        target = min(boids, key=lambda b: math.dist(self.position,
           b.position))
        direction = [target.position[i] - self.position[i] for i in
            range(3)]
        dist = math.sqrt(sum(d**2 for d in direction))
        if dist != 0:
            for i in range(3):
                self.velocity[i] = (direction[i] / dist) * 0.5  #
                   Vitesse vers la cible

        for i in range(3):
            self.position[i] += self.velocity[i]
            if self.position[i] > 10 or self.position[i] < -10:
                self.velocity[i] *= -1
    else:
        # En manuel : deplacement via touches
        keys = pygame.key.get_pressed()
        speed = 0.3
        if keys[K_UP]:
            self.position[1] += speed
        if keys[K_DOWN]:
            self.position[1] -= speed
        if keys[K_LEFT]:
            self.position[0] -= speed
        if keys[K_RIGHT]:
            self.position[0] += speed
        if keys[K_PAGEUP]:
            self.position[2] += speed
        if keys[K_PAGEDOWN]:
            self.position[2] -= speed

        for i in range(3):
            if self.position[i] > 10:
                self.position[i] = 10
            if self.position[i] < -10:
                self.position[i] = -10
```

3.3 Affichage

Le rendu des boids et du prédateur est effectué à l'aide de la bibliothèque OpenGL, avec des formes primitives comme des triangles pour les boids et des sphères pour le prédateur et les obstacles. La scène est rendue en 3D avec une perspective et une gestion des caméras à l'aide des touches directionnelles et des touches spécifiques pour le contrôle de la caméra(z,q,s,d,a,e).

Voici l'extrait de code pour dessiner un boid et un prédateur :

```
def draw(self):
    glPushMatrix()
    glTranslatef(*self.position)
    glScalef(0.2, 0.2, 0.2)
    glBegin(GL_TRIANGLES)
    glColor3f(1, 1, 1)
    glVertex3f(0, 0, 1)
    glVertex3f(-0.5, 0, -1)
    glVertex3f(0.5, 0, -1)
    glEnd()
    glPopMatrix()

def drawPredator(self):
    glPushMatrix()
    glTranslatef(*self.position)
    glColor3f(1, 0, 0)
    quadric = gluNewQuadric()
    gluSphere(quadric, 0.5, 16, 16)
    gluDeleteQuadric(quadric)
    glPopMatrix()
```

Voici l'extrait de code pour dessiner le cube de simulation et la gestion de la camera :

```
def draw_box():
    glColor3f(0.5, 0.5, 0.5)
    glBegin(GL_LINES)
    for x in [-10, 10]:
        for y in [-10, 10]:
            glVertex3f(x, y, -10)
            glVertex3f(x, y, 10)
    for x in [-10, 10]:
        for z in [-10, 10]:
            glVertex3f(x, -10, z)
            glVertex3f(x, 10, z)
    for y in [-10, 10]:
        for z in [-10, 10]:
            glVertex3f(-10, y, z)
            glVertex3f(10, y, z)
    glEnd()

def handle_camera_keys():
    keys = pygame.key.get_pressed()
    if keys[K_z]:
```

```
            glTranslatef(0, 0, 0.2)
        if keys[K_s]:
            glTranslatef(0, 0, -0.2)
        if keys[K_q]:
            glTranslatef(0.2, 0, 0)
        if keys[K_d]:
            glTranslatef(-0.2, 0, 0)
        if keys[K_a]:
            glTranslatef(0, 0.2, 0)
        if keys[K_e]:
            glTranslatef(0, -0.2, 0)
```

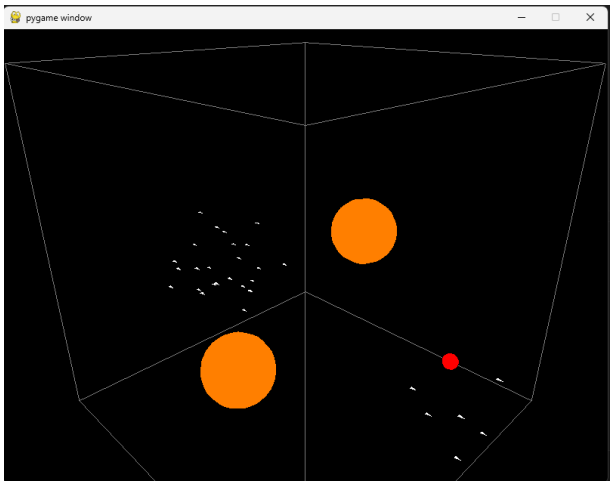

FIGURE 2 – Rendu Simulation 3D

4 Réflexions

L'algorithme est volontairement simple pour rester lisible :
— L'ensemble des comportements est combiné en une seule accélération par boid.
— Les obstacles sont fixes et parfaitement circulaires.
— Le prédateur agit simplement, sans stratégie complexe.
— Le contrôle manuel du prédateur rend la simulation interactive.

Cependant, plusieurs simplifications sont présentes :
— Les vitesses sont constantes (pas d'accélérations variables selon les situations).
— Les collisions entre boids ou avec les obstacles sont évitées de manière basique (forces de répulsion).

5 Propositions d'amélioration

Voici quelques pistes d'amélioration possibles :
— **Ajout d'un cône de vision** : les boids ne devraient percevoir que ce qu'ils voient devant eux.
— **Gestion des collisions physiques** : rebonds réalistes sur les obstacles.

— **Prédateurs plus intelligents** : comportements de chasse plus complexes (stratégies de poursuite, embuscades).
— **Variabilité** : ajouter différentes espèces de boids avec des comportements légèrement différents.
— **Effets visuels** : tracer la trajectoire du prédateur.
— **Obstacles mobiles** : ajouter des obstacles qui bougent.

6 Conclusion

Ce projet est une première approche du comportement collectif. De nombreuses extensions sont possibles pour rendre la simulation plus réaliste et intéressante. Par exemple la version 3D de la simulation ajoute une dimension supplémentaire, permettant de visualiser les comportements des boids dans un espace tridimensionnel.