2020105599 김영원

구현 목표

게임엔진에서 물리적인 작동을 구현하기 위해 그와 관련된 Collision Detection, Rigid Body Dynamics, Rotation을 구현하려고 한다. 위의 세 가지는 서로 충분히 상호작용을 하는 분야이기 때문에 특히 중요하다고 생각되어 이러한 물리엔진 구현을 목표로 잡고 개발을 진행하였다.

게임 엔진 디자인 및 구조

게임엔진 디자인을 위해 크게 하나의 부모 클래스와 두 개의 자식 클래스를 만들기로 생각했다. 원과 사각형 같은 기초적인 도형을 이용하여 물리 엔진을 디자인하는 것이 구현 난이도도 내려가고 얼마나 구현했는지 쉽게 알 수 있을 것이라 생각했다. Object라는 부모 클래스에는 다른 자식 클래스가 가져야 할 RigidBody, Rotation, Collision Detection 등이 있고 자식 클래스인 Circle과 Rect는 각각 자신이 가지는 고유한 값(Rect의 경우 네 꼭짓점의 위치)를 갖게 하기로 했다.

모든 오브젝트에는 속도와 힘, 중력을 가지고 있으며 x축방향과 y축방향의 값을 모두 가질 수 있게 만들었으나 기본적으로 중력은 y축 하단 방향(파이게임 좌표계 기준 y가 양의 값)으로 힘을 받게 설정했다.

코드 설명

```
class Object
class Circle(Object)
class Rect(Object)
```

기본적으로 세 개의 클래스를 만들었다.

```
def colisionDetect_AABB(origin : Object, other : Object)
```

또한 잠시 테스트용으로 만들었다가 object 클래스 내부에 집어넣는 걸 잊고 너무 많은 기능이 생겨나버린 비운의 함수 colisionDetect_AABB라는 함수 역시 존재한다. 함수 이름에 오타가 있지만 이미 너무 많은 곳에 집어넣어서 바뀌기는 요원하다.

```
class Object :
    x_pos = 0
    y_pos = 0
    mass = 1
    x_max_min = [0,0]
    y_max_min = [0,0]
```

```
    width = 0
    height = 0
    gravity_force = [0, 1]
    force = [0,0]

    rotate_value = 0
    is_rotated = False #True 일 경우 회전중임
    fix_rotate = True # False 일 경우 어떠한 경우에도 회전하지 않음

    velocity = [0,0]
    max_velocity = [10,10]
    is_gravity = False # True 일 경우 중력 활성화
    active_rigidbody = False # True 일 경우 리지드바디 활성화

    active_elastic_collision = False #True 일 경우 탄성충돌

    colision_dectect_aabb = False #True 일 경우 aabb 충돌 판정
    colision_detect_obb = False #미구현
```

오브젝트 클래스 내부 변수들은 이렇다. init에서 초기화하기 전에 확인하기 쉽기 위해서 미리 작성해두었다.

```
  def __init__(self) :
        globalObjectList.append(self)
        self.active_elastic_collision = False
        self.gravity_force = [0,1]
        self.force = [0,0]
        self.is_move = True
        self.fixed_object = False
        return

    def __init__(self, x_pos, y_pos) :
        globalObjectList.append(self)
        self.active_elastic_collision = False
        self.gravity_force = [0,1]
        self.force = [0,0]
        self.x_pos = x_pos
        self.y_pos = y_pos

        self.is_move = True
        self.fixed_object = False
        return True

    def Position(self) :
        #자식 class 에게 넘기는 용
        #자식클래스에서는 무게중심을 반환
```

```
        return 0
```

위는 init함수와 position함수이다. Init의 경우 기본적인 것들을 선언해준다. 생성되면 전역변수인 globalObjectList에 본인을 추가한다.

```
    def RotateStart(self, radian, is_collied) :
        if (self.fix_rotate) == True :
            return
        if (self.is_rotated == False) :
            if (is_collied != -1) :
                self.is_rotated = True
                self.Rotate(radian)
        else :
            if (is_collied == -1) :
                self.Rotate(self.rotate_value)
            else :
                self.Rotate(radian)

    def Rotate(self, radian) :
        self.rotate_value = radian
        angle = math.radians(radian)
        sin, cos = np.sin(angle), np.cos(angle)
        rotation_matrix = np.array([(cos, -sin),(sin, cos)])
        self.originXY = self.originXY @ rotation_matrix

        return self.originXY
```

회전함수는 다음과 같다. RotateStart의 경우 회전각 라디안과 감지되었는지를 알려주는 변수를 인자로 받는다. fix_rotate가 True인 경우 회전할 필요가 없어 즉시 탈출하며, 그렇지 않은 경우 지금 회전중인가 아닌가에 따라 회전중을 의미하는 변수 값을 바꾸며 회전을 시작한다. 본인이 회전중이고 충돌이 없다면 본인의 rotate_value에 의해 회전을 계속하고, 충돌한 상태라면 충돌에서 얻은 rotate_value(radian)에 따라 회전을 결정한다.

Rotate 함수의 경우 radian 값을 받아서 본인의 rotate_value를 새롭게 갱신하고 회전을 시작한다.

```
    def RigidBody(self, target) :
        #실행이 안되는 조건
        if (target == -1) : # 만약 target 이 없으면
            return False
```

```
        if (self.active_rigidbody == False) or (target.active_rigidbody ==
False) : #타겟이나 자신이 리지드바디 혀옹을 안 하면
            return False

        velo = self.velocity

        #허용한다면
        if (self.active_elastic_collision) : #탄성충돌 ( 무한정 충돌)
            if (target.fixed_object == True) :
                self.velocity = [-self.velocity[X_POS], -self.velocity[Y_POS]]
                target.velocity = [0,0]

                return True

            self.velocity = [target.velocity[X_POS], target.velocity[Y_POS]]
            target.velocity = [velo[X_POS], velo[Y_POS]]
            return True

        #비탄성충돌

        if (target.fixed_object == True) or (self.fixed_object == True) :
            self.is_move = False
            self.velocity = [0,0]
            target.velocity = [0,0]
            return True

        if (self.is_move == False) :
            target.is_move = False
            target.y_pos -= self.height/2
            return True

        elif (target.is_move == False) :
            self.y_pos -= target.height/2
            self.is_move = False

        x_velo , y_velo = (target.velocity[X_POS] + velo[X_POS])/ 2,
(target.velocity[Y_POS] + velo[Y_POS]) / 2
        self.velocity = [x_velo, y_velo]
        target.velocity = [x_velo, y_velo]



        return True
```

리지드 바디의 경우 본인이 더 이상 움직이지 않는다면 타겟 역시 그러한 상태로 만든다. 비탄성충돌의 경우 서로 만나면 각각의 속도 성분을 더해 반으로 나눈 값을 서로 갖고, 완전탄성충돌의

경우에는 서로 부딛히면 서로의 각 성분을 바꾸어 움직인다.

```
    def Move(self) :

        if (self.is_move == False) :
            return 0
        if self.fixed_object == True :
            return 0
        #
        self.velocity[X_POS] += self.force[X_POS]
        self.velocity[Y_POS] += self.force[Y_POS]

        # 중력 관련해서 업데이트
        if (self.is_gravity == True) :
            if(useGlobalGravity == True) :
                self.velocity[X_POS] += global_gravity[X_POS]
                self.velocity[Y_POS] += global_gravity[Y_POS]
            else :
                self.velocity[X_POS] += self.gravity_force[X_POS]
                self.velocity[Y_POS] += self.gravity_force[Y_POS]

        if self.velocity[0] > self.max_velocity[0] :
            self.velocity[0] = self.max_velocity[0]

        if self.velocity[1] > self.max_velocity[1] :
            self.velocity[1] = self.max_velocity[1]



        self.x_pos += self.velocity[X_POS]
        self.y_pos +=  self.velocity[Y_POS]


        return self.velocity
```

Move 함수의 경우에는 is_move가 True인 경우에만 작동한다. 힘과 중력을 계산하고, 만약 속도가 미리 정한 값 이상으로 넘어가버린다면 최댓값으로 제한시키고 좌표를 갱신한다.

```
    def Update(self) :
        if(self.colision_dectect_aabb or self.colision_detect_obb) :
            for i in globalObjectList[globalObjectList.index(self):] :
                if ( i == self) :
                    self.RotateStart(self.rotate_value, -1)
                if (i != self) :
                    other = self.colisionDetect(i)
                    self.RigidBody(other[1])
                    self.RotateStart(other[0], other[1])
```

업데이트의 경우 글로벌 오브젝트 리스트에서 본인의 위치부터 시작해 본인 이후의 오브젝트들에게 물리 연산을 시작한다.

```
def colisionDetect_AABB(origin : Object, other : Object) :
    MAX = 0
    MIN = 1
    if ((origin.x_max_min[MAX] >= other.x_max_min[MIN] and
origin.x_max_min[MIN] < other.x_max_min[MAX]) and (origin.y_max_min[MAX] <
other.y_max_min[MIN] and origin.y_max_min[MIN] > other.y_max_min[MAX] )) :

        if (origin.Position()[0] == other.Position()[0] or
origin.Position()[1] == other.Position()[1]) :
            return (0, other)

        ori_vel_vec = np.array([origin.velocity[0],
origin.velocity[1]])
        other_vel_vec = np.array([other.velocity[0],
other.velocity[1]])
        ori_to_other_vec = other_vel_vec - ori_vel_vec

        costheta = np.dot(ori_vel_vec, ori_to_other_vec)

        costheta = costheta/(np.linalg.norm(ori_vel_vec) +
np.linalg.norm(ori_to_other_vec))

        angle = costheta

        other.RotateStart(-angle, 1)

        return (angle, other)
    return (0, -1)
```

잠깐 비운의 함수 콜리전디텍트를 살펴보자면 AABB를 이용하여 충돌판정을 실시한다. 회전을 위해서 두 충돌물체의 각각의 무게중심을 계산하고, 무게중심까지 이은 벡터와 원래 이동하는 벡터를 내적을 통해 회전방향과 회전양을 결정한다. 그리고 다른 부딪힌 물체 역시 회전을 시작시킨다. 회전하는 값 세타와 회전을 하는지 안 하는지를 알려주는 1과 -1을 튜플 형태로 반환한다.

```
class Rect(Object) :
    def __init__(self, x_pos, y_pos, width, height, color = (255, 255, 255)) :
        super().__init__(x_pos, y_pos)
        self.width = width
        self.height = height
        self.color = color
        self.x_max_min = [x_pos + width, x_pos]
```

```
        self.y_max_min = [y_pos, y_pos + height]
        self.velocity = [0,0]
        self.force = [0,0]
        self.mass = 1
        self.originXY =np.array([(-width/2, -height/2),
                                 (width/2, -height/2),
                                 (width/2, height/2),
                                (-width/2, height/2)])
        self.location = (x_pos, y_pos)
        self.is_rotated = False
        self.fix_rotate = True

        self.now_position = self.originXY + self.location
    =
```

여기서 볼 점은 회전연산을 위해서 originXY라는 함수가 각 꼭짓점을, location이란 함수가 x,y의 위치를, 그리고 그 둘을 더한 현재 위치 변수가 있다는 것이다. 위에서 본 Rotate에서 위의 origin을 회전시키고 난 뒤에 location을 더해서 회전이 갱신된다.

```
    def Move(self):
        super().Move()

        if (self.is_move == False) :
            return 0
        if self.fixed_object == True :
            return 0

        if (self.x_pos + self.width/2 >= globalWidth) :
            self.x_pos = globalWidth - self.width/2
            if (self.active_elastic_collision == True) :
                self.velocity[X_POS] = -self.velocity[X_POS]
            else :
                self.is_move = False

        elif ((self.x_pos - self.width/2 <= 0)) :
            self.x_pos = self.width/2
            if (self.active_elastic_collision == True) :
                self.velocity[X_POS] = -self.velocity[X_POS]
            else :
                self.is_move = False



        if (self.y_pos + self.height/2 >= globalHeight) :
            self.y_pos = globalHeight - self.height/2
            if (self.active_elastic_collision == True) :
                self.velocity[Y_POS] = -self.velocity[Y_POS]
            else :
```

```
                self.is_move = False
        elif (self.y_pos - self.height/2 <= 0) :
            self.y_pos = self.height/2
            if (self.active_elastic_collision == True) :
                self.velocity[Y_POS] = -self.velocity[Y_POS]
            else :
                self.is_move = False

    def Update(self):
        super().Update()
        self.Move()
        self.location = (self.x_pos, self.y_pos)
        self.now_position = self.originXY + self.location
        self.x_max_min = [self.x_pos + self.width/2, self.x_pos -
self.width/2]
        self.y_max_min = [self.y_pos - self.height/2, self.y_pos +
self.height/2]
```

move와 업데이트의 경우는 초기 pygame으로 사각형을 그리다가 실패해서 Object와 합병을 실패한 흔적이 남아있다. Move의 경우가 그런데, object에 남겨둬도 괜찮은 것처럼 보인다.

업데이트의 경우 부모함수에서 업데이트시키고, Rect의 Move 역시 업데이트시킨다. 또한 위치 정보를 동기화시킨다.

개발 환경

Python 3.11.4

Pygame 2.5.1