WOJSKOWA AKADEMIA TECHNICZNA

Sprawozdanie z przedmiotu:

Sztuczna Inteligencja

Prowadzący: mgr inż. Przemysław Czuba

Autor: Mateusz Jasiński WCY20IY2S1

1. **Aktualna postać programu implementuje jedynie rozdzielność. Zaimplementuj pozostałe dwie reguły (spójność i wyrównanie). Nagraj krótki film pokazujący działanie programu i dołącz je do sprawozdania.**

```
def cohesion(self):
    steering = np.zeros(2)
    neighbours_number = 0
    for other_boid in self.flock:
        distance_between_boids = np.linalg.norm(np.subtract(self.position,
other_boid.position))
        if other_boid is not self and distance_between_boids <
BOID_PERCEPTION_DISTANCE:
            steering = np.add(steering, other_boid.position)
            neighbours_number += 1
    if neighbours_number > 0:
        steering = steering/neighbours_number
        steering = np.subtract(steering, self.position)
        steering = set_vector_magnitude(steering, BOID_MAX_SPEED)
        steering = np.subtract(steering, self.velocity)
        steering = set_vector_magnitude(steering, BOID_COHESION_FACTOR)
    return steering

def alignment(self):
    steering = np.zeros(2)
    neighbours_number = 0
    for other_boid in self.flock:
        distance_between_boids = np.linalg.norm(np.subtract(self.position,
other_boid.position))
        if other_boid is not self and distance_between_boids <
BOID_PERCEPTION_DISTANCE:
            steering = np.add(steering, other_boid.velocity)
            neighbours_number += 1
    if neighbours_number > 0:
        steering = steering/neighbours_number
        steering = set_vector_magnitude(steering, BOID_MAX_SPEED)
        steering = np.subtract(steering, self.velocity)
        steering = set_vector_magnitude(steering, BOID_ALIGNMENT_FACTOR)
    return steering
```

2. W aktualnej postaci programu agent “patrzy do tyłu”. Dodaj maksymalny kąt pod jakim agent może patrzeć na boki.

W pliku params.py dodaję parametr MAX_ANGLE przedstawiający maksymalny kąt, pod jakim agent może patrzeć na boki.
Implementuję funkcje, która oblicza kąt między dwoma wektorami i zwraca kąt wyrażony w stopniach:

```
def get_angle(vec_1, vec_2):
    unit_vec_1 = vec_1 / np.linalg.norm(vec_1)
    unit_vec_2 = vec_2 / np.linalg.norm(vec_2)
    dot_product = np.dot(unit_vec_1, unit_vec_2)
    angle = abs(np.rad2deg(np.arccos(dot_product)))
    return angle
```

Dodatkowo muszę uwzględnić w kodzie zaimplementowaną funkcję, w każdej regule dodając/modyfikując warunek:

```
if other_boid is not self and distance_between_boids <
BOID_PERCEPTION_DISTANCE:
    if get_angle(self.velocity, np.subtract(self.position,
other_boid.position)) < MAX_ANGLE:
```

3. Zdefactoruj program tak, aby pozbyć się powtarzającego się kodu.

Dodaję dwie nowe metody, które zawierając w sobie kod powtarzający się w trzech zasadach:

```
def get_neighbours(self, neighbours_number, steering, rule):
    for other_boid in self.flock:
        distance_between_boids = np.linalg.norm(np.subtract(self.position,
other_boid.position))
        if other_boid is not self and distance_between_boids <
BOID_PERCEPTION_DISTANCE:
            if get_angle(self.velocity, np.subtract(self.position,
other_boid.position)) < MAX_ANGLE:
                if rule == "separation":
                    difference = np.subtract(self.position,
other_boid.position)
                    steering = np.add(steering, difference /
distance_between_boids)
                if rule == "cohesion":
                    steering = np.add(steering, other_boid.position)
                if rule == "alignment":
                    steering = np.add(steering, other_boid.velocity)
                neighbours_number += 1
    return neighbours_number, steering

def set_steering(self, neighbours_number, steering, factor, rule):
    if neighbours_number > 0:
        steering = steering / neighbours_number
        if rule == "cohesion":
            steering = np.subtract(steering, self.position)
        steering = set_vector_magnitude(steering, BOID_MAX_SPEED)
        steering = np.subtract(steering, self.velocity)
        steering = set_vector_magnitude(steering, factor)
    return steering
```

Dzięki temu mogę pozbyć się powtarzającego się kodu w metodach cohesion, alignment i separation:

```
def cohesion(self):
    steering = np.zeros(2)
    neighbours_number = 0
    neighbours_number, steering = self.get_neighbours(neighbours_number,
steering, "cohesion")
    steering = self.set_steering(neighbours_number, steering,
BOID_COHESION_FACTOR, "cohesion")
    return steering

def alignment(self):
    steering = np.zeros(2
    neighbours_number = 0
    neighbours_number, steering = self.get_neighbours(neighbours_number,
steering, "alignment")
    steering = self.set_steering(neighbours_number, steering,
BOID_ALIGNMENT_FACTOR, "alignment")
    return steering

def separation(self):
    steering = np.zeros(2)
    neighbours_number = 0
    neighbours_number, steering = self.get_neighbours(neighbours_number,
steering, "separation")
    steering = self.set_steering(neighbours_number, steering,
BOID_SEPARATION_FACTOR, "separation")
    return steering
```

4. Jaka jest złożoność algorytmu. Podaj ograniczenie górne wraz z uzasadnieniem. Co można zrobić, aby zoptymalizować algorytm? Opisz możliwości.

Podstawowe implementacje algorytmu są bardzo mało efektywne. Boid musi wziąć pod uwagę wszystkie inne boidy, aby obliczyć swoją prędkość, mimo że jedynie te blisko niego mają na nią wpływ. Złożoność tego algorytmu wynosi $O(n^2)$, gdzie n to liczba boidów. Ograniczenie górne zależy od specyfikacji sprzętu.
W miarę dodawania do symulacji kolejnych boidów znacznie obciążamy program dodatkowymi obliczeniami. Dobrym sposobem na zwiększenie wydajności boidów jest wykorzystanie struktur danych przestrzennych, które przechowują boidy na podstawie ich pozycji, np. kafelkowanie. Pozwala to na porównywanie tylko boidów przechowywanych razem, oszczędzając obliczenia.
Jednak lepszym sposobem jest użycie struktury drzewa k-wymiarowego (kd). Jest ono drzewem binarnym, w którym w każdym węźle zewnętrznym znajduje się k-wymiarowy punkt. Każdy węzeł wewnętrzny tworzy hiperpłaszczyznę podziału, która dzieli przestrzeń na dwie podprzestrzenie. Punkty po lewej stronie hiperpłaszczyzny reprezentują lewe poddrzewo zaczynające się w tym węźle, a prawe punkty – prawe poddrzewo. Kierunek hiperpłaszczyzny jest wybierany zgodnie z wektorem normalnym do niej.
Budowanie statycznego drzewa kd z n punktów ma następującą złożoność:

- $O(n \log^2 n)$, jeśli używane, do znalezienia mediany na każdym poziomie powstającego drzewa, jest sortowanie o złożoności $O(n \log n)$, takie jak sortowanie sterty, sortowanie przez scalanie lub sortowanie szybkie,
- $O(n \log n)$, jeśli algorytm mediany median $O(n)$ jest używany do wyboru mediany na każdym poziomie drzewa,
- $O(kn \log n)$, jeśli n punktów jest wstępnie posortowanych w każdym z wymiarów k przy użyciu sortowania $O(n \log n)$, takiego jak sortowanie sterty, sortowanie przez scalanie lub sortowanie szybkie.

Dzięki zmodyfikowanemu wyszukiwaniu najbliższych sąsiadów, sąsiedzi aktualnie wybranego boidu mogą być znalezieni w średnim czasie $O(\log n)$. Całkowity czas wyszukiwania dla każdego boida wynosiłby $O(n \log n)$, który jest znacznie niższy (dla większej liczebności boidów) niż przy naszym pierwotnym rozwiązaniu. Po znalezieniu wszystkich najbliższych sąsiadów są oni przekazywani do funkcji, która stosuje reguły i oblicza siły sterujące. Przez takie rozwiązanie czas wyszukiwania jest dość skrócony dzięki użyciu drzewa kd i algorytmu wyszukiwania najbliższego sąsiada.