SCIENCES SORBONNE UNIVERSITÉ

Projet ARE: Comportement collectif et essaimage

Alexandre EGUIZABAL Ziyan TANG Ryadh FOUDAD
Aurelien DANET

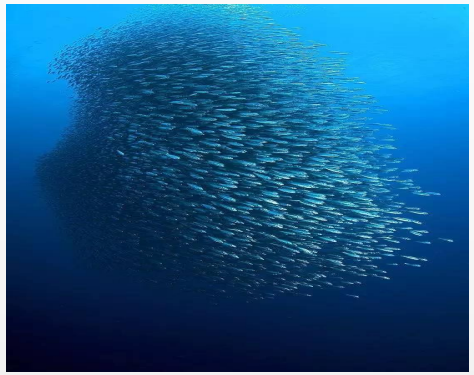
Le 17 avril 2023

Sommaire

Introduction

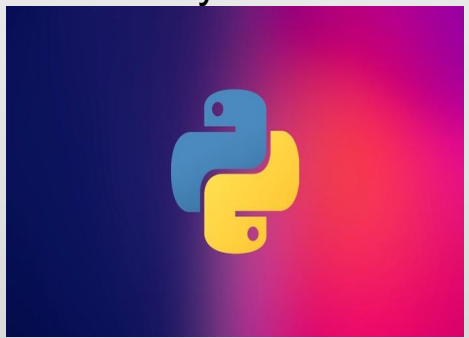
◆Contexte et objectif du projet :

Ce projet utilise la bibliothèque Pygame pour simuler des systèmes multi-intelligents et le comportement de la population afin d'explorer les effets de différents paramètres sur le comportement du système afin de mieux comprendre et prédire des phénomènes complexes dans le monde réel.

◆ Outils et techniques utilisés :

Ce projet utilise le langage de programmation Python et la bibliothèque de développement de jeux Pygame pour créer des simulations de systèmes multi-intelligents et utilise la bibliothèque numpy pour les opérations et les calculs vectoriels, ce qui permet une présentation et une analyse visuelles faciles.



Principes fondamentaux de la simulation

◆ Définition et caractéristiques des systèmes corporels multi-intelligents et du comportement de groupe

Un système multi-agent est composé de plusieurs agents intelligents, chacun étant une entité de prise de décision autonome, travaillant en communiquant et en coopérant les uns avec les autres pour réaliser un comportement collectif **(contrôle distribué)**. Les agents peuvent avoir différentes capacités, comportements et stratégies, ce qui crée une diversité et une hétérogénéité dans le système **(diversité et hétérogénéité)**. Les agents peuvent s'auto-organiser et former des comportements collectifs en interagissant et en coopérant entre eux **(auto-organisation).**

◆ Les comportements et interactions entre les agents et les prédateurs

exclusion, alignement, désalignement, attraction, prédation.

◆ Les paramètres et les réglages de la simulation

Nombre d'agents **(N)** : qui affecte l'interaction et la coopération entre les agents.

Vitesse **(speed)** : qui affecte la vitesse de mouvement des agents et des prédateurs.

Rayon de perception **(collision_radius, alignment_radius, attraction_radius, blind_spot, predator_radius)** : qui affecte la portée de perception des agents et des prédateurs.

```
# Simulation parameters
N = 150
width, height = 500, 500
speed = 20
wiggle = 0.1
collision_radius = 20
alignment_radius = 50
attraction_radius = 100
blind_spot = 120
predator_radius = 0
eating_distance = 5
```

Code d'implémentation de la simulation

◆ Structure et composition du code

nous avons défini deux classes : **"Agent" et "Predator"**, qui décrivent le comportement et l'interaction des agents et des prédateurs. Nous avons également défini plusieurs paramètres et réglages pour contrôler le comportement et les résultats de la simulation. Nous utilisons les bibliothèques **numpy et pygame** pour effectuer des opérations vectorielles, des dessins graphiques, etc.

```python
import pygame
import numpy as np
from pygame.locals import *
```

```python
class Agent:
    def __init__(self, position, heading):
        self.position = position
        self.heading = heading

    def update(self, neighbors, predator):
        # Repulsion
        repulsion = np.zeros(2)
        for other in neighbors:
            if np.linalg.norm(self.position - other.position) < collision_radius:
                repulsion += (self.position - other.position)

        # Alignment
        alignment = np.zeros(2)
        count = 0
        for other in neighbors:
            if np.linalg.norm(self.position - other.position) < alignment_radius:
                angle = np.arccos(np.dot(self.heading, other.heading) / (np.linalg.norm(self.heading) * np.linalg.norm(other.heading)))
                if angle < np.radians(blind_spot):
                    alignment += other.heading
                    count += 1
        if count > 0:
            alignment /= count

        # Attraction
        attraction = np.zeros(2)
        count = 0
        for other in neighbors:
            if np.linalg.norm(self.position - other.position) < attraction_radius:
                attraction += other.position
                count += 1
        if count > 0:
            attraction /= count
            attraction -= self.position

        # Predator
        predator_avoidance = np.zeros(2)
        if np.linalg.norm(self.position - predator.position) < predator_radius:
            predator_avoidance += (self.position - predator.position)
            predator_avoidance *= 2
        self.heading += repulsion + alignment + attraction + predator_avoidance
        random_wiggle = np.random.uniform(-wiggle, wiggle, 2)
        self.heading += random_wiggle
        self.heading /= np.linalg.norm(self.heading)
        self.position += speed * self.heading
        self.check_boundaries()

    def check_boundaries(self):
        if self.position[0] < 0:
            self.position[0] = 0
            self.heading[0] = -self.heading[0]
        elif self.position[0] > width:
            self.position[0] = width
            self.heading[0] = -self.heading[0]

        if self.position[1] < 0:
            self.position[1] = 0
            self.heading[1] = -self.heading[1]
        elif self.position[1] > height:
            self.position[1] = height
            self.heading[1] = -self.heading[1]
```

◆ Fonctions et méthodes utilisées

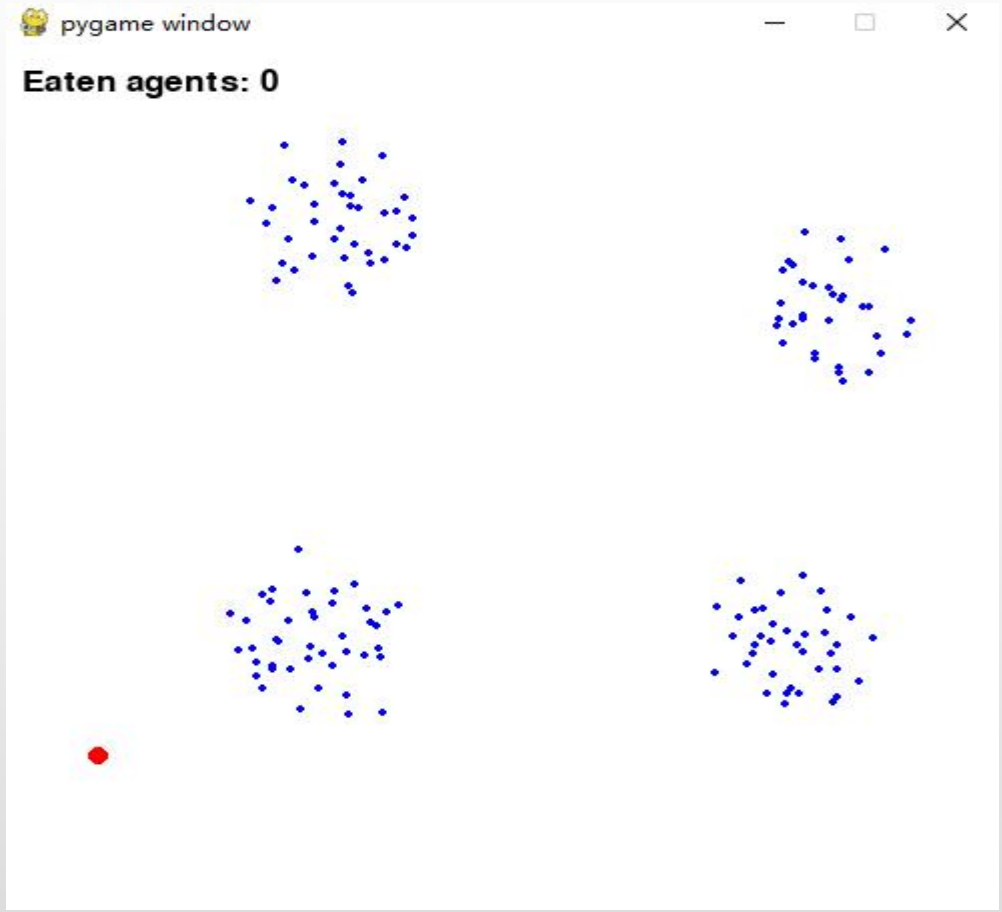
- Ce projet utilise de nombreuses fonctions et méthodes de la bibliothèque numpy et de pygame, notamment :
- **numpy.random.rand()**: utilisé pour générer des tableaux numpy de positions et de directions aléatoires.
- **numpy.linalg.norm()**: utilisé pour calculer la norme d'un vecteur.
- **numpy.dot()**: utilisé pour calculer le produit scalaire de deux vecteurs.
- **numpy.arccos()**: utilisé pour calculer l'angle entre deux vecteurs.
- **numpy.ndarray.astype()**: utilisé pour convertir un tableau numpy en tableau d'entiers.
- **pygame.init()**: utilisé pour initialiser la bibliothèque pygame.
- **pygame.display.set_mode()**: utilisé pour créer une fenêtre.
- **pygame.time.Clock()**: utilisé pour créer un objet d'horloge de jeu.
- **pygame.font.Font()**: utilisé pour créer un objet de police de caractères.
- **pygame.event.get()**: utilisé pour obtenir les événements de la fenêtre.
- **pygame.event.type()**: utilisé pour obtenir le type d'un événement.
- **pygame.KEYDOWN et pygame.KEYUP**: utilisé pour détecter les événements de pression et de relâchement de touche.
- **pygame.draw.circle()**: utilisé pour dessiner des cercles.
- **pygame.display.flip()**: utilisé pour actualiser la fenêtre.
- **clock.tick()**: utilisé pour contrôler le taux de rafraîchissement de la boucle de jeu.

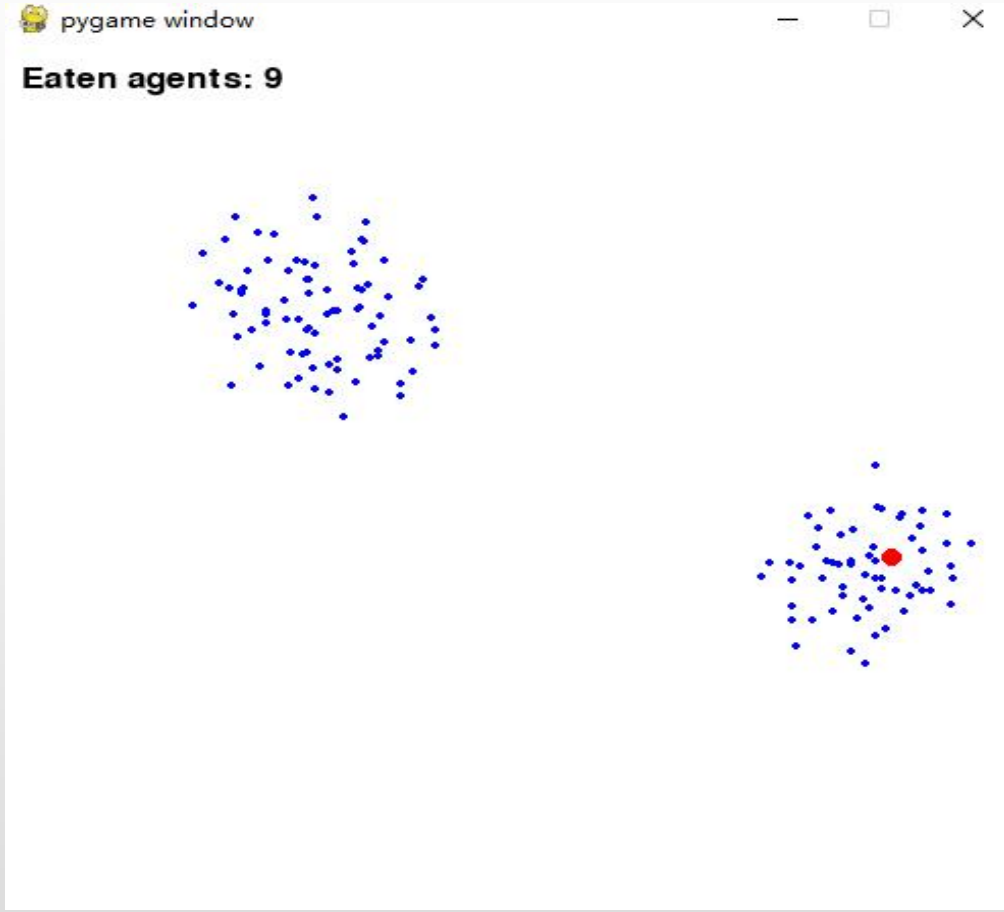
```python
# Initialise Pygame
pygame.init()
screen = pygame.display.set_mode((width, height))
clock = pygame.time.Clock()

eaten_count = 0
```

```python
def display_eaten_count():
    global eaten_count
    font = pygame.font.Font(None, 25)
    text = font.render(f'Eaten agents: {eaten_count}', True, (0, 0, 0))
    screen.blit(text, (10, 10))

agents = [Agent(np.random.rand(2) * np.array([width, height]), np.random.rand(2) - 0.5) for _ in range(N)]
predator = Agent(np.random.rand(2) * np.array([width, height]), np.random.rand(2) - 0.5)

running = True
while running:
    screen.fill((255, 255, 255))

    for agent in agents[:]:
        agent.update(agents, predator)
        pygame.draw.circle(screen, (0, 0, 255), agent.position.astype(int), 2)

        # Check if the predator is close enough to eat the agent
        if np.linalg.norm(agent.position - predator.position) < eating_distance:
            agents.remove(agent)
            eaten_count += 1  # Increment the eaten count

    predator.position += speed * predator.heading
    if predator.position[0] < 0:
        predator.position[0] = 0
        predator.heading[0] = -predator.heading[0]
    elif predator.position[0] > width:
        predator.position[0] = width
        predator.heading[0] = -predator.heading[0]

    if predator.position[1] < 0:
        predator.position[1] = 0
        predator.heading[1] = -predator.heading[1]
    elif predator.position[1] > height:
        predator.position[1] = height
        predator.heading[1] = -predator.heading[1]

    pygame.draw.circle(screen, (255, 0, 0), predator.position.astype(int), 5)

    display_eaten_count()
    pygame.display.flip()
    clock.tick(120)

    for event in pygame.event.get():
        if event.type == QUIT:
            running = False
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                running = False
        elif event.type == pygame.KEYUP:
            pass

pygame.quit()
```

◆ Logique principale et idées de mise en œuvre du code

- Ce code simule le comportement et l'interaction de plusieurs **agents (particules)** et d'un **prédateur (cercle rouge)**. Les agents se déplacent selon certaines règles et paramètres (comme la répulsion, l'alignement, l'attraction, etc.) et évitent les collisions. Le prédateur se déplace aléatoirement et tente de manger les agents proches. Le programme répond aux événements de la fenêtre, notamment les événements de sortie et de touche de clavier, pour contrôler l'exécution du programme.

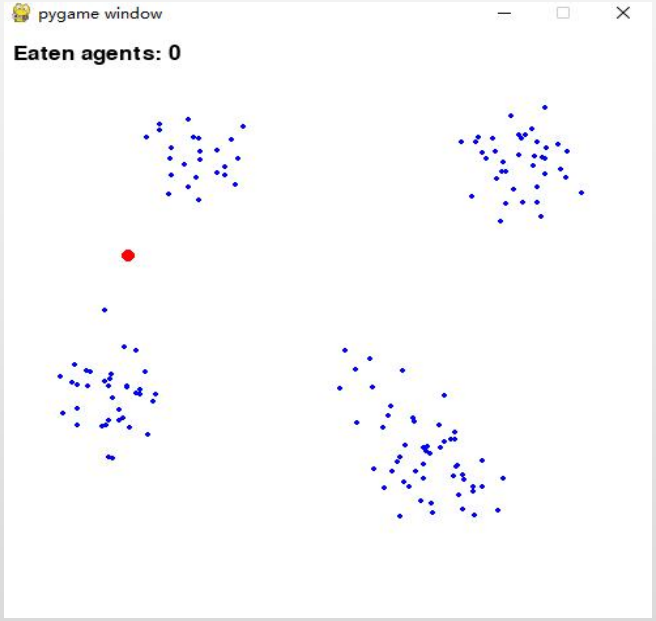


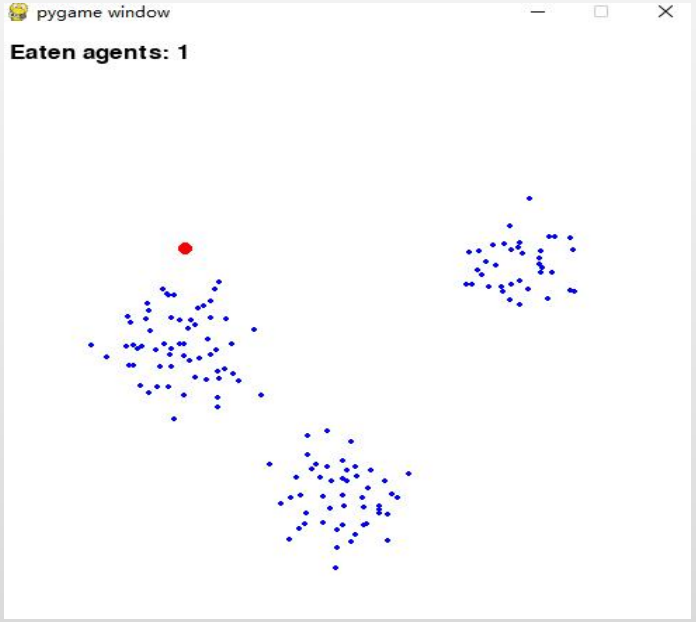


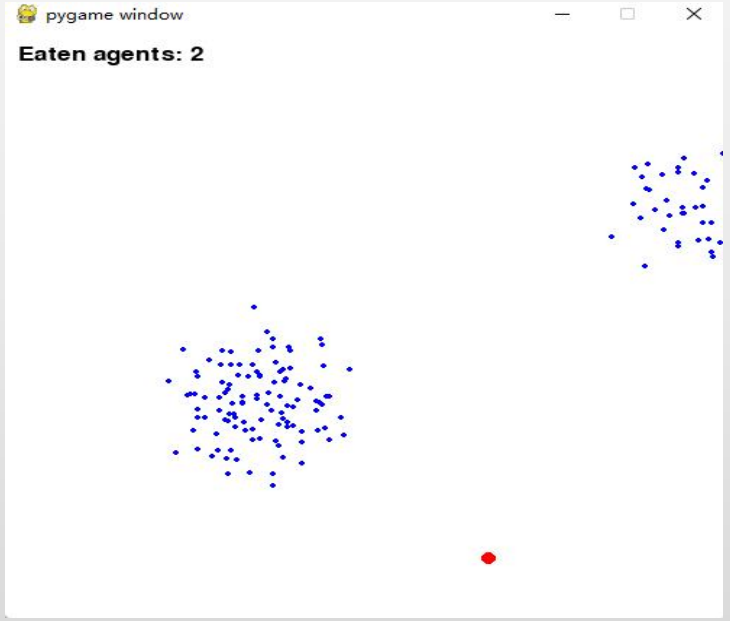
Résultats et analyse de la simulation

◆ Résultats et caractéristiques de la simulation

- La distribution des agents et des prédateurs dans l'espace présente des caractéristiques de regroupement et de dispersion. Au cours de la simulation, les agents et les prédateurs **se rapprochent et s'éloignent** constamment les uns des autres, créant des phénomènes de regroupement et de dispersion.
- L'interaction et la coopération entre les agents peuvent former des comportements collectifs, tels que la polarisation et la concentration. Dans la simulation, lorsque la distance entre les agents est inférieure à un **certain seuil**, ils commencent à coopérer mutuellement, formant ainsi certains comportements collectifs, tels que la polarisation et la concentration.
- Les paramètres qui contrôlent le comportement des prédateurs (**tels que le rayon de prédation, la vitesse, etc**.) ont un impact significatif sur la distribution et les comportements des agents. Dans la simulation, nous avons constaté que lorsque le rayon de prédation et la vitesse des prédateurs augmentent, ils sont plus susceptibles de capturer des agents, ce qui entraîne des changements dans la distribution et les comportements des agents.





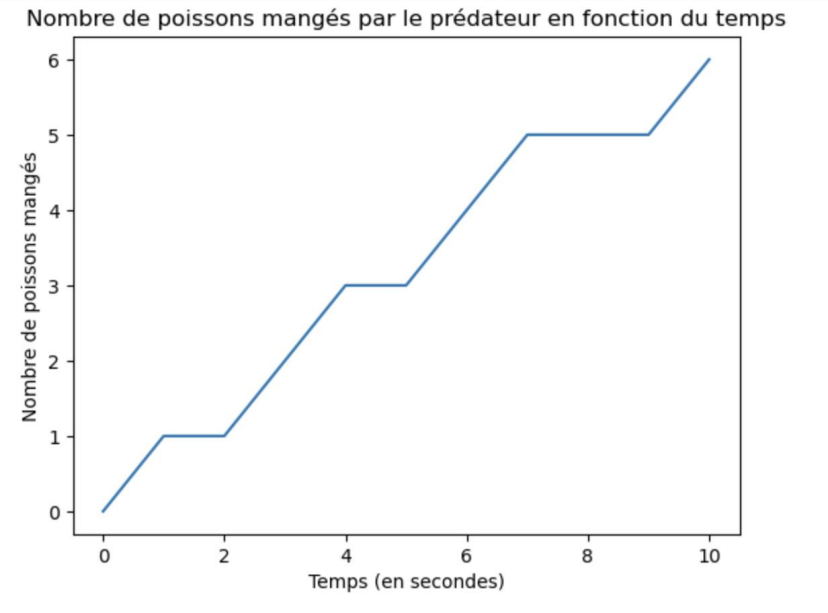


◆ Impact des paramètres de la simulation sur les résultats

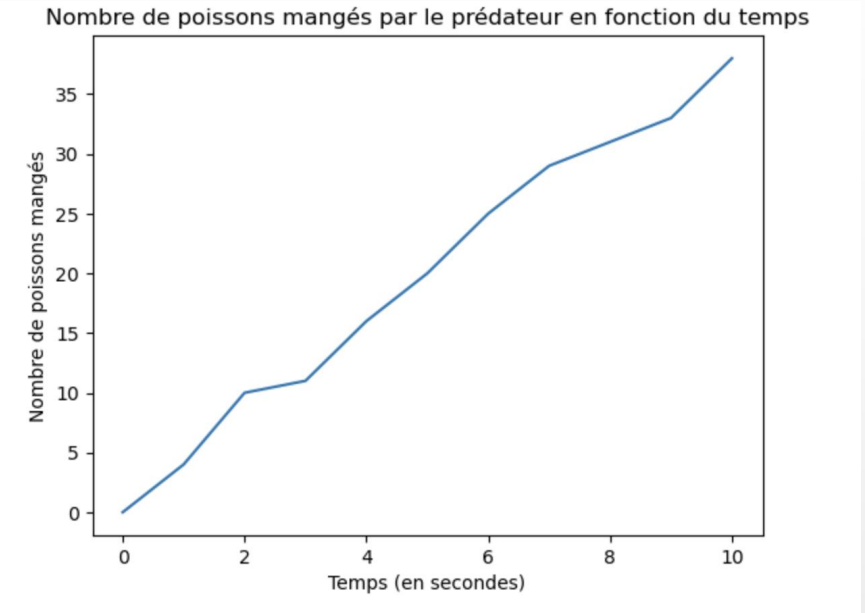
- Dans cette simulation, nous avons défini plusieurs paramètres et réglages qui peuvent influencer les résultats et les comportements de la simulation. Par exemple :
- **Vitesse** : lorsque la vitesse des agents et des prédateurs augmente, leurs trajectoires de mouvement deviennent plus rapides et plus complexes.
- **Champ de vision** : lorsque le champ de vision des agents et des prédateurs augmente, ils peuvent percevoir plus d'informations sur les voisins et l'environnement, ce qui modifie leurs comportements et leurs distributions.
- **Rayon de prédation** : lorsque le rayon de prédation des prédateurs augmente, ils peuvent plus facilement capturer des agents, ce qui entraîne des changements dans la distribution et les comportements des agents.

Impact des paramètres de la simulation sur les résultats: Représentation graphique.

- Le paramètre "eating_distance" représente la distance à laquelle le prédateur doit être d'une proie pour la manger.



Eating Radius : 10



Eating Radius : 50

- Une valeur trop élevée peut également rendre la chasse trop facile pour le prédateur, ce qui peut rendre la simulation moins intéressante. Il est donc important de trouver un équilibre entre la portée de l'attaque du prédateur et la difficulté de la chasse pour maintenir un niveau de complexité intéressant dans la simulation.

Applications et significations de la simulation

- Les systèmes multi-agents et les comportements de groupe ont des applications pratiques dans la vie quotidienne, telles que l'évolution des comportements de groupe, la planification de la circulation urbaine, etc.
- La simulation peut être utilisée pour optimiser la planification de la circulation routière, prédire les migrations animales, etc.
- La simulation est significative car elle permet de mieux comprendre les principes et les caractéristiques des systèmes multi-agents et des comportements de groupe.

Limites de la simulation et développements futurs

- Les limites et les insuffisances de la simulation incluent : l'utilisation de nombreuses **hypothèses et paramètres simplifiés** dans la simulation, qui ne reflètent pas pleinement la complexité et la diversité des systèmes à multiples agents et des comportements de groupe dans le monde réel ; une **sensibilité** élevée des résultats de la simulation aux états initiaux et aux paramètres, ce qui peut conduire à des résultats différents ; et le manque de considération des facteurs tels que **la différenciation et l'adaptation individuelles** entre les agents.

- Les futures directions de développement et d'amélioration de la simulation pourraient inclure : l'ajout de comportements et de caractéristiques **plus complexes**, comme la simulation des **comportements sociaux** entre les agents, la simulation des processus cognitifs et décisionnels des agents ; l'augmentation de **la complexité** de la simulation, comme la simulation des interactions entre différents types d'agents et de prédateurs, la simulation de la compétition entre les agents ; et le développement d'**algorithmes** et de technologies de simulation plus avancés, tels que l'utilisation de méthodes d'**apprentissage profond** pour simuler les systèmes à multiples agents et les comportements de groupe.

Merci beaucoup！！！