

Project 3

Develop Your Own Physics Engine

소프트웨어융합학과

2022105503

한지우

1. 수업 및 실습에서 다룬 내용과 직접 구현한 내용 및 기여사항

이번 팀 프로젝트에서는 Pygame 프레임워크를 기반으로 “Visibility Polygon Engine(가시 다각형 엔진)”을 구현했습니다. 이 엔진은 플레이어의 위치와 바라보는 방향, 그리고 게임 월드에 배치된 여러 개의 벽 정보를 입력으로 받아, 플레이어가 실제로 어떤 영역을 시야로 확보하고 있는지를 다각형 형태로 계산·표현하는 역할을 합니다. 구현 과정에서 수업 시간에 다루었던 벡터 연산, 선분 교차 판정, 운동학(Kinematics) 내용들을 바탕으로 삼았고, 여기에 가시 다각형 생성 알고리즘과 동적 환경 처리라는 심화 요소를 결합하여 하나의 엔진 형태로 구조화했습니다.

1) 수업 및 실습에서 배운 기능·알고리즘을 활용한 부분

먼저, 엔진의 수학적 기반을 담당하는 Vec2 클래스를 직접 구현했습니다. 이 클래스는 2차원 벡터의 덧셈, 뺄셈, 스칼라 곱, 내적(dot), 외적(cross), 길이 계산, 정규화(normalize), 각도 추출(atan2) 기능을 제공합니다. 이는 수업에서 학습했던 벡터 연산 공식을 코드 차원에서 재구현한 것으로, 이후 레이 방향 계산, 충돌 지점 계산, 회전 변환 등 엔진의 거의 모든 부분에서 사용되는 핵심 기반이 되었습니다. 특히 외적 연산은 레이-선분 교차 판정에서 분자/분모를 계산할 때 직접 사용했습니다.

```
4 # =====
5 # Math / Geometry Utilities
6 # =====
7
8 class Vec2:
9     __slots__ = ("x", "y")
10
11     def __init__(self, x=0.0, y=0.0):
12         self.x = float(x)
13         self.y = float(y)
14
15     def __add__(self, other): # 벡터의 덧셈
16         return Vec2(self.x + other.x, self.y + other.y)
17
18     def __sub__(self, other): # 벡터의 뺄셈
19         return Vec2(self.x - other.x, self.y - other.y)
20
21     def __mul__(self, scalar): # 벡터의 곱셈
22         return Vec2(self.x * scalar, self.y * scalar)
23
24     __rmul__ = __mul__
```

```
26 def dot(self, other): # 내적 계산
27     return self.x * other.x + self.y * other.y
28
29 def cross(self, other): # 외적 계산 (2D라서 스칼라)
30     return self.x * other.y - self.y * other.x
31
32 def length(self): # 벡터의 크기 계산
33     return math.hypot(self.x, self.y)
34
35 def normalized(self): # 벡터 정규화
36     l = self.length()
37     if l == 0:
38         return Vec2(0, 0)
39     return Vec2(self.x / l, self.y / l)
40
41 def tuple(self): # 튜플 변환
42     return (self.x, self.y)
43
44 def angle(self): # 각도 계산
45     return math.atan2(self.y, self.x)
46
47 @staticmethod
48 def from_angle(theta): # 각도를 입력받아 단위 방향 벡터 생성
49     return Vec2(math.cos(theta), math.sin(theta))
```

충돌 검출 부분에서는 수업에서 다루었던 직선/선분 교차 판정 알고리즘을 참고하여 ray_segment_intersection 함수를 작성했습니다. 레이는 $\text{origin} + t * \text{direction}$ 으로, 선분은 $a + u * (b - a)$ 로 표현하고, 두 식을 연립하여 t와 u를 구한 뒤, $t \geq 0$ 이면서 $0 \leq u \leq 1$ 인 경우에만 실제 교차가 일어나는 것으로 판정했습니다. 이 과정에서 분모가 0에 근접할 경우를 고려하여 평행한 경우를 예외 처리했습니다. 이러한 구현은 수업에서 배운 수식을 그대로 코드로 옮기면서도, 실시간 게임 환경에서 안정적으로 동작하도록 수치 오차를 신경 써서 작성했습니다.

```

256 def ray_segment_intersection(origin, direction, a, b):
257     # origin : 레이가 시작되는 점(플레이어 위치)
258     # direction : 레이의 방향 단위 벡터
259     # a, b : 선분의 두 끝점
260     # t : 레이 식 "origin + t * direction"에서의 이동량
261     # u : 선분 식 "a + u * (b - a)"에서의 상대 위치(0~1 범위일 때 교차)
262     # origin + t * direction = a + u*(b-a)
263     v1 = origin - a
264     v2 = b - a
265     denom = direction.cross(v2)
266     if abs(denom) < 1e-6:
267         return None
268     t = v2.cross(v1) / denom
269     u = direction.cross(v1) / denom
270     if t < 0:
271         return None
272     return t, u

```

또한, 플레이어의 이동과 일부 벽의 이동에는 Kinematics(운동학)을 적용했습니다. 플레이어의 위치는 $\text{position} = \text{position} + \text{velocity} * dt$ 공식을 사용해 매 프레임 갱신했고, 이동하는 벽의 경우도 마찬가지로 속도 벡터를 적용해 실시간으로 위치가 변하도록 구현했습니다. 회전하는 벽에 대해서는 수업에서 다루었던 회전 변환을 사용하여, 회전 중심을 기준으로 각속도(angular velocity)를 누적한 뒤 cos, sin을 이용해 두 끝점을 회전시키도록 했습니다. 이를 통해, 단순히 정적인 선분이 아니라 시간에 따라 위치와 방향이 변하는 벽을 환경에 포함시킬 수 있었습니다.

```

68 class Wall:
69     # velocity : 벽의 선형 이동 속도
70     # angular_speed : 벽의 각속도(라디안/초)
71     # rotation_center : 회전의 기준점
72     # self.angle : 프레임마다 누적되는 회전값
73     def __init__(self, p1, p2, dynamic=False, rotation_center=None, angular_speed=0.0, velocity=None):
74         self.base_p1 = Vec2(p1.x, p1.y)
75         self.base_p2 = Vec2(p2.x, p2.y)
76         self.segment = Segment(Vec2(p1.x, p1.y), Vec2(p2.x, p2.y))
77
78         #Dynamics
79         self.dynamic = dynamic
80         self.rotation_center = rotation_center # 벡터2 혹은 없음
81         self.angular_speed = angular_speed # radians per second
82         self.velocity = velocity or Vec2(0, 0) # 속도
83         self.angle = 0.0
84
85     def update(self, dt):
86         if not self.dynamic:
87             return
88
89         # 회전
90         if self.rotation_center is not None and self.angular_speed != 0.0:
91             self.angle += self.angular_speed * dt
92             # 회전 중심점을 기준으로 베이스 포인트를 회전시킨다
93             self.segment.p1 = rotate_around(self.base_p1, self.rotation_center, self.angle)
94             self.segment.p2 = rotate_around(self.base_p2, self.rotation_center, self.angle)
95
96         # 평행 이동
97         if self.velocity.x != 0 or self.velocity.y != 0:
98             dp = self.velocity * dt
99             self.segment.p1 = self.segment.p1 + dp
100             self.segment.p2 = self.segment.p2 + dp
101
102     def get_segment(self):
103         return self.segment

```

시야각(FOV) 판정 역시 수업 내용을 바탕으로 했습니다. 플레이어가 바라보는 방향 벡터와 후보 점(벽의 꼭짓점) 방향 벡터 사이의 각도 차이를 구할 때, 두 벡터의 내적을 활용하면 코사인 값을 통해 각도를 계산할 수 있다는 점을 수업에서 배웠습니다. 저는 이 개념을 조금 단순화하여, 각도 그 자체는 atan2 로 얻고, 두 각의 차이를 angle_diff 함수로 정규화한 다음, 이 차이가 미리 설정한 FOV의 절반보다 작은 경우에만 시야 내에 있는 각도로 간주하도록 구현했습니다. 이로써, 수업에서 실습했던 “내적 기반 시야 판정” 개념을 제 엔진의 핵심 입력 필터로 활용했습니다.

```
class VisibilityEngine:
    def __init__(self, walls):
        # walls: list[Wall]
        self.walls = walls

    def compute_visibility_polygon(self, origin, facing_angle, fov_angle, full_360=False):
        # origin : 벡터 2 (플레이어 위치)
        # facing_angle : 방향 라디안
        # fov_angle : 시야각 라디안 (360도 모드 = False일 때만 사용)
        # 반환 (points, hit_walls)
        # points : list[Vec2], 가시 폴리곤
        # hit_walls : 모든 ray에 대하여 hit된 벽들 모음

        # 모든 벽 끝점 기준으로 후보 각도 생성
        for wall in self.walls:
            seg = wall.get_segment()
            for p in (seg.p1, seg.p2):
                rel = p - origin
                angle = math.atan2(rel.y, rel.x)
                angles.extend([angle - eps, angle, angle + eps])

        # 360도 모드면 전체 각도 샘플, 아니면 FOV 주변 추가 샘플
        if full_360:
            steps = 64
            for i in range(steps):
                a = -math.pi + 2 * math.pi * i / steps
                angles.append(a)
        else:
            steps = 32
            half = fov_angle / 2.0
            for i in range(steps + 1):
                t = -half + fov_angle * i / steps
                angles.append(facing_angle + t)

        # 정규화 + FOV 필터링
        unique_angles = []
        seen = set()
        for a in angles:
            # Normalize to [-pi, pi]
            a = math.atan2(math.sin(a), math.cos(a))
            key = round(a, 4)
            if key in seen:
                continue
            seen.add(key)
            if not full_360:
                # 시야각(FOV)을 facing_angle 주변으로 제한한다
                diff = angle_diff(a, facing_angle)
                if abs(diff) > fov_angle / 2.0:
                    continue
            unique_angles.append(a)
        unique_angles.sort()

        points = []
        hit_walls = set()

        for a in unique_angles:
            hit_point, hit_wall = self.cast_ray(origin, a)
            if hit_point is not None:
                points.append((a, hit_point))
                if hit_wall is not None:
                    hit_walls.add(hit_wall)

        if full_360:
            # 360도 모드는 그냥 절대각도로 정렬
            unique_angles.sort()
            ordered_angles = unique_angles
        else:
            # FOV 모드: 바라보는 각도 기준의 상대각(diff)으로 정렬
            angle_pairs = []
            for a in unique_angles:
                diff = angle_diff(a, facing_angle) # [-pi, pi] 범위의 상대각
                angle_pairs.append((diff, a))

            angle_pairs.sort(key=lambda x: x[0]) # diff 기준으로 정렬
            ordered_angles = [a for diff, a in angle_pairs]

        points = []
        hit_walls = set()

        for a in ordered_angles:
            hit_point, hit_wall = self.cast_ray(origin, a)
            if hit_point is not None:
                points.append((a, hit_point))
                if hit_wall is not None:
                    hit_walls.add(hit_wall)

        # Sort by angle and return positions only
        # points.sort(key=lambda ap: ap[0])
        poly_points = [p for _, p in points]

        return poly_points, hit_walls
```

2) 수업 기반 내용을 발전·심화시킨 본인의 기여사항

수업에서 다루었던 내용은 주로 “보인다 / 안 보인다”를 판정하는 단일 충돌 테스트 수준이었습니다. 저는 여기서 한 단계 더 나아가, 플레이어를 기준으로 실제로 보이는 공간 전체를 다각형으로 구성하는 Visibility Polygon Engine을 만들었습니다. 이를 위해 먼저, 월드에 존재하는 모든 벽의 각 끝점을 기준으로 후보 각도를 생

성했습니다. 각 끝점에 대해 $\text{angle} = \text{atan2}(y - \text{origin.y}, x - \text{origin.x})$ 를 계산하고, 각도 주변에 작은 ϵ 를 더한 값과 뺀 값까지 포함시켜 벽 모서리를 정확히 통과하는 레이를 추가했습니다. 이렇게 생성된 각도들에 대해 레이를 쏘고, 각 레이가 가장 먼저 맞는 교차점을 찾아 모으면, 이 교차점들이 곧 “보이는 영역의 외곽”을 형성하게 됩니다.

```
# 모든 벽 끝점 기준으로 후보 각도 생성
for wall in self.walls:
    seg = wall.get_segment()
    for p in (seg.p1, seg.p2):
        rel = p - origin
        angle = math.atan2(rel.y, rel.x)
        angles.extend([angle - eps, angle, angle + eps])
```

여기서 문제가 된 부분은 각도 정렬과 시야각(FOV) 처리였습니다. 절대 각도만 단순히 정렬하면, 플레이어가 화면 왼쪽 방향(π rad 근처)을 바라볼 때 FOV 범위가 π 를 기준으로 좌우로 갈라져서, 정렬 순서가 끊어져 보이는 문제가 발생했습니다. 이를 해결하기 위해, 저는 `angle_diff`라는 함수를 작성하여, 모든 각도를 “현재 바라보는 각도(`facing_angle`)를 기준으로 한 상대각”으로 변환한 뒤, 이 상대각을 기준으로 정렬하는 방식을 도입했습니다. 이렇게 하여 항상 왼쪽 끝 레이 → 중앙 → 오른쪽 끝 레이 순서가 유지되도록 만들었고, 그 결과 FOV 모드에서도 부채꼴 형태의 시야 다각형이 안정적으로 그려지도록 했습니다.

```
def angle_diff(a, b):
    # 두 각도 사이의 가장 짧은 방향(양수/음수)으로의 차이 [-pi, pi]
    diff = a - b
    while diff <= -math.pi:
        diff += 2 * math.pi
    while diff > math.pi:
        diff -= math.pi * 2
    return diff
```

```
if full_360:
    # 360도 모드는 그냥 절대각도로 정렬
    unique_angles.sort()
    ordered_angles = unique_angles
else:
    # FOV 모드: 바라보는 각도 기준의 상대각(diff)으로 정렬
    angle_pairs = []
    for a in unique_angles:
        diff = angle_diff(a, facing_angle) # [-pi, pi] 범위의 상대각
        angle_pairs.append((diff, a))

    angle_pairs.sort(key=lambda x: x[0]) # diff 기준으로 정렬
    ordered_angles = [a for diff, a in angle_pairs]
```

또한, 시야각 모드에서 시각적으로 자연스러운 표현을 위해, 가시 다각형을 그릴 때 단순히 레이의 교차점들만 이어서 채우는 것이 아니라 플레이어 위치를 다각형의 첫 꼭짓점으로 포함하도록 그리기 방식을 수정했습니다. 이를 통해 “플레이어 → 레이 교차점들 → 다시 플레이어”로 이어지는 부채꼴 형태가 만들어졌고, 실제 게임에서 흔히 볼 수 있는 시야 콘(FOV cone)과 유사한 표현이 가능해졌습니다.

```
# 가시 다각형 그리기 (반투명)
if len(poly_points) >= 2:
    poly_surface = pygame.Surface((self.WIDTH, self.HEIGHT), pygame.SRCALPHA)
    if self.full_360:
        # 360도 모드: 기존처럼 그대로 사용
        draw_points = [p.tuple() for p in poly_points]
    else:
        # FOV 모드: 플레이어를 꼭짓점으로 포함해서 부채꼴 모양으로 그림
        draw_points = [self.player_pos.tuple()] + [p.tuple() for p in poly_points]

    pygame.draw.polygon(
        poly_surface,
        (255, 255, 100, 60),
        draw_points
    )
    self.screen.blit(poly_surface, (0, 0))
```

심화 기능으로는 두 가지 시야 모드(FOV/360도) 지원과 동적 환경 처리를 구현했습니다. FOV 모드에서는 플레이어가 바라보는 방향과 시야각을 기준으로 제한된 각도 범위에만 레이를 발사했고, 360도 모드에서는 전체 방향으로 레이를 균일하게 샘플링하여 플레이어를 둘러싼 전체 가시 영역을 계산했습니다. 하나의 compute_visibility_polygon 함수 내부에서 두 모드를 모두 처리할 수 있도록 설계하여, 모드 전환 시에도 동일한 엔진 로직이 재사용되도록 했습니다.

```
# 360도 모드면 전체 각도 샘플, 아니면 FOV 주변 추가 샘플
if full_360:
    steps = 64
    for i in range(steps):
        a = -math.pi + 2 * math.pi * i / steps
        angles.append(a)
else:
    steps = 32
    half = fov_angle / 2.0
    for i in range(steps + 1):
        t = -half + fov_angle * i / steps
        angles.append(facing_angle + t)
```

```
if full_360:
    # 360도 모드는 그냥 절대각도로 정렬
    unique_angles.sort()
    ordered_angles = unique_angles
```

마지막으로, 가시 다각형에 실제로 의해 감지된 벽을 시각적으로 하이라이트하는 기능을 추가했습니다. 각 레이가 어느 벽과 처음으로 교차했는지를 추적하여, 최소 한 번이라도 레이의 대상이 된 벽은 다른 색으로 그리도록 했습니다. 이 기능을 통해, 단순히 “영역”만 보여주는 것을 넘어서 엔진이 어떤 객체를 실제로 인식하고 있는지를 직관적으로 확인할 수 있게 했습니다. 이는 디버깅과 시연(데모) 측면에서 모두 유용한 기능으로, 프로젝트 발표 시 엔진의 동작을 설명하는 데 큰 도움이 되도록 의도했습니다.

```
def update(self, dt): # Scene.update
    # 벽 업데이트 + 단순 바운스 처리
    for wall in self.walls:
        wall.update(dt)
        if wall.dynamic and wall.velocity.length() > 0:
            seg = wall.get_segment()
            for p in [seg.p1, seg.p2]:
                if p.x < 80 or p.x > 720:
                    wall.velocity.x *= -1
                if p.y < 80 or p.y > 520:
                    wall.velocity.y *= -1
```

```
def update(self, dt): # Wall.update
    if not self.dynamic:
        return

    # 회전
    if self.rotation_center is not None and self.angular_speed != 0.0:
        self.angle += self.angular_speed * dt
        # 회전 중심점을 기준으로 베이스 포인트를 회전시킨다
        self.segment.p1 = rotate_around(self.base_p1, self.rotation_center, self.angle)
        self.segment.p2 = rotate_around(self.base_p2, self.rotation_center, self.angle)

    # 평행 이동
    if self.velocity.x != 0 or self.velocity.y != 0:
        dp = self.velocity * dt
        self.segment.p1 = self.segment.p1 + dp
        self.segment.p2 = self.segment.p2 + dp
```

종합하면, 저는 수업에서 배운 충돌 판정, 벡터/각도 연산, 운동학 내용을 토대로 Ray-Casting 기반의 Visibility Polygon Engine을 직접 구현했고, 각도 정렬 문제 해결, FOV/360 모드 지원, 동적 벽 처리, 하이라이트 기능과 같은 요소들을 추가하여 수업 내용을 실제 게임 엔진 수준의 심화 기능으로 발전시켰습니다.

2. 엔진 기술 개발 중요성과 필요성

이번 프로젝트에서 구현한 Visibility Polygon Engine은 단순한 시각적 효과를 넘어서, 실제 게임 엔진 및 시뮬레이션 환경에서 핵심적으로 사용되는 기능을 축소 구현한 것입니다. 물체의 시야와 가시성을 계산하는 기능은 충돌 검출, AI 의사결정, 빛과 그림자 처리, 플레이어 경험 설계 등 다양한 게임 시스템의 기반을 이루기 때문에 물리 엔진의 중요한 구성 요소라고 판단했습니다. 이러한 이유로 본 과제에서는 수업 시간에 배운 수학적 기반과 충돌 판정 기법을 활용하여, 실제 게임 엔진에서 사용하는 시야 계산 방식에 가까운 형태로 기능을 발전시켜 구현했습니다.

1) 본인이 선택한 기능의 물리 엔진으로서의 중요성

가시성(Visibility) 계산은 게임에서 매우 본질적인 요소입니다. 예를 들어, AI가 플레이어를 인지하는지 여부, 적 캐릭터가 들키지 않고 이동해야 하는 스텔스 게임의 시야 범위, 실시간 전략 게임의 안개(Fog of War) 시스템, 자유 시점 카메라 기반 시뮬레이션에서의 충돌 예측 등에서는 필연적으로 “어떤 객체가 어느 시점에서 보이는가?”를 정확히 판단해야 합니다. 이러한 기능은 대부분 레이캐스팅(ray casting) 또는 충돌 기반 시야 검출 기법을 사용하여 구현되며, 이는 물리 엔진 내부에 포함되는 중요한 기능입니다.

이번에 구현한 Visibility Polygon Engine은 단순히 한 점이 보이는지(True/False)를 판단하는 데서 그치지 않고, 현재 프레임에서 플레이어가 실제로 확보한 시야 영역 전체를 다각형 형태로 계산하여 제공하는 구조입니다. 이는 실제 게임 엔진이 광원, 그림자, 시야, 오브젝트 인식 등을 처리할 때 사용하는 알고리즘과 동일한 원리를 기반으로 하기 때문에, 게임 엔진 요소 중에서도 매우 실용적이며 중요한 기술이라고 할 수 있습니다. 특히 시야가 벽에 의해 가려지는지 여부를 정확히 반영해야 하므로, 이는 충돌 검출과 물리 계산이 자연스럽게 통합된 엔진 기능입니다.

또한 움직이는 벽이나 회전하는 벽처럼 환경이 동적으로 변화하는 상황에서도 시야를 정확히 계산하기 위해서는 매 프레임마다 벽의 새로운 위치를 반영하고, 모든 레이를 다시 계산해야 합니다. 이를 신속하고 정확하게 처리하는 기능은 충돌 기반 시뮬레이션이 요구되는 물리 엔진의 핵심 역할 중 하나입니다.

2) 발전시킨 방향의 필요성과 의미

수업에서는 주로 정적인 문제(고정된 선분 충돌, 단순 시야 판정, 정해진 방향의 회전 등)를 다루었습니다. 하지만 실제 게임 환경에서는 이러한 요소들이 끊임없이 변하기 때문에, 이를 처리할 수 있는 엔진 설계가 필요했습니다. 이번 프로젝트에서는 수업 내용을 기반으로 다음과 같은 확장과 발전을 수행했습니다.

첫째, 단일 충돌 검출 → 다중 레이 기반의 가시영역(Polygon) 생성으로 업그레이드했습니다. 단순히 “보인다 / 안 보인다”가 아니라, 플레이어 주변의 실제 시야 경계를 구성하는 모든 교차점을 계산하고 정렬하여, 하나의 다각형으로 연결하는 과정을 구현했습니다. 이것은 실제 게임 엔진에서 Shadow Casting 또는 FOV(field-of-view) 알고리즘으로 알려진 방식이며, 수업에서 배우지 않은 내용을 확장하여 적용하였습니다. 이를 통해 단순 판단에서 벗어나 실제 공간 구조를 반영한 시야 엔진을 구축할 수 있었습니다.

둘째, 동적 환경(dynamic environment)을 고려해 엔진을 설계했습니다. 벽이 회전하거나 이동할 수 있도록 구현하고, 매 프레임마다 이러한 변화를 반영해 시야 영역을 재계산하도록 했습니다. 이 기능은 기본적인 충돌 판정을 넘어, 물리 기반 시뮬레이션에서 발생하는 동적 오브젝트 처리 문제를 직접 해결했다는 점에서 의미가 있습니다. 이를 통해 엔진은 정적인 환경뿐만 아니라 실제 게임 환경과 유사한 상황까지 다룰 수 있게 발전했습니다.

셋째, FOV 모드와 360도 모드를 모두 지원하도록 설계했습니다. 이를 위해 각도 wrap-around 문제($-\pi$ 와 $+\pi$ 경계 문제)를 해결하는 정렬 알고리즘을 직접 제작했고, 이를 통해 바라보는 방향 중심으로 안정적인 시야 다각형을 생성할 수 있었습니다. 이 알고리즘은 수업에서 제공된 코드에서 추가적으로 엔진 개발 과정에서 마주치는 수학적 오류를 분석하고 해결하였습니다.

마지막으로, 엔진의 내부 동작을 시각적으로 이해할 수 있도록 가시 영역에 실제로 감지된 벽을 하이라이트하는 기능을 추가했습니다. 이는 엔진의 디버깅에 도움이 될 뿐만 아니라, 물리 엔진이 어떤 방식으로 환경을 인식하고 처리하는지를 시각적으로 설명하는 데 중요한 역할을 했습니다. 엔진 구조 자체를 모듈화하고 재사용 가능하게 설계함으로써, 향후 충돌 엔진이나 AI 인식 엔진 등 더 복잡한 시스템으로도 확장 가능하도록 기반을 마련했습니다.

3. 엔진 기술 개발 기본 구현 (20%)

이번 프로젝트의 기본 구현 단계에서는 수업 시간에 배운 핵심 개념들을 기반으로, 가시성 엔진이 정상적으로 동작하기 위해 필요한 가장 근본적인 기능들을 직접 구현했습니다. 이 단계는 엔진의 기초를 구성하는 부분으로서, 벡터 연산, 레이-선분 충돌 판정, 시야각 계산, 객체의 기초적인 이동 처리 등 물리 엔진이 갖추어야 하는 기본 요소를 충실하게 구현하는 과정이었습니다.

1) 벡터 연산을 위한 Vec2 클래스 구현

-코드 위치 : 파일 상단 class Vec2

물리 엔진의 모든 기능은 벡터 연산을 기반으로 이루어지기 때문에, 먼저 2차원 벡터를 다루기 위한 Vec2 클래스를 작성했습니다. 이 클래스는 물리 연산에서 필수적인 기능들을 제공합니다.

주요 메서드는 다음과 같습니다:

- `__add__`, `__sub__` : 위치 벡터 간 이동 및 변위 계산
- `dot()` : 내적 연산으로 두 벡터의 방향 유사도 계산
- `cross()` : 외적(2D에서는 스칼라)로 레이-선분 교차 판정에 사용
- `length()` / `normalized()` : 벡터의 크기와 단위 방향 벡터 계산
- `from_angle()` : 극좌표(각도)를 직교좌표계 벡터로 변환

가시 엔진에서는 레이 방향 벡터, 벽 교차점 계산, 플레이어 이동 방향 계산 등 대부분의 연산에 Vec2가 사용되었으며, 이를 통해 물리 엔진의 기반이 되는 수학적 구조를 확립했습니다. 이 부분은 수업 중 Mathematics 파트에서 배운 벡터 연산 개념을 실제 코드로 구현한 것입니다.

```
4 # =====
5 # Math / Geometry Utilities
6 # =====
7
8 class Vec2:
9     __slots__ = ("x", "y")
10
11     def __init__(self, x=0.0, y=0.0):
12         self.x = float(x)
13         self.y = float(y)
14
15     def __add__(self, other): # 벡터의 덧셈
16         return Vec2(self.x + other.x, self.y + other.y)
17
18     def __sub__(self, other): # 벡터의 뺄셈
19         return Vec2(self.x - other.x, self.y - other.y)
20
21     def __mul__(self, scalar): # 벡터의 곱셈
22         return Vec2(self.x * scalar, self.y * scalar)
23
24     __rmul__ = __mul__
25
26     def dot(self, other): # 내적 계산
27         return self.x * other.x + self.y * other.y
28
29     def cross(self, other): # 외적 계산(2D라서 스칼라)
30         return self.x * other.y - self.y * other.x
31
32     def length(self): # 벡터의 크기 계산
33         return math.hypot(self.x, self.y)
34
35     def normalized(self): # 벡터 정규화
36         l = self.length()
37         if l == 0:
38             return Vec2(0, 0)
39         return Vec2(self.x / l, self.y / l)
40
41     def tuple(self): # 튜플 변환
42         return (self.x, self.y)
43
44     def angle(self): # 각도 계산
45         return math.atan2(self.y, self.x)
46
47     @staticmethod
48     def from_angle(theta): # 각도를 입력받아 단위 방향 벡터 생성
49         return Vec2(math.cos(theta), math.sin(theta))
```

2) Ray-Segment 교차 판정 기본 알고리즘 구현

-코드 위치 : `def ray_segment_intersection(origin, direction, a, b)`

가시성 엔진의 핵심 기능은 "특정 방향으로 레이를 쏘았을 때 어떤 벽과 먼저 충돌하는가"를 계산하는 것입니다. 이를 위하여 수업에서 배운 선분 교차 공식과 외적 기반의 판정 방식을 직접 사용하여 다음과 같은 형태로 구현했습니다.

사용된 주요 변수의 의미는 다음과 같습니다:

- origin : 레이가 시작되는 위치(플레이어 위치)
- direction : 단위 방향 벡터(각도를 기반으로 생성)
- a, b : 벽의 두 끝점
- t : 레이 식 $\text{origin} + t * \text{direction}$ 에서 이동 거리
- u : 선분 식 $a + u * (b - a)$ 에서의 상대적 위치

사용된 교차 조건은 다음과 같습니다:

- $\text{origin} + t * \text{direction} = a + u * (b - a)$
- $t \geq 0$
- $0 \leq u \leq 1$

t가 0보다 작으면 레이의 반대 방향이므로 무효, u가 [0, 1] 범위를 벗어나면 선분 위의 점이 아니므로 무효 처리했습니다. 이 식은 수업에서 다룬 교차 판정 원리와 동일하게 적용하였습니다. 엔진에서는 각 레이마다 이 함수를 호출하여 가장 가까운 교차점을 추출하는 기본 동작을 수행합니다.

3) FOV(시야각) 및 각도 계산 구조 구현

-코드 위치 : `compute_visibility_polygon()` 내부, `angle_diff()` 함수

플레이어가 바라보는 방향을 기준으로 특정 각도 범위만을 시야각으로 인정하기 위해, 기본적인 각도 계산 기능을 구현했습니다. 우선, `atan2(y, x)`를 활용하여 레이 가 향하는 절대 각도를 구했으며, 플레이어의 바라보는 각도(`facing_angle`)와 비교하기 위해 다음 함수를 사용했습니다.

```
def angle_diff(a, b):  
    # 두 각도 사이의 가장 짧은 방향(양수/음수)으로의 차이 [-pi, pi]  
    diff = a - b  
    while diff <= -math.pi:  
        diff += 2 * math.pi  
    while diff > math.pi:  
        diff -= math.pi * 2  
    return diff
```

```

# 정규화 + FOV 필터링
unique_angles = []
seen = set()
for a in angles:
    # Normalize to [-pi, pi]
    a = math.atan2(math.sin(a), math.cos(a))
    key = round(a, 4)
    if key in seen:
        continue
    seen.add(key)
    if not full_360:
        # 시야각(FOV)을 facing angle 주변으로 제한한다
        diff = angle_diff(a, facing_angle)
        if abs(diff) > fov_angle / 2.0:
            continue
    unique_angles.append(a)

```

angle_diff() 함수는 두 각도의 차이를 $[-\pi, \pi]$ 범위로 정규화하여, 시야 내의 각도인지 여부를 판단할 수 있게 해줍니다. 또한, 단일 레이가 아니라 여러 개의 후보 각도(벽의 끝점 방향)를 기반으로 레이를 생성하기 위해 각도를 정렬했고, 이를 통해 “왼쪽 가장자리 레이 → 중앙 레이 → 오른쪽 가장자리 레이” 순서로 자연스러운 다각형을 구성할 수 있는 기반을 마련했습니다. 이러한 각도 처리 방식은 수업의 Dot Product 기반 각도 개념을 실제 알고리즘에 적용한 기본 구현 단계입니다.

4) FOV(시야각) 및 각도 계산 구조 구현

-코드 위치 : 플레이어 이동: Game.update(), 벽 이동: Wall.update()

수업에서 다뤘던 선형 운동 방정식인 $\text{position} = \text{position} + \text{velocity} * \text{dt}$ 을 그대로 적용하여 플레이어와 이동하는 벽의 위치를 계산했습니다. 플레이어는 WASD 입력에 따라 속도 벡터를 갖고 이동하며, 벽의 경우 사용자가 지정한 속도 벡터를 기반으로 화면 내에서 왕복하도록 구성했습니다.

```

def update(self, dt): # Scene.update
    # 벽 업데이트 + 단순 바운스 처리
    for wall in self.walls:
        wall.update(dt)
        if wall.dynamic and wall.velocity.length() > 0:
            seg = wall.get_segment()
            for p in [seg.p1, seg.p2]:
                if p.x < 80 or p.x > 720:
                    wall.velocity.x *= -1
                if p.y < 80 or p.y > 520:
                    wall.velocity.y *= -1

```

```
def update(self, dt): # Wall.update
    if not self.dynamic:
        return

    # 회전
    if self.rotation_center is not None and self.angular_speed != 0.0:
        self.angle += self.angular_speed * dt
        # 회전 중심점을 기준으로 베이스 포인트를 회전시킨다
        self.segment.p1 = rotate_around(self.base_p1, self.rotation_center, self.angle)
        self.segment.p2 = rotate_around(self.base_p2, self.rotation_center, self.angle)
    # 평행 이동
    if self.velocity.x != 0 or self.velocity.y != 0:
        dp = self.velocity * dt
        self.segment.p1 = self.segment.p1 + dp
        self.segment.p2 = self.segment.p2 + dp
```

이 부분은 물리 엔진의 가장 기본적인 기능인 운동학(Kinematics)을 코드로 옮긴 구현으로서, “시간 변화에 따라 물체의 위치가 어떻게 달라지는가”를 처리하는 데 필수적인 요소입니다.

5) Pygame을 활용한 기본 시각화 구현

-코드 위치 : Game.draw()

엔진에서 계산된 가시 영역을 실제 화면에 렌더링하기 위하여, Pygame의 draw.polygon, draw.line, draw.circle 기능을 사용하여 다음 요소들을 시각화했습니다.

시각화 요소는 다음과 같습니다:

- 플레이어 위치(원)
- 벽(선분)
- 가시 다각형(반투명 폴리곤)
- 감지된 벽 하이라이트

이어서 FOV 모드일 때는 플레이어 위치를 다각형의 첫 꼭짓점으로 추가하여, 수업에서 다룬 "시야각" 개념을 시각적으로 표현할 수 있도록 구성했습니다.

```
def draw(self):
    self.screen.fill((20, 20, 30))

    # 가시 다각형 계산
    poly_points, hit_walls = self.visibility_engine.compute_visibility_polygon(
        self.player_pos, self.facing_angle, self.fov_angle, full_360=self.full_360
    )

    # 가시 다각형 그리기 (반투명)
    if len(poly_points) >= 2:
        poly_surface = pygame.Surface((self.WIDTH, self.HEIGHT), pygame.SRCALPHA)
        if self.full_360:
            # 360도 모드: 기존처럼 그대로 사용
            draw_points = [p.tuple() for p in poly_points]
        else:
            # FOV 모드: 플레이어를 꼭짓점으로 포함해서 부채꼴 모양으로 그림
            draw_points = [self.player_pos.tuple()] + [p.tuple() for p in poly_points]

        pygame.draw.polygon(
            poly_surface,
            (255, 255, 100, 60),
            draw_points
        )
        self.screen.blit(poly_surface, (0, 0))

    # 벽 그리기 (가시 다각형에 맞은 벽은 빨간색으로 하이라이트)
    for wall in self.scene.get_walls():
        seg = wall.get_segment()
        color = (120, 120, 120)
        if wall in hit_walls:
            color = (255, 80, 80) #하이라이트
        pygame.draw.line(
            self.screen,
            color,
            seg.p1.tuple(),
            seg.p2.tuple(),
            3
        )

    # 플레이어
    pygame.draw.circle(self.screen, (80, 200, 255), self.player_pos.tuple(), 8)

    # 시야 방향 표시
    dir_vec = Vec2.from_angle(self.facing_angle) * 40
    pygame.draw.line(
        self.screen,
        (80, 200, 255),
        self.player_pos.tuple(),
        (self.player_pos + dir_vec).tuple(),
        2
    )
```

4. 엔진 기술 개발 심화 구현

이번 프로젝트의 핵심은 수업에서 배운 기초적인 충돌 판정 및 벡터 연산 기능을 넘어, 실제 게임 엔진 수준의 ****가시 영역(visibility polygon)****을 생성하는 고급 기능들을 직접 구현하는 것이었습니다. 이를 위해 단순한 충돌 계산 이상의 알고리즘 설계, 구조화된 엔진 구성, 시야각(FOV) 처리 알고리즘, 동적 환경 지원 등 다양한 심화 요소를 발전시켜 엔진에 반영했습니다. 아래는 이번 프로젝트에서 수행한 심화 구현 요소입니다.

1) Visibility Polygon 알고리즘 구현 (가시 다각형 생성)

-코드 위치 : `compute_visibility_polygon()` 전체

수업에서는 한 번의 충돌 판정 또는 단일 시야각 계산을 다루었지만, 이번 프로젝트에서는 한 단계 더 나아가 플레이어 기준으로 실제 “보이는 영역 전체”를 다각형 형태로 계산하는 알고리즘을 직접 개발했습니다.

심화 알고리즘은 다음 단계로 구성했습니다:

1. 국소적 후보각도 생성
 - 모든 벽의 꼭짓점 p 에 대해 각도 $\text{angle} = \text{atan2}((p - \text{origin}).y, (p - \text{origin}).x)$ 를 계산했습니다.
 - 벽 모서리에서 발생하는 경계 오차를 해결하기 위해 $\text{angle} \pm \varepsilon$ 값을 추가하여 더 정밀한 레이 샘플링을 수행했습니다.
2. 각 레이에 대해 Ray-Segment 충돌 검사 수행
 - 각도마다 레이를 발사하고 가장 가까운 충돌점을 찾았습니다.
 - 결과적으로 “보이는 점”들의 집합을 얻게 되었습니다.
3. 충돌점들을 각도 순으로 정렬하여 다각형 생성
 - 정렬된 교차점들을 잇는 것이 바로 가시 다각형이며, 화면에서 왼쪽 끝 레이 → 오른쪽 끝 레이까지 자연스럽게 연결하도록 구성했습니다.

이 방식은 수업에서 배우지 않았던 내용으로, Shadow Casting 알고리즘을 기반 하였습니다. 실제 게임 엔진에서 사용하는 시야조명 시스템의 핵심 원리를 참고하였습니다.

2) 시야각(FOV) 모드와 360도 모드 지원

-코드 위치 : FOV : `compute_visibility_polygon()`, 모드 전환 : `handle_events()`

FOV 모드 (정해진 시야각 기반 시야 계산)을 구현하였습니다. 플레이어가 바라보는 방향(`facing_angle`)을 중심으로 설정된 시야각(`fov_angle`) 내에서만 가시점들을 계산했습니다. 이 기능을 통해 FPS 게임이나 스텔스 게임에서 자주 사용되는 “앞만 보이는” 시야를 구현했습니다.

360도 모드 (전 방향 가시)도 구현하였습니다. 전 방향으로 균등하게 레이를 발사하여 플레이어를 중심으로 한 완전한 가시 영역을 계산했습니다. 이는 실제 빛이나 센서 기반의 시야 모델을 단순화하여 구현한 형태입니다.

가장 복잡한 문제는 플레이어가 왼쪽(π 라디안 근처)을 바라볼 때 각도 정렬이 무너지는 현상이었습니다. 예를 들어, 시야각이 90도라면, FOV는 $135^\circ \sim 225^\circ$ 와 같이 나올 수 있습니다. 하지만 각도를 $[-\pi, \pi]$ 로 정규화하면 225° 가 -135° 로 변해, 정렬 시 갑자기 순서가 뒤섞이는 문제가 발생했습니다.

이를 해결하기 위해 다음 알고리즘을 설계했습니다:

- `diff = angle_diff(a, facing_angle)`
- `a`는 레이의 각도
- `facing_angle`은 플레이어의 방향
- `angle_diff()`는 두 각도를 기준 방향으로부터의 상대각으로 변환

이 값으로 정렬하면 항상 왼쪽 끝 각도 \rightarrow 가운데 \rightarrow 오른쪽 끝 각도 순서가 유지되며, 부채꼴 형태의 FOV가 정확하게 생성됩니다.

3) 동적 환경(Dynamic Environment) 처리

-코드 위치 : 벽 업데이트: `Wall.update()`, 장면 업데이트: `Scene.update()`

가시성 엔진은 정적인 벽뿐만 아니라, 실시간으로 움직이거나 회전하는 벽까지 지원하도록 설계했습니다.

먼저 이동하는 벽에 대하여, 선형 운동 방정식 $position = position + velocity * dt$ 을 모든 벽에 적용하여 화면에서 실제로 이동하도록 구현했습니다. 회전하는 벽도 회전 중심(`rotation_center`)을 기준으로 각속도(`angular_speed`)를 누적하고, 매 프레임 회전 변환을 적용하여 벽이 계속 회전하도록 구성했습니다. 매 프레임 벽의 위치와 방향이 변하기 때문에 레이는 항상 “현재 상태의 벽”과 충돌 검사를 수행해야 했습니다. 이 구조를 통해 플레이어는 동적으로 변화하는 환경 속에서도 정확한 가시 정보를 얻을 수 있습니다.

4) 하이라이트 기능(Detected Wall Visualization) 구현

-코드 위치 : `draw()` 함수 내부 `hit_walls` 사용 부분

엔진 내부에서는 각 레이가 충돌한 벽 정보를 기록하여 `hit_walls` 집합에 저장했습니다. 이를 시각화 단계에서 활용하여, 가시 영역에 실제로 감지된 벽은 빨간색으로 표시했습니다. 이 기능은 단순한 시각적 효과를 넘어, “엔진이 어떤 벽과 상호작용하고 있으며 충돌 계산이 어떻게 이루어졌는지”를 직관적으로 보여주기 위한 디버깅 및

해석용 기능입니다. 이를 통해 엔진 사용자(플레이어)는 보이지 않는 내부 계산을 시각적으로 확인할 수 있으며, 계산 과정이 정상적으로 이루어졌는지 검증할 수 있습니다.

5) 엔진 구조(Engine Architecture) 설계

-코드 구성:

- /math 해당: Vec2, 회전 함수
- /collision 해당: ray_segment_intersection()
- /visibility 해당: VisibilityEngine
- /scene 해당: Wall, Scene
- /game 해당: Game

이번 프로젝트에서는 기능을 단일 클래스에 혼합하지 않고, 엔진 구조를 모듈 단위로 분리하여 설계했습니다.