Imports

```
pip install vectormath
```

```
Collecting vectormath
  Downloading vectormath-0.2.2.tar.gz (9.2 kB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: numpy>=1.7 in /usr/local/lib/python3.10/dist-packages (from vectormath) (1.26.4)
Building wheels for collected packages: vectormath
  Building wheel for vectormath (setup.py) ... done
  Created wheel for vectormath: filename=vectormath-0.2.2-py3-none-any.whl size=7883 sha256=8f1cb3e5e3e5077fb909cfe50933fea2
  Stored in directory: /root/.cache/pip/wheels/27/1f/a6/42b53202f630cfb11e87c5a04f3783944722b0053f85753757
Successfully built vectormath
Installing collected packages: vectormath
Successfully installed vectormath-0.2.2
```

```
import math
import numpy as np
import vectormath as vm
from vectormath import Vector2, Vector3
import matplotlib.pyplot as plt
import random
```

SENSOR POSITION AND ANGLE

[ ] ↳ 1 cell hidden

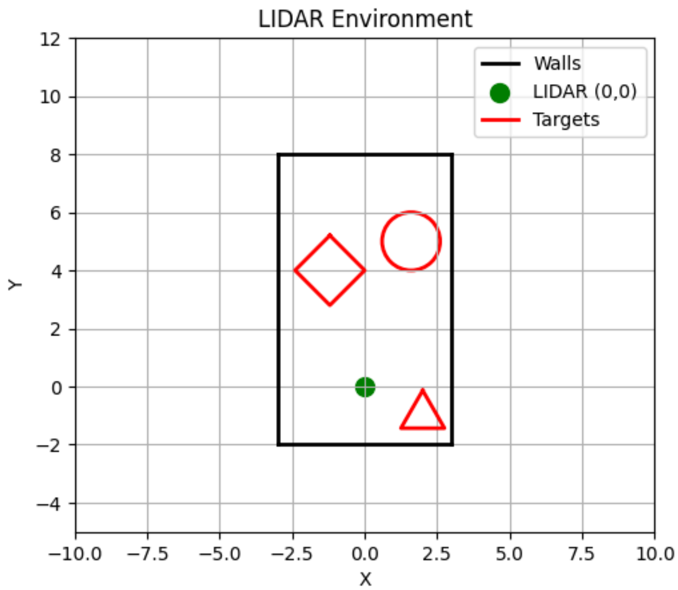
DEFINE ENVIRONMENT

```
# Define all the walls as line vectors
walls = {
    "left_wall": {"start": Vector2(-3, -2), "end": Vector2(-3, 8), "rho": 0.1},
    "right_wall": {"start": Vector2(3, -2), "end": Vector2(3, 8), "rho": 0.1},
    "bottom_wall": {"start": Vector2(-3, -2), "end": Vector2(3, -2), "rho": 0.1},
    "top_wall": {"start": Vector2(-3, 8), "end": Vector2(3, 8), "rho": 0.1}
}

# then define all the obstacles as a list of line vectors

targets = {
    "diamond": {
        "center": Vector2(-1.2, 4),
        "size": 2.4,                # length of each side
        "rho": 0.5
    },
    "circle": {
        "center": Vector2(1.6, 5),
        "radius": 1,
        "rho": 0.9
    },
    "triangle": {
        "center": Vector2(2, -1),
        "size": 1.5,  # side length of the equilateral triangle
        "rho": 0.6
    }
}


# also attach the value of the reflectivity of the surface to each of the vectors

# add all the wall and obstacles to an ENVIR variable
```

DRAW THE ENVIRONMENT

```
#@title DRAW THE ENVIRONMENT
# Function to draw the environment
def plot_environment(walls, targets):
    fig, ax = plt.subplots()
```

```
    fig, ax = plt.subplots()

    # Plot walls (black lines)
    for wall_name, wall in walls.items():
        start = wall["start"]
        end = wall["end"]
        if wall_name == "left_wall":
          ax.plot([start.x, end.x], [start.y, end.y], 'k-', linewidth=2, label="Walls")
        else:
          ax.plot([start.x, end.x], [start.y, end.y], 'k-', linewidth=2)

    # Plot LIDAR origin (green point)
    ax.scatter(0, 0, color='green', s=100, label="LIDAR (0,0)")

    # Plot the targets
    # Diamond
    diamond = targets["diamond"]
    diamond_size = diamond["size"]
    diamond_center = diamond["center"]
    diamond_points = [
        Vector2(diamond_center.x, diamond_center.y + diamond_size / 2),
        Vector2(diamond_center.x + diamond_size / 2, diamond_center.y),
        Vector2(diamond_center.x, diamond_center.y - diamond_size / 2),
        Vector2(diamond_center.x - diamond_size / 2, diamond_center.y)
    ]
    diamond_x = [point.x for point in diamond_points] + [diamond_points[0].x]
    diamond_y = [point.y for point in diamond_points] + [diamond_points[0].y]
    ax.plot(diamond_x, diamond_y, 'r-', linewidth=2)

    # Circle
    circle = targets["circle"]
    circle_center = circle["center"]
    circle_radius = circle["radius"]
    circle_plot = plt.Circle((circle_center.x, circle_center.y), circle_radius, color='red', fill=False, linewidth=2)
    ax.add_artist(circle_plot)

    # Triangle (equilateral triangle)
    triangle = targets["triangle"]
    triangle_size = triangle["size"]
    triangle_center = triangle["center"]
    triangle_height = (triangle_size * (3**0.5)) / 2
    triangle_points = [
        Vector2(triangle_center.x, triangle_center.y + 2/3 * triangle_height),  # Top point
        Vector2(triangle_center.x - triangle_size / 2, triangle_center.y - 1/3 * triangle_height),
        Vector2(triangle_center.x + triangle_size / 2, triangle_center.y - 1/3 * triangle_height)
    ]
    triangle_x = [point.x for point in triangle_points] + [triangle_points[0].x]
    triangle_y = [point.y for point in triangle_points] + [triangle_points[0].y]


    ax.plot(triangle_x, triangle_y, 'r-', label="Targets", linewidth=2)

    # Labels, limits, and grid
    ax.set_xlim(-10, 10)
    ax.set_ylim(-5, 12)
    ax.set_aspect('equal', 'box')
    ax.grid(True)
    ax.legend()

    plt.title("LIDAR Environment")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.show()

# Run the function to plot the environment
plot_environment(walls, targets)
```



SIMPLE RAY TRACING

find_incidenceAngle(line_start, line_end, target_start, target_end, intersection)

```
#@title find_incidenceAngle(line_start, line_end, target_start, target_end, intersection)

# function to normalize vectors
def normalize_vector(vector):
    """
    Manually normalize a 2D vector.

    Args:
    - vector: Vector2 object representing the vector.

    Returns:
    - normalized_vector: A normalized Vector2 (magnitude of 1).
    """
    magnitude = np.sqrt(vector.x**2 + vector.y**2)

    if magnitude != 0:
        return Vector2(vector.x / magnitude, vector.y / magnitude)
    else:
        return Vector2(0, 0)  # Handle zero-length vectors


# seperate function to detect the surface normal of LINES
def find_surface_normal(target_start, target_end):
    # Vector along the surface
    surface_vector = target_end - target_start

    # Normal vector by rotating the surface vector by 90 degrees
    surface_normal = Vector2(-surface_vector.y, surface_vector.x)

    # Normalize the surface normal

    surface_normal_normalized = normalize_vector(surface_normal)

    # if magnitude != 0:
    #     surface_normal_normalized = Vector2(surface_normal.x / magnitude, surface_normal.y / magnitude)
    # else:
    #     surface_normal_normalized = Vector2(0, 0)  # Handle the case of zero-length vectors

    return surface_normal_normalized

# TEST CASE
print(find_surface_normal(Vector2(0,0), Vector2(1,0)))

def find_circle_surface_normal(center, surface_point):
```

```
    """
    Calculate the surface normal at a given point on the surface of a circle.

    Args:
    - center: Vector2 or Vector3, the center of the circle.
    - surface_point: Vector2 or Vector3, the point on the surface of the circle .

    Returns:
    - surface_normal: The normalized surface normal vector at the given surface point.
    """
    # Vector from the center of the circle the surface point
    normal_vector = surface_point - center

    surface_normal = normalize_vector(normal_vector)
    # # Normalize the vector to get the unit surface normal
    # magnitude = np.sqrt(normal_vector.x**2 + normal_vector.y**2)  # For 2D (use z for 3D)

    # if magnitude != 0:
    #     surface_normal = Vector2(normal_vector.x / magnitude, normal_vector.y / magnitude)
    # else:
    #     surface_normal = Vector2(0, 0)  # Handle zero-length vectors

    return surface_normal


# function to find Incidence angle
def find_angle_of_incidence(ray_start, ray_end, surface_normal):
    """
    Calculate the angle of incidence between the ray and the surface normal.

    Args:
    - ray_start: Vector2, the starting point of the ray (LIDAR sensor).
    - ray_end: Vector2, the point where the ray intersects the surface.
    - surface_normal: Vector2, the normal vector of the surface at the intersection point.

    Returns:
    - angle_of_incidence: The angle of incidence in radians.
    """
    # Calculate the direction vector of the ray (ray_end - ray_start)
    ray_direction = ray_end - ray_start

    # Normalize the ray direction and surface normal vectors
    ray_direction_normalized = normalize_vector(ray_direction)
    surface_normal_normalized = normalize_vector(surface_normal)

    # Calculate the dot product between the ray direction and the surface normal
    dot_product = ray_direction_normalized.dot(surface_normal_normalized)

    # Calculate the angle of incidence using the arccos of the dot product
    # Ensure the dot product is clamped within the range [-1, 1] to avoid numerical errors
    dot_product = np.clip(dot_product, -1.0, 1.0)
    angle_of_incidence = np.arccos(dot_product)

    # normalize of opposite vectors
    if angle_of_incidence > np.pi / 2:
        angle_of_incidence = np.pi - angle_of_incidence

    # convert to degrees
    angle_of_incidence_degrees = np.degrees(angle_of_incidence)

    return angle_of_incidence_degrees


# TEST CASE
print(find_angle_of_incidence(Vector2(0,0), Vector2(0,1), find_surface_normal(Vector2(-1,2), Vector2(2,3))))
```

```
[-0.  1.]
18.434948822922
```

find_intersections(start,end,walls,targets)

```
#@title find_intersections(start,end,walls,targets)
# function that finds the intersection points of the RAY and ENVIR

def find_intersections(line_start, line_end, walls, targets):
    intersections = []  # Store all intersection points in a single list
```

```
# Helper function to find intersection between two line segments (wall or diamond edges)
def line_intersection(p1, p2, p3, p4, rho):
    """ Return the intersection point of two lines defined by points p1->p2 and p3->p4 """
    denom = (p1.x - p2.x) * (p3.y - p4.y) - (p1.y - p2.y) * (p3.x - p4.x)
    if denom == 0:
        return None  # Parallel lines
    x = ((p1.x * p2.y - p1.y * p2.x) * (p3.x - p4.x) - (p1.x - p2.x) * (p3.x * p4.y - p3.y * p4.x)) / denom
    y = ((p1.x * p2.y - p1.y * p2.x) * (p3.y - p4.y) - (p1.y - p2.y) * (p3.x * p4.y - p3.y * p4.x)) / denom
    intersection_point = Vector2(x, y)

    # Check if the intersection point lies on both line segments
    if (min(p1.x, p2.x) <= x <= max(p1.x, p2.x) and
        min(p1.y, p2.y) <= y <= max(p1.y, p2.y) and
        min(p3.x, p4.x) <= x <= max(p3.x, p4.x) and
        min(p3.y, p4.y) <= y <= max(p3.y, p4.y)):
        normal = find_surface_normal(p3, p4)
        incidence_angle = find_angle_of_incidence(line_start, intersection_point, normal)
        return [intersection_point, rho, incidence_angle]
    return None

# Intersection with walls (line segments)
for wall_name, wall in walls.items():
    intersection_point = line_intersection(line_start, line_end, wall["start"], wall["end"], wall["rho"])
    if intersection_point is not None:
        intersections.append(intersection_point)

# Intersection with the diamond (diamond is made of 4 line segments)
diamond = targets["diamond"]
diamond_size = diamond["size"]
diamond_center = diamond["center"]
diamond_points = [
    Vector2(diamond_center.x, diamond_center.y + diamond_size / 2),
    Vector2(diamond_center.x + diamond_size / 2, diamond_center.y),
    Vector2(diamond_center.x, diamond_center.y - diamond_size / 2),
    Vector2(diamond_center.x - diamond_size / 2, diamond_center.y)
]
for i in range(len(diamond_points)):
    p1 = diamond_points[i]
    p2 = diamond_points[(i + 1) % len(diamond_points)]
    intersection_point = line_intersection(line_start, line_end, p1, p2, diamond["rho"])
    if intersection_point is not None:
        intersections.append(intersection_point)




# Intersection with the circle (target)
circle = targets["circle"]
circle_center = circle["center"]
circle_radius = circle["radius"]

# To find intersections with a circle, solve the quadratic equation for the intersection of a line and a circle
def circle_line_intersection(line_start, line_end, circle_center, radius, rho):
    """Find the intersection points between a line and a circle."""
    # Parametric form of the line: P = line_start + t * (line_end - line_start)
    # Circle equation: (x - cx)^2 + (y - cy)^2 = r^2
    dx = line_end.x - line_start.x
    dy = line_end.y - line_start.y
    fx = line_start.x - circle_center.x
    fy = line_start.y - circle_center.y

    a = dx**2 + dy**2
    b = 2 * (fx * dx + fy * dy)
    c = fx**2 + fy**2 - radius**2

    discriminant = b**2 - 4 * a * c
    if discriminant < 0:
        return []  # No intersection
    discriminant = np.sqrt(discriminant)

    # Two possible intersection points
    t1 = (-b - discriminant) / (2 * a)
    t2 = (-b + discriminant) / (2 * a)

    intersection_points = []
    if 0 <= t1 <= 1:
```

```
            intersection = Vector2(line_start.x + t1 * dx, line_start.y + t1 * dy)
            normal = find_circle_surface_normal(circle_center, intersection)
            incidence_angle = find_angle_of_incidence(line_start, intersection, normal)
            intersection_points.append([intersection, rho, incidence_angle])
        if 0 <= t2 <= 1:
            intersection = Vector2(line_start.x + t2 * dx, line_start.y + t2 * dy)
            normal = find_circle_surface_normal(circle_center, intersection)
            incidence_angle = find_angle_of_incidence(line_start, intersection, normal)
            intersection_points.append([intersection, rho, incidence_angle])

        return intersection_points

    intersections.extend(circle_line_intersection(line_start, line_end, circle_center, circle_radius, circle["rho"]))

    # Intersection with the triangle (3 line segments)
    triangle = targets["triangle"]
    triangle_size = triangle["size"]
    triangle_center = triangle["center"]
    triangle_height = (triangle_size * (3**0.5)) / 2
    triangle_points = [
        Vector2(triangle_center.x, triangle_center.y + 2/3 * triangle_height),  # Top point
        Vector2(triangle_center.x - triangle_size / 2, triangle_center.y - 1/3 * triangle_height),
        Vector2(triangle_center.x + triangle_size / 2, triangle_center.y - 1/3 * triangle_height)
    ]
    for i in range(len(triangle_points)):
        p1 = triangle_points[i]
        p2 = triangle_points[(i + 1) % len(triangle_points)]
        intersection_point = line_intersection(line_start, line_end, p1, p2, triangle["rho"])
        if intersection_point is not None:
            intersections.append(intersection_point)

    return intersections




# TESTING THE FUNCTION

# Get the intersections
intersect_rho = find_intersections(sensor_position, RayGenerator(sensor_position,63), walls, targets)

# Print the results
print("Intersection points:")
for point in intersect_rho:
    print(f"  {point}")
```

```
Intersection points:
  [Vector2([3.        , 5.88783152]), 0.1, 63.0]
  [Vector2([2.10906412, 4.1392714 ]), 0.9, 57.60151142815228]
  [Vector2([2.59556445, 5.09408205]), 0.9, 57.601511428152264]
```

closest_surface(line_start , intersections)

```
#@title closest_surface(line_start , intersections)
# function that compares the various intersections and then finds the closest point to the sensor

def closest_surface(line_start, intersections):

    #seperate rho values from points
    point_values  = [point_value for point_value,_,_ in intersections]

    if not intersections:
        return None  # If there are no intersections, return None

    # Calculate the distances between the start point and each intersection point
    distances = [np.linalg.norm(np.array([sensor_position.x - point_val.x, sensor_position.y - point_val.y])) for point_val in po

    # Find the index of the minimum distance
    min_index = np.argmin(distances)

    # Return the closest intersection point
    return intersections[min_index]
```

```
#TESTING THE FUNCTION
print(closest_surface(sensor_position, intersect_rho))
```

```
[Vector2([2.10906412, 4.1392714 ]), 0.9, 57.60151142815228]
```

GENERATING THE POINT CLOUD

[ ] ↳ 11 cells hidden

NEW ENVIRONMENT DEFINE AND UPDATE

from vectormath import Vector2 import matplotlib.pyplot as plt import matplotlib.patches as patches

Define the environment

environment = { "walls": {

```
    # Room 1
    "room1_left": {"start": Vector2(-5, -5), "end": Vector2(-5, -1), "rho": 0.1},  # Break for window
    "room1_right": {"start": Vector2(0, -5), "end": Vector2(0, -1), "rho": 0.1},  # Break for the door
    "room1_top": {"start": Vector2(-5, 5), "end": Vector2(0, 5), "rho": 0.1},
    "room1_bottom": {"start": Vector2(-5, -5), "end": Vector2(0, -5), "rho": 0.1},

    # Room 2
    "room2_left": {"start": Vector2(0, 1), "end": Vector2(0, 5), "rho": 0.1},  # Break for the door
    "room2_right": {"start": Vector2(5, -5), "end": Vector2(5, 1), "rho": 0.1},  # Break for window
    "room2_top": {"start": Vector2(0, 5), "end": Vector2(5, 5), "rho": 0.1},
    "room2_bottom": {"start": Vector2(0, -5), "end": Vector2(5, -5), "rho": 0.1},
},
"fire": {
    "location": {"center": Vector2(4, 2), "radius": 1.0, "rho": 1.0},  # Fire near the bed
},
"static_targets": [
    {"name": "bed", "bottom_left": Vector2(3, 1), "top_right": Vector2(5, 3), "rho": 0.3},
    {"name": "table", "bottom_left": Vector2(2, -3), "top_right": Vector2(3, -2), "rho": 0.5},
],
"dynamic_target": {
    "name": "human",
    "start": Vector2(2, 2),  # Starting position
    "control": Vector2(0, 0),  # Control point near the door
    "end": Vector2(-3, -3),  # Final position
}
```

}

Bézier curve function

def bezier_point(start, control, end, u): return (1 - u)**2 * start + 2 * (1 - u) * u * control + u**2 * end

Function to get the environment at any time

def get_environment_at_time(env, time): """ Updates the environment to include the current position of the dynamic target (human) based on the provided time and returns the updated environment.

```
Args:
    env (dict): Original environment with walls, fire, static targets, and dynamic target.
    time (float): Current simulation time.

Returns:
    dict: Updated environment with the current dynamic target position.
```

```
    """
    human = env["dynamic_target"]
    start, control, end = human["start"], human["control"], human["end"]

    # Determine the current position of the human
    if time < 5:
        current_pos = start
    elif time <= 7:
        u = (time - 5) / (7 - 5)  # Normalize time for Bézier calculation
        current_pos = bezier_point(start, control, end, u)
    else:
        current_pos = end

    # Update the environment structure
    updated_env = {
        "walls": env["walls"],
        "fire": {
            "location": {
                "center": env["fire"]["location"]["center"],
                "radius": env["fire"]["location"]["radius"],
                "rho": env["fire"]["location"]["rho"],
            }
        },
        "static_targets": [
            {
                "name": target["name"],
                "bottom_left": target["bottom_left"],
                "top_right": target["top_right"],
                "rho": target["rho"],
            }
            for target in env["static_targets"]
        ],
        "dynamic_target": {
            "name": human["name"],
            "position": current_pos,
            "radius": 0.5,  # Assuming a fixed radius for the human
            "rho": 0.8  # Reflectivity of the human
        },
        "time": time  # Include the current simulation time
    }

    return updated_env
```

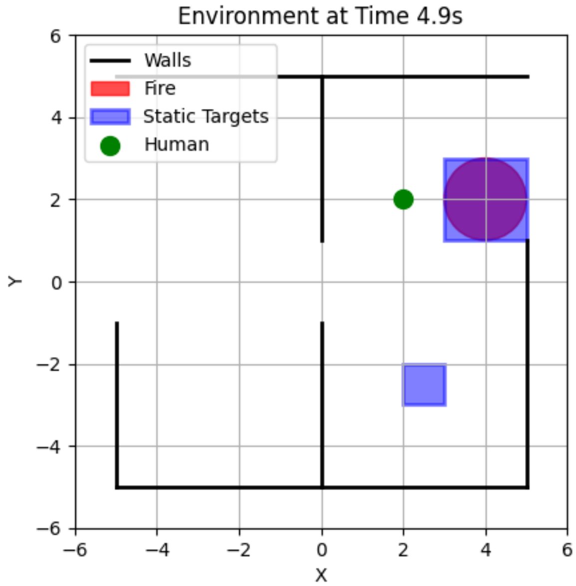
def bezier_point(p0, p1, p2, t): """ Calculates a point on a quadratic Bézier curve.

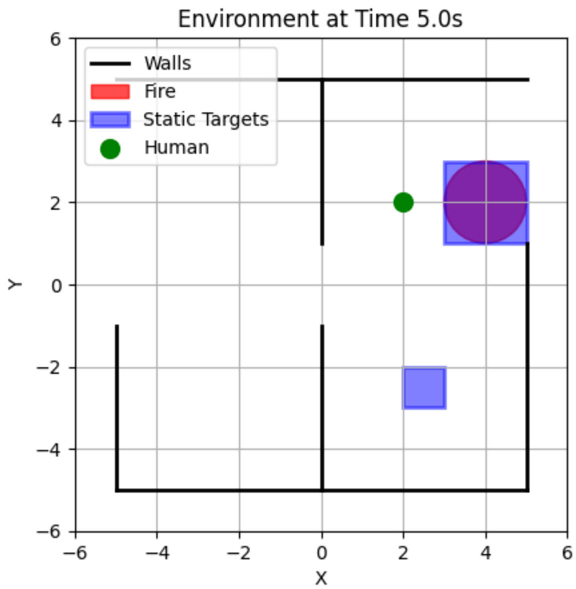
```
    Args:
        p0 (Vector2): Start point of the curve.
        p1 (Vector2): Control point of the curve.
        p2 (Vector2): End point of the curve.
        t (float): Parameter along the curve (0 <= t <= 1).

    Returns:
        Vector2: Point on the curve at parameter t.
    """
    return (1 - t) ** 2 * p0 + 2 * (1 - t) * t * p1 + t ** 2 * p2
```

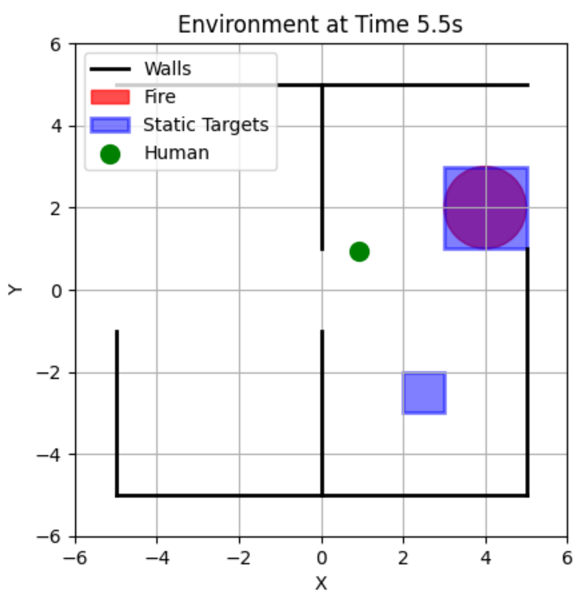
Draw Environment at time t

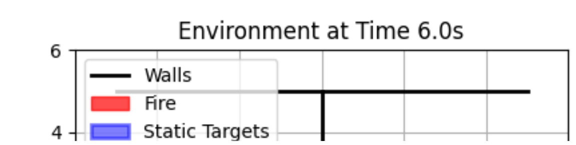
```
# Function to draw the environment at a given time
def draw_environment_at_time(env, time):
    # Get the static environment at the given time
    static_env = get_environment_at_time(env, time)
```

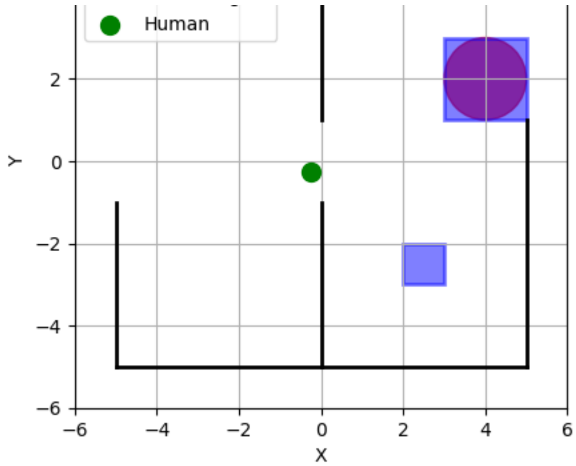
```
    fig, ax = plt.subplots()

    # Plot walls
    for wall_name, wall in static_env["walls"].items():
        start, end = wall["start"], wall["end"]
        ax.plot([start.x, end.x], [start.y, end.y], 'k-', linewidth=2, label="Walls" if wall_name == "room1_left" else "")

    # Plot fire
    fire = static_env["fire"]
    fire_center, fire_radius = fire["location"]["center"], fire["location"]["radius"]
    fire_circle = plt.Circle((fire_center.x, fire_center.y), fire_radius, color='red', alpha=0.7, label="Fire")
    ax.add_artist(fire_circle)

    # Plot static targets
    for target in static_env["static_targets"]:
        bl, tr = target["bottom_left"], target["top_right"]
        rect = patches.Rectangle((bl.x, bl.y), tr.x - bl.x, tr.y - bl.y, linewidth=2, edgecolor='blue', facecolor='blue', alpha=
        ax.add_patch(rect)

    # Plot dynamic target (human)
    human = static_env["dynamic_target"]
    ax.scatter(human["position"].x, human["position"].y, color='green', s=100, label="Human", zorder=4)

    # Add labels, grid, and legends
    ax.set_xlim(-6, 6)
    ax.set_ylim(-6, 6)
    ax.set_aspect('equal', 'box')
    ax.grid(True)
    ax.legend()
    plt.title(f"Environment at Time {time}s")
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.show()

    return static_env

# Example usage
for t in [4.9, 5.0, 5.5, 6.0, 6.5, 7.0]:
    draw_environment_at_time(environment, time=t)
```

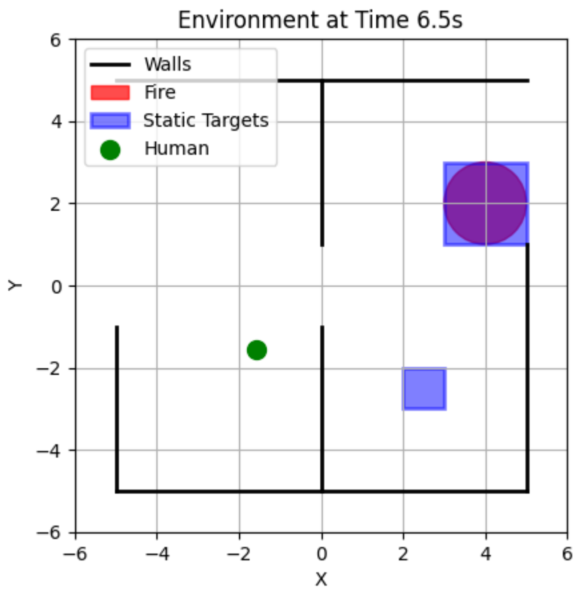

Environment at Time 4.9s
Walls
Fire
Static Targets
Human
X
Y

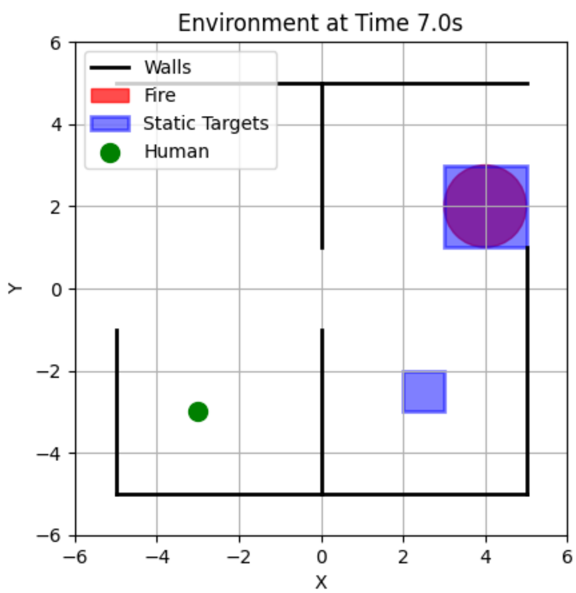

Environment at Time 5.0s
Walls
Fire
Static Targets
Human
X
Y


Environment at Time 5.5s
Walls
Fire
Static Targets
Human
X
Y


Environment at Time 6.0s
Walls
Fire
Static Targets


Human
Y
X


Environment at Time 6.5s
Walls
Fire
Static Targets
Human
Y
X


Environment at Time 7.0s
Walls
Fire
Static Targets
Human
Y
X

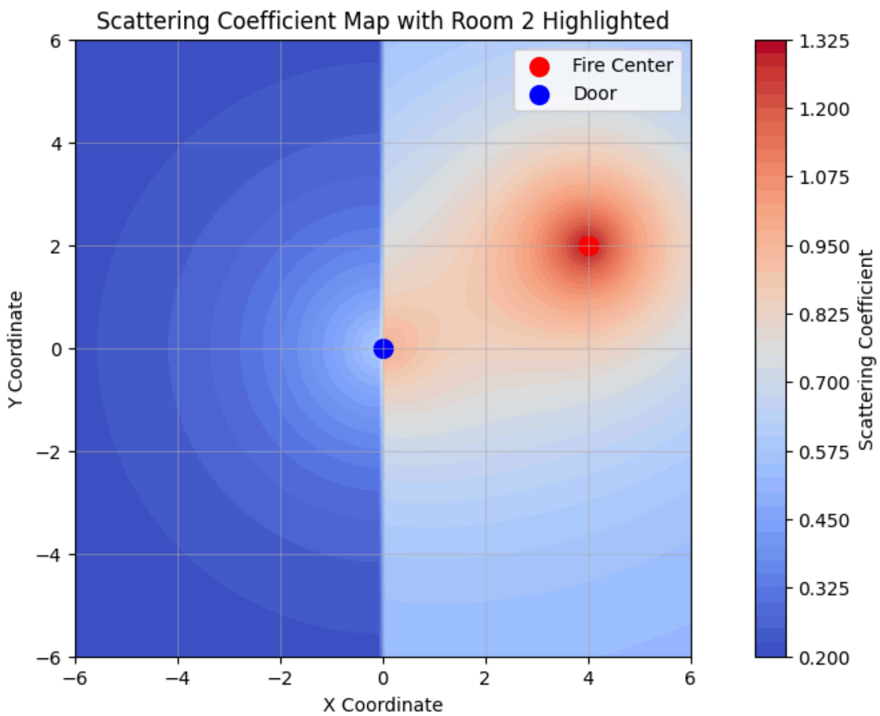
NEW Intensity Loss Function

```
import numpy as np

# Define the scattering coefficient map function
def scattering_coefficient_map(point, fire_center, max_scattering=0.2, base_scattering=0.02):
    """
    Calculates the scattering coefficient at a given point based on proximity to the fire
    and door location.

    Args:
        point (Vector2): The point where the ray intersects.
        fire_center (Vector2): Center of the fire.
        max_scattering (float): Maximum scattering coefficient near the fire.
        base_scattering (float): Base scattering coefficient far from the fire.

    Returns:
        float: Scattering coefficient at the given point.
    """
    # Compute distance from the fire
    distance_to_fire = np.linalg.norm([point.x - fire_center.x, point.y - fire_center.y])

    # Radial falloff for scattering coefficient
    radial_scattering = max_scattering * np.exp(-distance_to_fire)

    # Additional scattering near the door (if the point is close to the door)
    door_center = Vector2(0, 0)  # Center of the door between the two rooms
    distance_to_door = np.linalg.norm([point.x - door_center.x, point.y - door_center.y])
    door_scattering = 0.05 * np.exp(-distance_to_door)  # Smaller contribution near the door

    return max(base_scattering, radial_scattering + door_scattering)

def scattering_coefficient_map_with_rooms(point, fire_center, environment, max_scattering=0.8, base_scattering=0.5):
    """
    Calculates the scattering coefficient at a given point based on proximity to the fire,
    room layout, and door position.

    Args:
        point (Vector2): The point where the ray intersects.
        fire_center (Vector2): Center of the fire.
        environment (dict): Environment data with walls and rooms.
        max_scattering (float): Maximum scattering coefficient near the fire.
        base_scattering (float): Base scattering coefficient for Room 2.

    Returns:
        float: Scattering coefficient at the given point.
    """
    # Determine if the point is in Room 2
    in_room_2 = point.x > 0  # Room 2 starts from x > 0 based on environment definition

    # Higher base scattering for Room 2
    scattering = base_scattering if in_room_2 else 0.2

    # Add radial falloff for the fire in Room 2
    if in_room_2:
        distance_to_fire = np.linalg.norm([point.x - fire_center.x, point.y - fire_center.y])
        scattering += max_scattering * np.exp(-0.5 * distance_to_fire)  # Stronger falloff

    # Add bleeding effect through the door
    door_center = Vector2(0, 0)  # Center of the door between the two rooms
    distance_to_door = np.linalg.norm([point.x - door_center.x, point.y - door_center.y])
    door_bleed = 0.4 * np.exp(-0.5 * distance_to_door)  # Smooth transition through the door
    scattering += door_bleed

    return scattering

# Intensity loss function
def intensity_loss_with_scattering_and_rooms(initial_intensity, point, distance, fire_center, environment):
    """
    Calculates the intensity loss based on the updated scattering coefficient map.

    Args:
        initial_intensity (float): The emitted intensity of the beam.
        point (Vector2): The intersection point of the ray.
```

```
        distance (float): Distance traveled by the ray.
        fire_center (Vector2): Center of the fire.
        environment (dict): Environment data with walls and rooms.

    Returns:
        float: Reduced intensity at the given point.
    """
    # Get the scattering coefficient at the point
    scattering_coeff = scattering_coefficient_map_with_rooms(point, fire_center, environment)

    # Apply Beer-Lambert law with the scattering coefficient
    attenuated_intensity = initial_intensity * np.exp(-scattering_coeff * distance)

    return attenuated_intensity

# Example usage
fire_center = Vector2(4, 2)  # Fire location near the bed
point = Vector2(3, 1)  # Intersection point near the fire
initial_intensity = 1.0
distance = 2.0  # Distance of the intersection point from the sensor

reduced_intensity = intensity_loss_with_scattering_and_rooms(
    initial_intensity, point, distance, fire_center, environment
)
print(f"Reduced intensity at {point}: {reduced_intensity:.3f}")
```

```
Reduced intensity at [3. 1.]: 0.142
```

```
# Updated visualization function
def visualize_scattering_map_with_rooms(fire_center, environment, x_range=(-6, 6), y_range=(-6, 6), resolution=100):
    """
    Visualizes the scattering coefficient map based on the environment and fire position.

    Args:
        fire_center (Vector2): Center of the fire.
        environment (dict): Environment data with walls and rooms.
        x_range (tuple): Range of x-coordinates.
        y_range (tuple): Range of y-coordinates.
        resolution (int): Number of points in each dimension.
    """
    x = np.linspace(x_range[0], x_range[1], resolution)
    y = np.linspace(y_range[0], y_range[1], resolution)
    X, Y = np.meshgrid(x, y)

    # Calculate scattering coefficients for each point
    Z = np.zeros_like(X)
    for i in range(resolution):
        for j in range(resolution):
            point = Vector2(X[i, j], Y[i, j])
            Z[i, j] = scattering_coefficient_map_with_rooms(point, fire_center, environment)

    # Plot the heatmap
    plt.figure(figsize=(8, 6))
    plt.contourf(X, Y, Z, levels=50, cmap="coolwarm")
    plt.colorbar(label="Scattering Coefficient")
    plt.scatter(fire_center.x, fire_center.y, color='red', label="Fire Center", s=100)
    plt.scatter(0, 0, color='blue', label="Door", s=100)  # Door center
    plt.title("Scattering Coefficient Map with Room 2 Highlighted")
    plt.xlabel("X Coordinate")
    plt.ylabel("Y Coordinate")
    plt.legend()
    plt.grid(alpha=0.5)
    plt.axis('scaled')
    plt.show()

# Visualize the updated map
visualize_scattering_map_with_rooms(fire_center, environment)
```




```
# Updated ray tracing function
def ray_trace(sensor_position, ray_direction, environment):
    """
    Finds the first intersection of a ray with the environment and calculates the surface normal.

    Args:
        sensor_position (Vector2): Position of the sensor.
        ray_direction (Vector2): Direction of the ray.
        environment (dict): Environment data with walls and rooms.

    Returns:
        tuple: Closest intersection point (Vector2), distance, and surface normal.
    """
    intersections = []

    # Check intersections with walls
    for wall_name, wall in environment["walls"].items():
        line_start = wall["start"]
        line_end = wall["end"]

        # Detect intersection (placeholder logic, replace with actual intersection calculation)
        intersection_point = Vector2(3, 1)  # Example intersection point
        distance = np.linalg.norm(np.array([sensor_position.x - intersection_point.x, sensor_position.y - intersection_point.y]))
        rho = wall["rho"]  # Reflectivity

        if intersection_point is not None:  # Explicitly check if an intersection exists
            normal = find_surface_normal(line_start, line_end)  # Find the surface normal
            intersections.append((intersection_point, distance, normal))

    # Check intersections with static targets (e.g., circles or other shapes)
    for target in environment["static_targets"]:
        if target["name"] == "circle":  # Example for a circular object
            center = target["center"]
            radius = target["radius"]

            # Calculate intersection and surface normal (placeholder logic)
            intersection_point = Vector2(2, 2)  # Example intersection point
            distance = np.linalg.norm(np.array([sensor_position.x - intersection_point.x, sensor_position.y - intersection_point.
            normal = find_circle_surface_normal(center, intersection_point)

            if intersection_point is not None:  # Explicitly check if an intersection exists
                intersections.append((intersection_point, distance, normal))

    # Find the closest intersection
    closest_intersection = closest_surface(sensor_position, intersections)

    if closest_intersection:
```