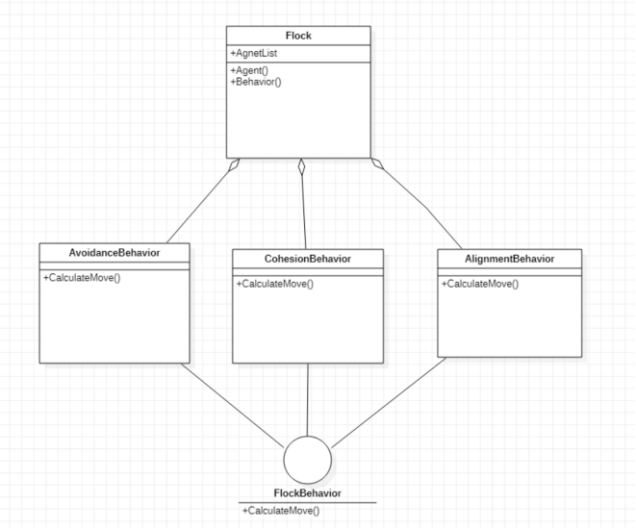
Project3

게임프로그래밍입문 00분반

소프트웨어융합학과 2022105458 박소영

1. Goal



Flocking algorithm은 떼를 이루는 동물의 행동을 모방하여 그룹으로 움직이는 객체들을 제어하는 기술이다.

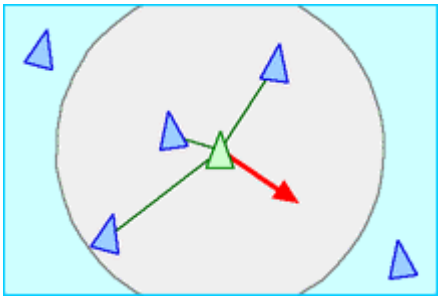
주로 새떼나 물고기 떼와 같은 자연의 집단 행동을 모방하는데 사용된다. 이 알고리즘을 통해 객체들은 서로 간의 상호작용을 통해 그룹 내에서 적절한 거리를 유지하거나 일정한 방향으로 움직이는 등의 행동을 할 수 있다. 이는 다양한 상황에 대응하여 그룹내 조율된 움직임을 가능하게 하며 게임에서는 이를 통해 NPC나 적들이 현실적인 움직임을 갖게 되어 게임 세계가 더 생생하고 자연스러워질 수 있다.

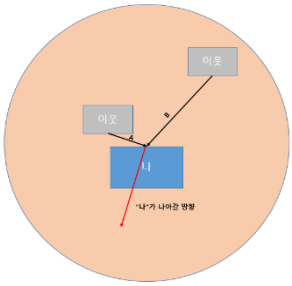
최종적인 목표는 원하는 수의 boid를 만들고 3가지 종류로 랜덤하게 나누어 각 종류가 flocking algorithm에 따라 flock을 이루는 형태를 보이도록 하는 것이다.

2. Game Engine Design & Structure

Flocking Algorithm은 기본적으로 3가지 규칙을 따르게 된다

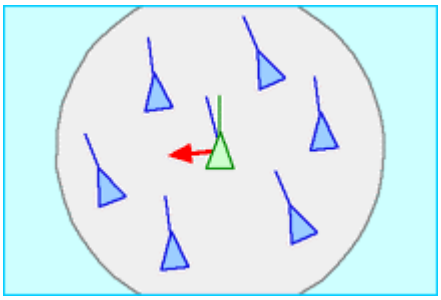
1) Seperation (분리)





Speration(분리)는 자기 주변의 객체들이 붐비는 것을 피하기 위해 근처 이웃들에서 벗어나는 규칙이다

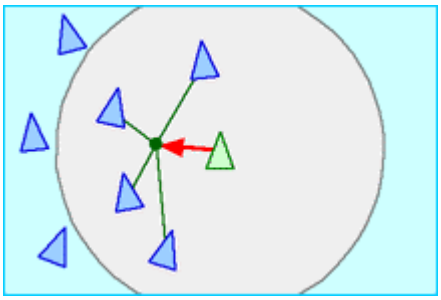
2) Alignment (정렬)



Alignment(정렬)은 이웃 객체들의 평균 방향으로 이동하는 규칙이다.

위 예시 이미지를 예로 들자면, 이웃 객체들이 11~12시 방향으로 이동하기 때문에 초록색의 이동 방향 또한 평균 방향으로 변경된다.

3) Cohension (응집력)

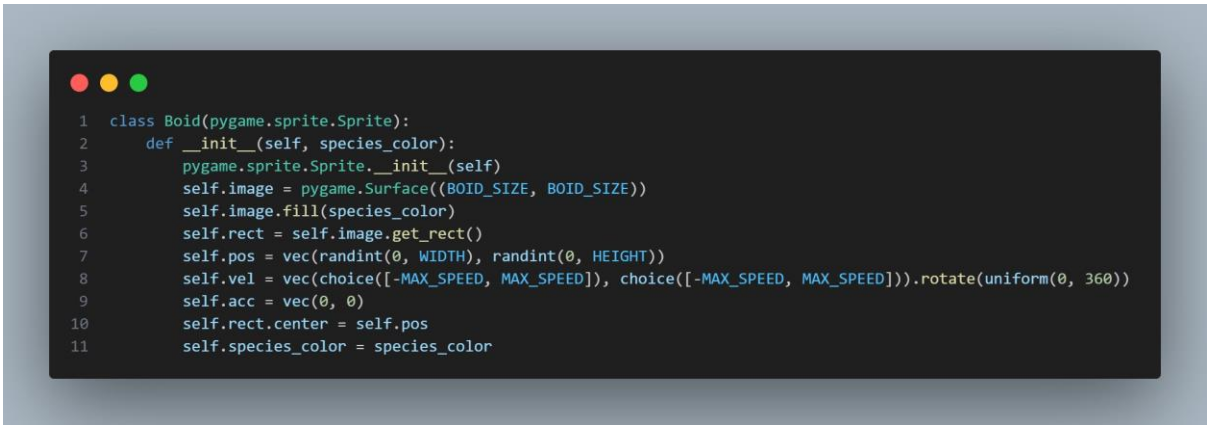


Cohension(응집력)은 모든 이웃 사이의 중간점(평균 위치)를 찾고 중간점을 향해 이동하는 규칙이다.

군집 이동을 할 객체는 3가지 규칙에 따라 나아갈 방향을 구하고 그를 향해 이동하게 된다.

3. Feature Description along with Code Description

<Boid Class>



```python
class Boid(pygame.sprite.Sprite):
    def __init__(self, species_color):
        pygame.sprite.Sprite.__init__(self)
        self.image = pygame.Surface((BOID_SIZE, BOID_SIZE))
        self.image.fill(species_color)
        self.rect = self.image.get_rect()
        self.pos = vec(randint(0, WIDTH), randint(0, HEIGHT))
        self.vel = vec(choice([-MAX_SPEED, MAX_SPEED]), choice([-MAX_SPEED, MAX_SPEED])).rotate(uniform(0, 360))
        self.acc = vec(0, 0)
        self.rect.center = self.pos
        self.species_color = species_color
```

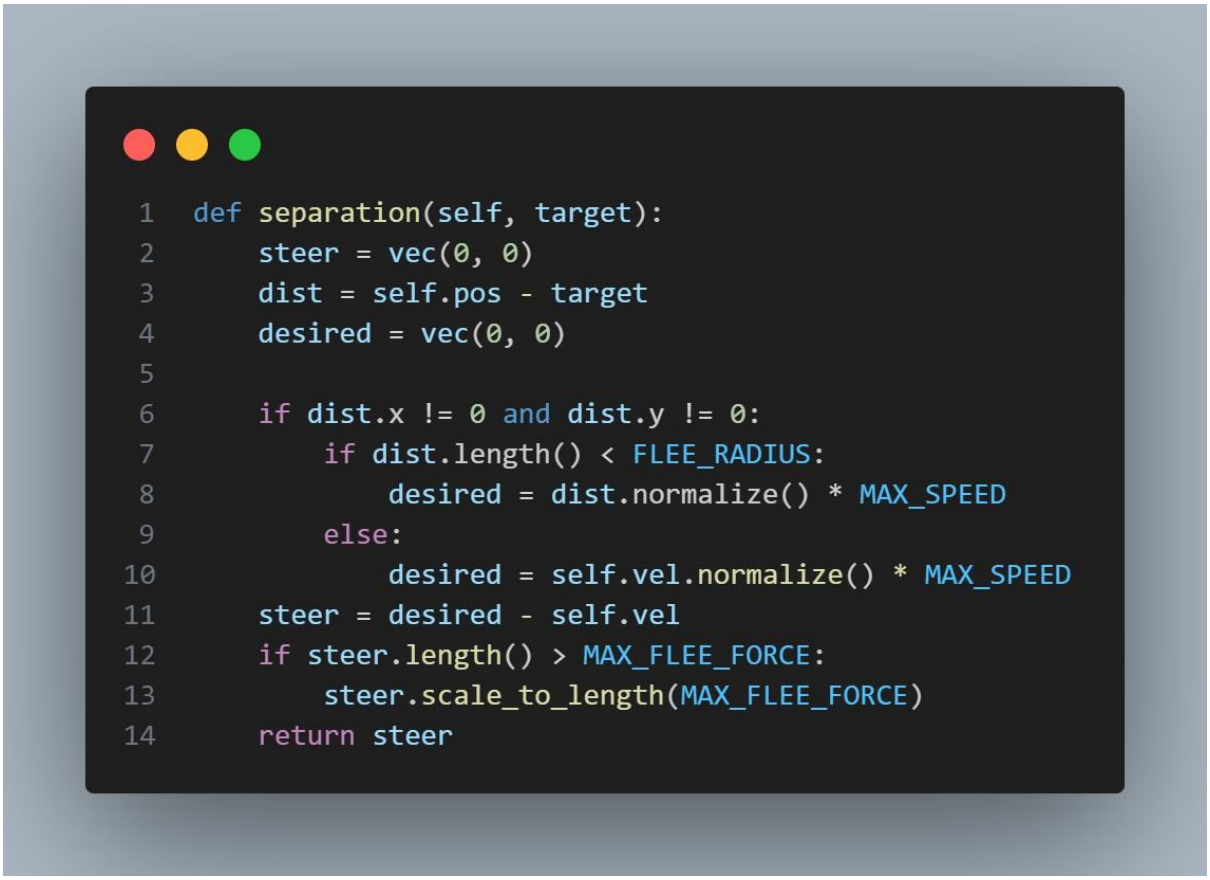
(BOID_SIZE, BODSIZE) 크기의 지정된 species_color로 채워 boid의 색상을 지정한다.

Boid의 초기좌표는 지정된 화면 크기 내 무작위 좌표로, 초기 속도는 -최대속도와 +최대 속도 사이의 벡터로 설정하고 무작위 각도로 회전시킨다.

Self.acc = (vec0,0)을 통해 boid의 가속도 벡터를 초기화한다

Boid의 위치는 rect로 만들어지는 이미지의 중심으로 설정한다.

나중에 참조하기위해 boid의 species의 색을 self.species _color에 저장한다.



```python
def separation(self, target):
    steer = vec(0, 0)
    dist = self.pos - target
    desired = vec(0, 0)

    if dist.x != 0 and dist.y != 0:
        if dist.length() < FLEE_RADIUS:
            desired = dist.normalize() * MAX_SPEED
        else:
            desired = self.vel.normalize() * MAX_SPEED
    steer = desired - self.vel
    if steer.length() > MAX_FLEE_FORCE:
        steer.scale_to_length(MAX_FLEE_FORCE)
    return steer
```

Desired 변수 – boid가 이동하려는 방향을 나타내는 벡터.

Steer 변수 – 방향을 조절하는 벡터로 boid가 현재 이동하는 방향과 desired 방향 간의 차이를 나타내어 현재 boid의 이동 방향을 desired 방향으로 조정하게 한다.

FlEE_RADIUS 변수 – 다른 boid로부터 얼마나 멀리 떨어져야 하는지를 나타낸다.

Dist 변수 – 현재 boid와 타겟 boid 사이의 거리를 나타낸다

MAX_FLEE_FORCE – 최대 피하기 힘

현재 boid와 타겟 boid 사이 거리가 0이 아닐 때 둘의 거리가 FLEE_RADIUS보다 작은지 확인한다.

1) FLEE_RADIUS보다 작은 경우 : 거리 벡터를 단위 벡터(방향 정보만 포함)로

정규화하고 MAX_SPEED를 곱해 원하는 속도 벡터를 만든다

2) FLEE_RADIUS보다 큰 경우 : 현재 boid의 속도 방향으로 이동하는 벡터를 설정한다

최종적으로, 이동하려는 방향 벡터인 desired 에서 현재의 속도 벡터를 빼서 최종적인 steer 벡터를 얻는다.

이 steer 벡터가 MAX_FLEE FORCE를 넘지 않도록 제한한다.

```python
def alignment(self):
    align = vec(0, 0)
    desired = vec(0, 0)
    for i in boids:
        if i != self and i.species_color == self.species_color:
            if i.vel.x != 0 and i.vel.y != 0:
                if (self.pos - i.pos).length() < ALIGN_RADIUS:
                    desired += i.vel.normalize() * MAX_SPEED

    align = desired - self.vel
    if len([b for b in boids if b.species_color == self.species_color]) > 0:
        align /= len([b for b in boids if b.species_color == self.species_color])

    if align.length() > MAX_SPEED:
        align.scale_to_length(MAX_SPEED)

    return align
```

Align : 최종적인 정렬 방향을 나타내는 벡터 변수

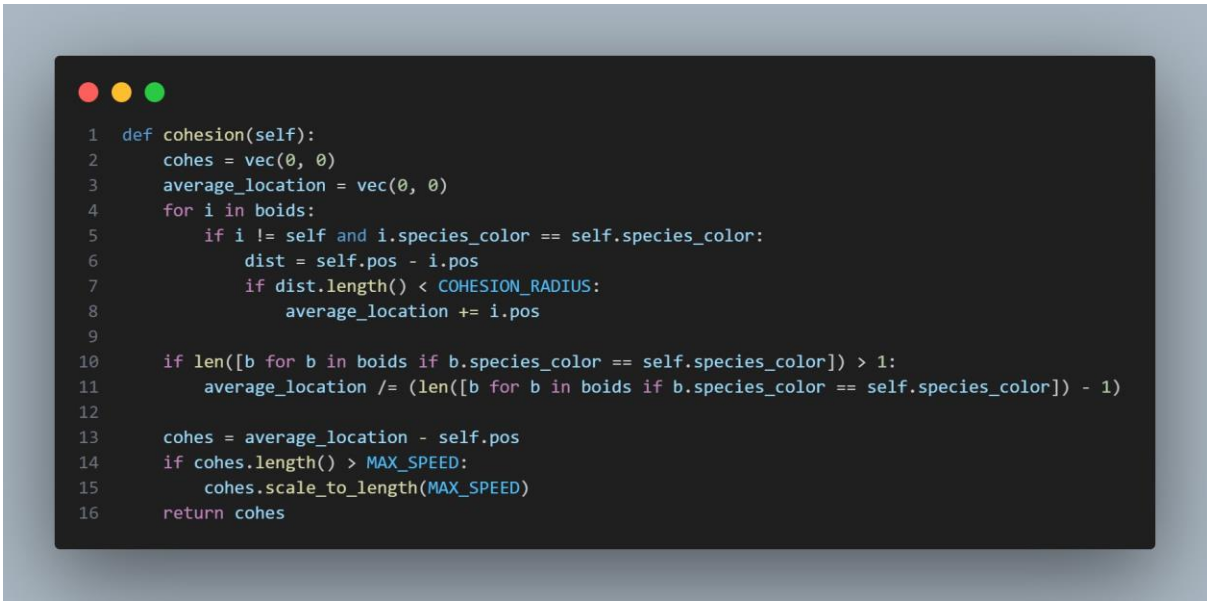
Desired : 이웃 boid들의 평균 이동 방향을 타나내는 벡터 변수

모든 boid들은 자신을 제외한 같은 species에 속한 boid에 대해서 계산을 한다.

같은 species의 이웃 boid의 속도 벡터가 올바르게 설정되어 있고 현재 boid와 이웃 boid 간의 거리 ALIGN_RADIUS보다 작은 경우 desired 벡터에 이웃 boid의 정규화된 속도 베터 * 최대 속도를 곱한 값을 더하여 desired에 저장한다.

최종적으로, 현재 boid의 속도 벡터에서 desired 벡터를 빼 최종적인 align 변수를 얻는다.

현재 species속한 boid가 적어도 하나 이상 있는 경우 해당 species의 모든 boid의 수로 align 벡터를 나누어 평균적인 속도를 계산하고 MAX_SPEED로 이를 제한한다.



```python
def cohesion(self):
    cohes = vec(0, 0)
    average_location = vec(0, 0)
    for i in boids:
        if i != self and i.species_color == self.species_color:
            dist = self.pos - i.pos
            if dist.length() < COHESION_RADIUS:
                average_location += i.pos

    if len([b for b in boids if b.species_color == self.species_color]) > 1:
        average_location /= (len([b for b in boids if b.species_color == self.species_color]) - 1)

    cohes = average_location - self.pos
    if cohes.length() > MAX_SPEED:
        cohes.scale_to_length(MAX_SPEED)
    return cohes
```

Cohes : 최종적인 응집 방향을 나타내는 벡터

Average_location : 이웃 boid들의 평균 위치를 나타내는 벡터

모든 boid드들에 대해 자신이 아니며 같은 species에 속하는 경우 자신과 이웃 boid 간의 거리 벡터를 계산한다.

이 거리가 cohesion_radius보다 작은 경우(이웃 boid가 일정 거리 내에 있는 경우)에만 해당 이웃을 고려하여 이웃 boid의 위치를 average_location에 더한다.

현재 species에 속한 boid가 1개 이상인 경우 average_location을 species의 속하는 boid들의 수로 나누어 평균 위치를 계산한다.

Cohes에 이웃 boid들의 평균 위치에서 현재 평균 boid의 위치를 빼 저장하여 MAX_SPEED로 제한한다.

```python
def update(self):
        self.acc = vec(0, 0)
        for i in boids:
            if i != self and i.species_color == self.species_color:
                self.acc += self.separation(i.rect.center)
        self.acc += self.alignment()
        self.acc += self.cohesion()

        self.vel += (self.acc * DELTA_TIME)

        if self.vel.length() > MAX_SPEED:
            self.vel.scale_to_length(MAX_SPEED)

        self.pos += self.vel

        # Screen boundaries check and adjustment
        if self.pos.x > WIDTH:
            self.pos.x = WIDTH
            self.vel.x *= -1
        elif self.pos.x < 0:
            self.pos.x = 0
            self.vel.x *= -1
        if self.pos.y > HEIGHT:
            self.pos.y = HEIGHT
            self.vel.y *= -1
        elif self.pos.y < 0:
            self.pos.y = 0
            self.vel.y *= -1

        self.rect.center = self.pos
```

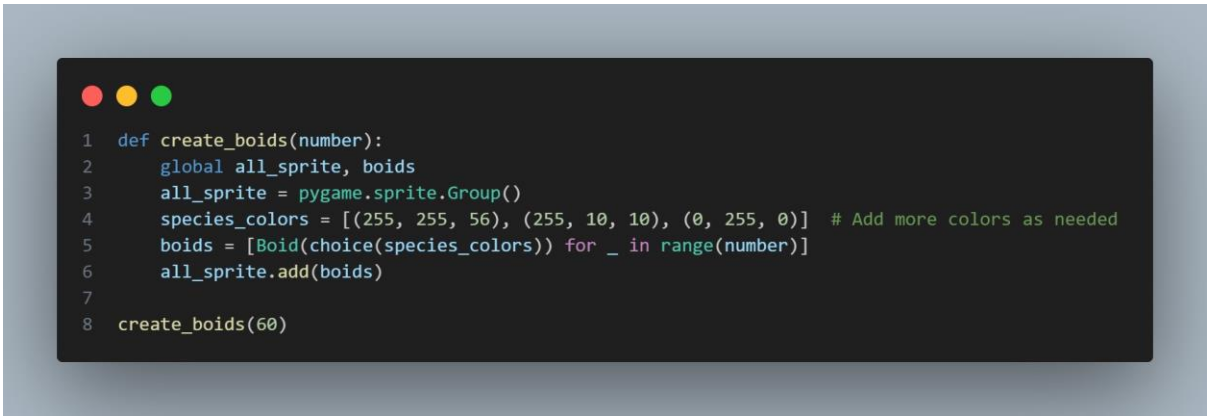
Self.acc : 현재 boid의 가속도 벡터

모든 boid에 대해 반복을 한다. 현재 boid 자신이 아니고 같은 species의 경우 이웃 boid의 중심에 대해 seperation을 수행하여 현재 boid의 가속도를 더한다. Alignement와 cohesion의 경우에는 함수 내부에서 species를 구분하므로 바로 self.acc에 더한다.

현재 boid의 속도에 가속도를 더하고 시간 간격을 곱하여 업데이트하며 이때 MAX_SPEED를 넘지 않도록 스케일링 해준다.

이때 boid들이 화면 밖으로 나가지 않도록 화면 경계에서 방향에 -1을 곱해준다

<전역 함수>



```python
def create_boids(number):
    global all_sprite, boids
    all_sprite = pygame.sprite.Group()
    species_colors = [(255, 255, 56), (255, 10, 10), (0, 255, 0)]  # Add more colors as needed
    boids = [Boid(choice(species_colors)) for _ in range(number)]
    all_sprite.add(boids)

create_boids(60)
```

함수 내에서 새로운 boid들을 생성하여 그룹에 추가한 후에도 함수 외부에서 접근 할 수 있도록 global로 all_sprite와 boids 함수를 선언한다

All_spirte : pygame.sprtie.Group()을 통해 생성된 스프라이트 그룹으로 모든 boid 스프라이트를 관리함

Species_colors : Boid의 색상을 나타내는 튜플의 리스트로 더 많은 색상을 추가할 수 있음

Boids : number만큼의 boid 객체를 생성하며 각 boid의 색상은 species_colors에서 무작위로 선택됨

4. Execution Environment

Python 3.10.11