Group 4
Nyumba Quest

Our Team

Claire Wachira-150491

Kiptoo Dorcas-151101

Tamara Chelagat-151782

Ian Gichuki-134209

Joey Rutozi- 149772

Project Overview

What is Nyumba Quest?

- A Pygame-based 2D game featuring:
- Exploration of a dynamic house layout.
- Interaction with AI enemies and collectible items.
- Real-time lighting effects with ray casting.
- Highlights: - Simple yet engaging mechanics.
- Strong focus on light interactions and player-object dynamics.

Tools and Frameworks Used

Programming Language: Python

-Graphics Framework: Pygame

-Sound Handling: pygame.mixer for audio effects.

-Code Management: Git/GitHub for collaboration.

Game Features

1.Dynamic Lighting: Implemented using ray casting for real-time light simulation.

2.Game Mechanics: Player movement, health management, and inventory collection.

3.AI Behavior:- AI enemies track and attack players based on proximity.

4.Interactive Environment: Collectibles like treasure and health packs.

Technical Details

- Player Implementation: Supports movement, collision detection, and item collection.
- Ray Casting for Vision: Simulates player's field of view using rays.
- AI Characters: Follow players and decrease health on close encounters.
- House Layout:Consists of walls, windows, and item spawn points.

Challenges and Solutions

Challenges:

1. Synchronizing AI and player interactions.
2. Smooth rendering of real-time lighting.
3. Balancing game difficulty and resource constraints.

Solutions:

1.Efficient data structures for real-time performance.
2.Simplified ray casting for smooth gameplay.
3.Parameterized AI behavior for adaptability

Class Ray

```python
class Ray:
    def __init__(self, pos, angle):
        self.pos = pos
        self.dir = Vector(math.cos(angle), math.sin(angle))

    def set_angle(self, angle):
        self.dir.set(math.cos(angle), math.sin(angle))

    def look_at(self, x, y):
        self.dir.x = x - self.pos.x
        self.dir.y = y - self.pos.y
        self.dir.normalize()

    def draw(self, screen):
        end_x = self.pos.x + self.dir.x * 10
        end_y = self.pos.y + self.dir.y * 10
        pygame.draw.line(screen, (100, 100, 100),
                         (self.pos.x, self.pos.y),
                         (end_x, end_y))

    def cast(self, wall):
        x1, y1 = wall.a.x, wall.a.y
        x2, y2 = wall.b.x, wall.b.y
        x3, y3 = self.pos.x, self.pos.y
        x4 = x3 + self.dir.x
        y4 = y3 + self.dir.y

        den = (x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4)
        if den == 0:
            return None

        t = ((x1 - x3) * (y3 - y4) - (y1 - y3) * (x3 - x4)) / den
        u = -((x1 - x2) * (y1 - y3) - (y1 - y2) * (x1 - x3)) / den

        if 0 <= t <= 1 and u > 0:
            pt = Vector()
            pt.x = x1 + t * (x2 - x1)
            pt.y = y1 + t * (y2 - y1)
            return pt
        return None
```

PlayerClass

Highlights how the player is implemented, including movement, health management, inventory system, and ray casting.

```python
class Player:
    def __init__(self, scene_width, scene_height):
        self.pos = Vector(scene_width // 2, scene_height // 2)
        self.scene_width = scene_width
        self.scene_height = scene_height

        self.rays = []
        self.fov = math.radians(FOV)
        self.heading = 0

        self.health = HEALTH_MAX
        self.inventory = []

        # Create rays for FOV
        for a in range(int(math.degrees(self.fov))):
            ray = Ray(self.pos, math.radians(a))
            self.rays.append(ray)

    def rotate(self, angle):
        self.heading += angle
        for i, ray in enumerate(self.rays):
            ray_angle = math.radians(i) + self.heading
            ray.set_angle(ray_angle)
```

PlayerClass :Move

```
def move(self, walls, amount):
    # Calculate movement vector
    forward = Vector(math.cos(self.heading), math.sin(self.heading))
    forward.normalize()
    movement = forward.copy()
    movement.x *= amount * SPEED_MULTIPLIER
    movement.y *= amount * SPEED_MULTIPLIER

    # Proposed new position
    new_x = self.pos.x + movement.x
    new_y = self.pos.y + movement.y

    # Check for wall collisions
    collision = False
    for wall in walls:
        dist = point_to_line_distance(new_x, new_y, wall.a.x, wall.a.y, wall.b.x, wall.b.y)
        if dist < WALL_COLLISION_THRESHOLD:  # Threshold for collision
            collision = True
            break

    # Move if no collision and within bounds
    if not collision and (0 <= new_x < self.scene_width) and (0 <= new_y < self.scene_height):
        self.pos.x = new_x
        self.pos.y = new_y
    else:
        pass
```

PlayerClass : collect Item

```
def collect_item(self, items):
    for item in items:
        if not item.collected:
            dist = math.sqrt((self.pos.x - item.pos.x)**2 + (self.pos.y - item.pos.y)**2)
            if dist < 15:  # Item pickup range
                item.collected = True
                self.inventory.append(item.type)

                # Special item effects
                if item.type == 'health':
                    item_pickup_sound.play()
                    self.health = min(HEALTH_MAX, self.health + HEALTH_PICKUP_AMOUNT)

                if item.type == 'treasure':
                    treasure_pickup_sound.play()
```

PlayerClass : Ray Casting Logic

Demonstrates how ray casting is used for light and field of view simulation.

```
class Ray:
    def __init__(self, pos, angle):
        self.pos = pos
        self.dir = Vector(math.cos(angle), math.sin(angle))

    def set_angle(self, angle):
        self.dir.set(math.cos(angle), math.sin(angle))

    def look_at(self, x, y):
        self.dir.x = x - self.pos.x
        self.dir.y = y - self.pos.y
        self.dir.normalize()

    def draw(self, screen):
        end_x = self.pos.x + self.dir.x * 10
        end_y = self.pos.y + self.dir.y * 10
        pygame.draw.line(screen, (100, 100, 100),
                         (self.pos.x, self.pos.y),
                         (end_x, end_y))

    def cast(self, wall):
        x1, y1 = wall.a.x, wall.a.y
        x2, y2 = wall.b.x, wall.b.y
        x3, y3 = self.pos.x, self.pos.y
        x4 = x3 + self.dir.x
        y4 = y3 + self.dir.y
```

PlayerClass : Ray Casting Logic

```
den = (x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4)
if den == 0:
    return None

t = ((x1 - x3) * (y3 - y4) - (y1 - y3) * (x3 - x4)) / den
u = -((x1 - x2) * (y1 - y3) - (y1 - y2) * (x1 - x3)) / den

if 0 <= t <= 1 and u > 0:
    pt = Vector()
    pt.x = x1 + t * (x2 - x1)
    pt.y = y1 + t * (y2 - y1)
    return pt
return None
```

PlayerClass : AI character class

Shows the implementation of AI movement and interactions with the player.

```python
class AICharacter:
    def __init__(self, x, y):
        self.pos = Vector(x, y)
        self.speed = AI_SPEED

    def move_towards(self, player_pos):
        # Calculate direction vector towards player
        dir_x = player_pos.x - self.pos.x
        dir_y = player_pos.y - self.pos.y
        mag = math.sqrt(dir_x**2 + dir_y**2)
        if mag > 0:
            dir_x /= mag
            dir_y /= mag

        # Move towards the player
        self.pos.x += dir_x * self.speed
        self.pos.y += dir_y * self.speed

    def draw(self, screen):
        ai_sprite = pygame.image.load('images/enemy.png').convert_alpha()
        player_sprite = pygame.transform.scale(ai_sprite, AI_CHARACTER_SIZE)
        screen.blit(player_sprite, (int(self.pos.x - 16), int(self.pos.y - 16)))
```

Real-World Applications

1. Gaming: Developing 2D adventure games with similar mechanics.
2.Education:Teaching ray tracing concepts.
3.Simulations: Indoor navigation training using dynamic house layouts.

Future Enhancements

1. Add dynamic weather effects (e.g., rain, fog).
2. Implement multiplayer support.
3. Use GPU-based rendering for better performance.
4. Expand levels with procedural generation.

Github:

https://github.com/clairewachira/Nyumba-quest