Project #3 Develop your own game engine

2019102212 이정호

1. Goal

물리 엔진의 가장 기본적인 기능에는 충돌 구현이 있다. 충돌을 구현하기 위해서는 여러 가지 요소들이 필요한데, 먼저 어떠한 두 객체가 충돌했다는 사실을 판단하기 위한 충돌 판정이 필요하고 이후에 충돌 판정이 난 두 객체에 어떠한 처리를 해야 하는지 충돌 처리를 해줘야 한다. 그래서 이번 프로젝트에서는 pygame에서 여러 가지 모양의 객체들에 대해서 충돌을 구현하고 이를 확인할 수 있도록 사용자가 마우스를 이용해서 객체들을 조종할 수 있도록 만들 예정이다.

2. Game Engine Design & Structure

이 문단에서는 게임 엔진에 포함될 충돌 판정과 충돌 처리가 어떤 식으로 구현되는지, 그리고 구현하기 위해서 이를 어떤 식의 구조를 가지고 있으면 좋을 지에 대해서 한 번 살펴볼 것이다. 가장 기본적으로 객체들의 이동을 관장하기 위해 원, 직선과 같은 새로운 클래스들을 만들 것이다. 객체들은 기본적으로 pygame의 draw 메소드를 통해서 화면 상에 표현되게 되며, 이를 통해 표현했을 때 필요한 수치값은 원의 경우 원의 중심을 나타내는 center_point이고, 직선의 경우 직선의 시작점 start_point와 end_point이다. 직선의 경우 벽처럼 움직이지 않도록 구현할 것이기 때문에 관계 없지만 원의 경우 계속해서 움직이도록 할 것이기 때문에 원의 중심인 center_point 값을 계속 update하면서 위치가 변하는 것을 표현할 예정이다. 또한 update의 이전 주기의 물체의 속력을 저장하고 있어야 하기 때문에 velocity라는 변수를 클래스에 포함시켜 이전 주기에서의 물체의 속력을 연속적으로 표현할 수 있도록 클래스를 구성했다.

3. Feature & Code Description

```python
class vec_2d:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def add(self, vec):
        return vec_2d(self.x + vec.x, self.y + vec.y)

    def sub(self, vec):
        return vec_2d(self.x - vec.x, self.y - vec.y)

    def mult(self, num):
        return vec_2d(self.x * num, self.y * num)

    def abs(self):
        return ((self.x ** 2 + self.y ** 2) ** (1 / 2))

    def dot(self, vec):
        return self.x * vec.x + self.y * vec.y
```

먼저 클래스들에 대해서 살펴보자. 충돌 처리 과정에서 2차원 벡터와 이를 이용한 연산들이 자주 등장할 것이므로 이에 대한 기본적인 저장과 두 벡터간의 연산들을 할 수 있는 vec_2d 클래스를 따로 만들었다.

```python
class Circle:
    def __init__(self, p_x, p_y, v_x, v_y, static):
        self.position_x = p_x
        self.position_y = p_y
        self.velocity_x = v_x
        self.velocity_y = v_y
        self.static = static

class Line:
    def __init__(self, sp_x, sp_y, ep_x, ep_y):
        self.start_position_x = sp_x
        self.start_position_y = sp_y
        self.end_position_x = ep_x
        self.end_position_y = ep_y
```

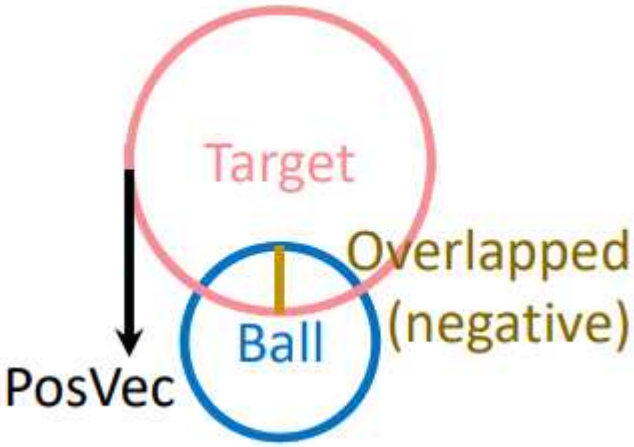
다음은 Circle과 Line 클래스이다. 각각 화면 상에 등장할 원과 직선 객체들을 표현하기 위해 필요한 수치적 정보인 position과 velocity를 저장하기 위해 만든 클래스이다.

```python
for circle in circles:
    circle.velocity_x *= 0.99
    circle.velocity_y *= 0.99
    circle.position_x += circle.velocity_x
    circle.position_y += circle.velocity_y
```

다음은 원 객체들의 속도를 담당하는 코드이다. 기본적으로 원 객체들은 생성과 동시에 circles라는 list에 저장되도록 했다. 매 update 주기에서 원 객체들은 화면 상에 position이라는 멤버 변수에 의해서 표현될 위치가 정해진다. 여기서 이전 주기와의 연속성을 위해 velocity 멤버 변수를 이용하는데 $s = s_0 + vt$이기 때문에 $t = 1$이므로 $s = s_0 + v$ 형태로 표현할 수 있게 된다. 이를 코드로 표현하면 위와 같이 표현할 수 있다. 또 마찰력을 추가하기 위해 속도에 1보다 작은 수치를 계속해서 더해줌으로써 속도가 작아지도록 구현했다. 정확한 방법은 아니지만 시각적으로 봤을 때는 차이를 느끼지 못한다.

```python
for circ (variable) pos_vec: vec_2d
    for
        pos_vec = vec_2d(circle1.position_x - circle2.position_x, circle1.position_y - circle2.position_y)
        overlap = pos_vec.abs() - 30

        if circle1 == circle2:
            continue

        else:
            if overlap <= 0:

                collisions.append([circle1, circle2])

                if pos_vec.abs() == 0:
                    circle1.position_x += 1
                    circle1.position_y += 1

                else:
                    circle1.position_x -= pos_vec.mult(overlap / pos_vec.abs() * 0.5).x
                    circle1.position_y -= pos_vec.mult(overlap / pos_vec.abs() * 0.5).y
                    circle2.position_x += pos_vec.mult(overlap / pos_vec.abs() * 0.5).x
                    circle2.position_y += pos_vec.mult(overlap / pos_vec.abs() * 0.5).y
```

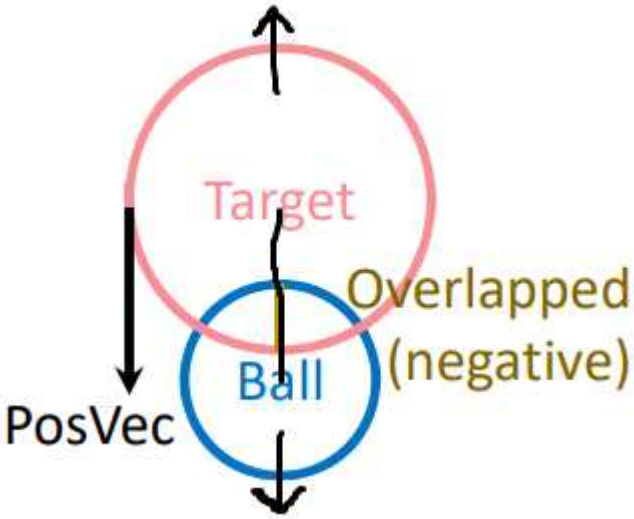
다음은 원과 원의 충돌 판정과 1차적인 충돌 처리에 대한 내용이다. 코드의 내용을 직접적으로 살펴보기에 앞서 원리에 대해 살펴보자.



다음 그림을 보면 어떤 방식으로 두 원 객체가 충돌했다고 판정하는지에 대해서 알 수 있다. 먼저 충돌을 판정할 두 객체의 center_point값을 vec_2d의 메소드 sub을 이용해서 차 값을 구해준다. 만일 이 차 값에서 두 원 객체의 반지름 값을 모두 빼준 값인 overlap 값이 음수 값을 가진다면, 두 객체가 겹쳐 있다는 것과 같은 의미이기 때문에 두 객체가 충돌했다고 판정할 수 있다. 따라서 이 overlap 값을 통해서 두 객체가 충돌했는지 판정할 것이다.

다시 코드로 돌아와서 circles 내부에 있는 모든 circle 객체에 대해서 확인한다. 서로 다른 두 개의 객체에 대해서 확인해야 하기 때문에 반복문을 같은 circles에 대해 두 번 중첩하여 구성한다. 만약 circle1과 circle2가 같은 객체, 즉 한 개의 객체에 대해서 충돌을 판정한다면 당연히 충돌했다고 판정될 것이기 때문에 이런 상황은 사전에 배제하고 할 수 있도록 한다.

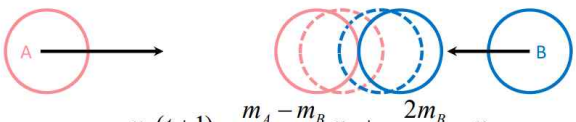
이후는 위의 그림을 통해서 설명했던 상황과 동일하다. pos_vec가 두 원 객체의 center_point를 잇는 벡터이고, abs 메소드를 이용해서 거리를 구해준다. 원 객체의 radius는 전부 15로 똑같이 고정되어 있기 때문에 30을 빼준 값을 overlap이라고 볼 수 있다. 그래서 이 overlap 값이 음수라면 두 원 객체가 충돌했다고 판정할 수 있다. 이후 충돌 사후처리를 위해 collisions list에 두 원 객체를 순서쌍의 형태로 넣어주고 다음 update 주기에서 또 충돌 처리가 발생하는 것을 방지하기 위해 pos_vec를 축으로 해서 서로 반대 방향으로 각 객체를 밀어준다. 만약 두 원 객체가 완벽하게 center_point가 겹친다면 임의로 한 쪽 원 객체를 x, y축으로 1만큼씩 밀어준다.



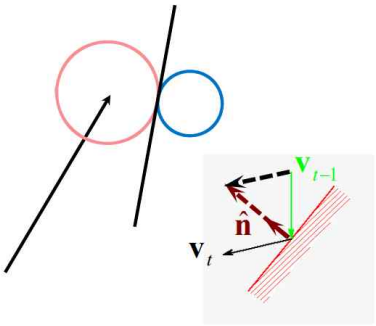
서로 반대 방향으로 overlap의 절반 만큼만 밀어주면 되기 때문에 pos_vec를 normalize 해주고 overlap의 절반 만큼을 곱해 circle의 position값에 적절히 더하거나 빼준다.

```
for collision in collisions:
    pos_vec = vec_2d(collision[0].position_x - collision[1].position_x, collision[0].position_y - collision[1].position_y)

    if pos_vec.abs() == 0:
        Normal = pos_vec.mult(1 / 1E-5)

    else:
        Normal = pos_vec.mult(1 / pos_vec.abs())

    dif_x = collision[0].velocity_x - collision[1].velocity_x
    dif_y = collision[0].velocity_y - collision[1].velocity_y
    p = Normal.x * dif_x + Normal.y * dif_y

    collision[0].velocity_x = collision[0].velocity_x - p * Normal.x
    collision[0].velocity_y = collision[0].velocity_y - p * Normal.y

    collision[1].velocity_x = collision[1].velocity_x + p * Normal.x
    collision[1].velocity_y = collision[1].velocity_y + p * Normal.y

collisions.clear()
```

다음은 충돌 처리이다. 이번에도 동일하게 충돌 처리 방식에 대해서 먼저 살펴보자.



$$v_A(t+1) = \frac{m_A - m_B}{m_A + m_B} v_A + \frac{2m_B}{m_A + m_B} v_B$$



$$\hat{\mathbf{n}} = \overrightarrow{AB}$$

$$p = \frac{2\left(\mathbf{v}_A \cdot \hat{\mathbf{n}} - \mathbf{v}_B \cdot \hat{\mathbf{n}}\right)}{m_A + m_B}$$

$$\mathbf{v}_A(t+1) = \mathbf{v}_A - pm_B\hat{\mathbf{n}}$$

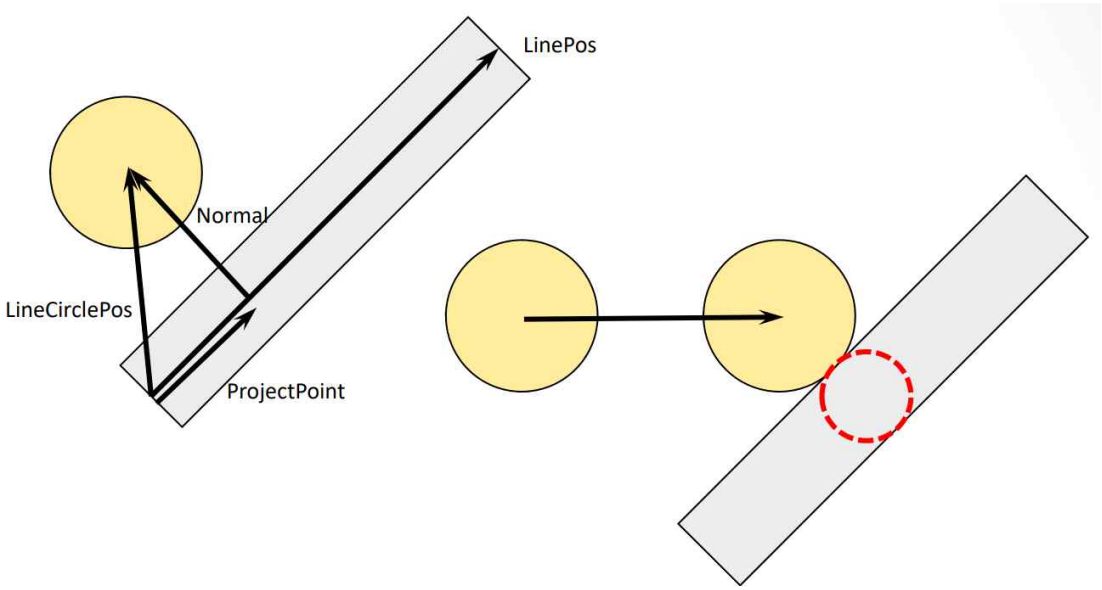
$$\mathbf{v}_B(t+1) = \mathbf{v}_B + pm_A\hat{\mathbf{n}}$$

$$\mathbf{v}(t+1) = \mathbf{v} - 2(\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$$

충돌은 크게 2가지 유형이 존재하는데 원과 원의 충돌과 원과 직선의 충돌이다. 여기서는 원과 원의 충돌에 대해서만 살펴볼 것이다. 두 원 객체는 완전 탄성 충돌한다. 운동량이 보존된다는 사실을 이용해서 충돌 이후에 두 객체의 속도를 확인할 수 있다. 위의 그림 속 수식에서 모든 원 객체의 질량은 동일하다고 가정할 수 있기 때문에 질량 관련 부분은 그냥 모두 1로 생각할 수 있다. 그럼 위의 코드처럼 구현할 수 있다.

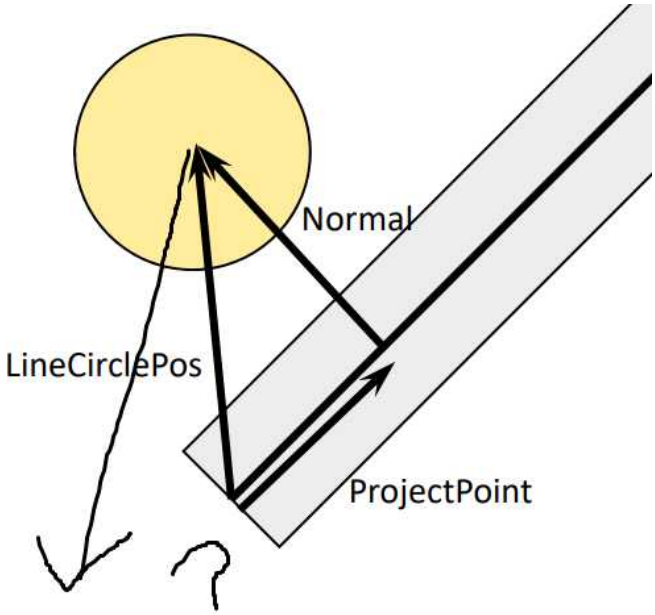
```
for circle in circles:
    for line in lines:
        line_ = vec_2d(line.end_position_x - line.start_position_x, line.end_position_y - line.start_position_y)
        pos_vec = vec_2d(circle.position_x - line.start_position_x, circle.position_y - line.start_position_y)
        pos_norm = line_.mult(1 / line_.abs())
        dot = line_.dot(pos_vec) / line_.abs()
        project = pos_norm.mult(dot)
        norm = pos_vec.sub(project)
        overlap = norm.abs() - 16
        abs = line_.abs()

        if dot > 0 and dot < abs:
            if overlap <= 0:
                mCircle = Circle(line.start_position_x + project.x, line.start_position_y + project.y, -circle.velocity_x, -circle.velocity_y, False)
                collisions.append([circle, mCircle])

                if norm.abs() == 0:
                    circle.position_x += 1
                    circle.position_y += 1

                else:
                    circle.position_x -= norm.mult(overlap / norm.abs()).x
                    circle.position_y -= norm.mult(overlap / norm.abs()).y
```

다음은 원과 직선의 충돌 판정이다. 이번에도 먼저 어떤 방식으로 충돌을 판정할 것인지 살펴보자.

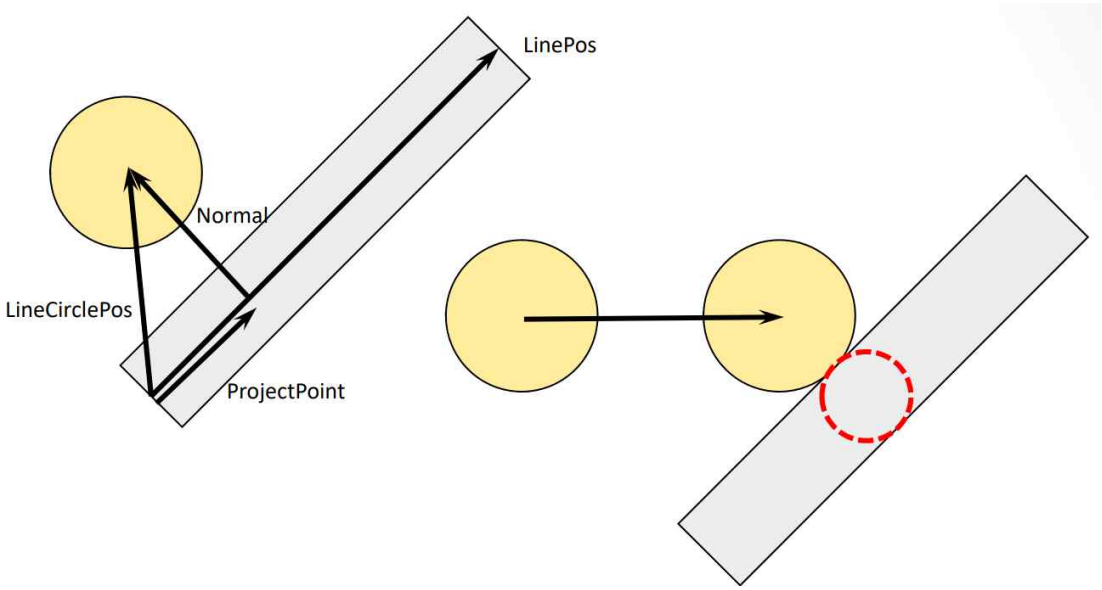


직선 객체는 start_point와 end_point를 가지고 있다. 먼저 이 start_point와 end_point를 잇는 벡터인 line_ 벡터를 구한다. 그리고 직선 객체의 start_point와 원 객체의 center_point를 잇는 벡터인 pos_vec을 구한다. 여기서 line_ 벡터를 normalize하여 단위 벡터로 만든 뒤에 pos_vec를 내적하면 직선 객체의 start_point에서 원 객체의 center_point에서 line_ 벡터에 내린 수선의 발을 잇는 벡터인 project 벡터를 구할 수 있다. 이후 pos_vec에서 project 벡터를 빼면 원 객체의 center_point에서 직선 객체에 내린 수선인 norm 객체를 최종적으로 구할 수 있다. 여기서 norm의 크기에서 원 객체의

radius와 직선 개체의 두께만큼을 뺀 값이 overlap이 음수가 된다면 원 객체와 직선 객체가 충돌했다고 '임시적'으로 볼 수 있다. 여기서 임시적으로 볼 수 있다는 의미는 다음과 같다.



만약 위의 그림처럼 원 객체가 직선 객체의 바깥쪽으로 통과했다고 생각해보자. 위에서 사용했던 충돌 판정 방식으로는 위의 상황도 충돌했다고 판정할 것이다. 이는 dot의 값이 음수이거나 line_ 벡터의 크기보다 크다면 아래 또는 위로 지나쳐가는 상황이기 때문에 이러한 상황은 충돌하지 않는다고 판정하여 이를 해결할 수 있다.



다음으로 1차적인 충돌 처리에 대해서 살펴보자. 위의 그림에서 오른쪽 그림처럼 수선의 발의 위치를 알고 있기 때문에 수선의 발의 위치를 center_point로 하고 radius를 직선 객체의

절반으로 하는 가상의 원으로 생성시켜 이를 이용해서 충돌 처리를 할 수 있도록 collisions에 원 객체와 가상의 원을 순서쌍으로 추가한다.

또한 원 객체들 간의 충돌과 동일하게 겹쳐진 두 객체를 서로 밀어내야 한다. 하지만 여기서 원 객체들 간의 충돌과 다른 점은 직선 객체는 움직이지 않는 static 객체이기 때문에 원만 밖으로 밀어내야 한다. 따라서 overlap만큼 밀어내어 코드처럼 처리한다.

```
if pygame.mouse.get_pressed()[0]:
    for circle in circles:
        x = pygame.mouse.get_pos()[0] - circle.position_x
        y = pygame.mouse.get_pos()[1] - circle.position_y

        if x > -15 and x < 15 and y > -15 and y < 15:
            circle.static = False

for circle in circles:
    if not circle.static:
        if pygame.mouse.get_pressed()[0]:
            circle.velocity_x = pygame.mouse.get_pos()[0] - circle.position_x
            circle.velocity_y = pygame.mouse.get_pos()[1] - circle.position_y
            circle.static = True
```

마지막으로 마우스를 이용하여 원 객체들을 이동시킬 수 있는 기능을 추가해보자. 기본적으로 pygame.mouse 모듈에서 마우스의 어떤 버튼이 눌렸는지 알 수 있는 get_pressed() 메소드와 마우스의 화면 상 위치를 가져오는 get_pos() 메소드를 이용하여 구현할 것이다. 먼저 매 update 주기마다 모든 원 객체에 대해서 객체가 클릭되었는지를 확인한다. 이후 클릭된 객체들에 대해서 위치 값이 아닌 속도값을 바꿔서 객체가 속도를 유지한 채로 이동할 수 있도록 한다.

4. Executing Environment

pygame 버전은 2.5.2이고 부가적으로 설치해야 하는 모듈은 없다. 단순히 실행하기만 하면 된다.