

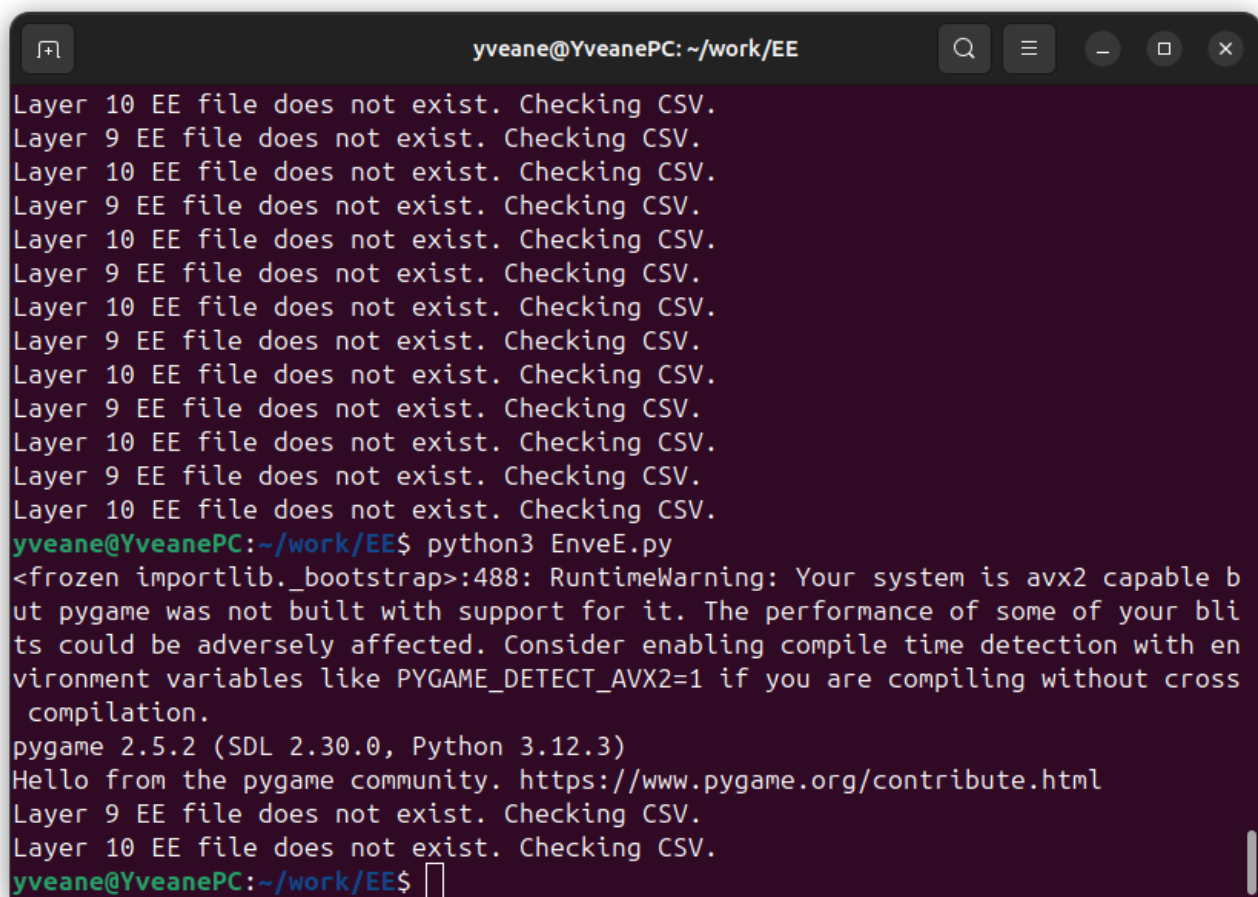
LAB 1 :

Student name: Donfack Tsopfack Yves Dylane

Step 1 - Environment Creation:

The environment in this code operates based on a multi-layered grid system generated through an app that I created, called **EE**. Each layer in this environment serves a specific purpose, such as containing objects, obstacles, the agent, or visual elements for aesthetics. The code for the app can be found on my github repos (<https://github.com/yvesdylane/EnviromentEditor>). The following details each layer and its purpose:

Running EE to Create Environment:



```
yveane@YveanePC: ~/work/EE
Layer 10 EE file does not exist. Checking CSV.
Layer 9 EE file does not exist. Checking CSV.
Layer 10 EE file does not exist. Checking CSV.
Layer 9 EE file does not exist. Checking CSV.
Layer 10 EE file does not exist. Checking CSV.
Layer 9 EE file does not exist. Checking CSV.
Layer 10 EE file does not exist. Checking CSV.
Layer 9 EE file does not exist. Checking CSV.
Layer 10 EE file does not exist. Checking CSV.
Layer 9 EE file does not exist. Checking CSV.
Layer 10 EE file does not exist. Checking CSV.
Layer 9 EE file does not exist. Checking CSV.
Layer 10 EE file does not exist. Checking CSV.
yveane@YveanePC:~/work/EE$ python3 EnveE.py
<frozen importlib._bootstrap>:488: RuntimeWarning: Your system is avx2 capable b
ut pygame was not built with support for it. The performance of some of your bli
ts could be adversely affected. Consider enabling compile time detection with en
vironment variables like PYGAME_DETECT_AVX2=1 if you are compiling without cross
compilation.
pygame 2.5.2 (SDL 2.30.0, Python 3.12.3)
Hello from the pygame community. https://www.pygame.org/contribute.html
Layer 9 EE file does not exist. Checking CSV.
Layer 10 EE file does not exist. Checking CSV.
yveane@YveanePC:~/work/EE$
```

Figure 1: Running and executind EE(EnveE)

1. **Layer 2 (Road):** This layer represents navigable paths where the agent can move freely.
2. **Layer 3 (Target):** The objective or target area that the agent aims to reach.
3. **Layer 5 (Agent):** Contains the agent's current position.
4. **Layer 8 (Obstacles):** Contains obstacles the agent must avoid during pathfinding.
5. **Other Layers (Decorations):** Reserved for visual elements in the environment.

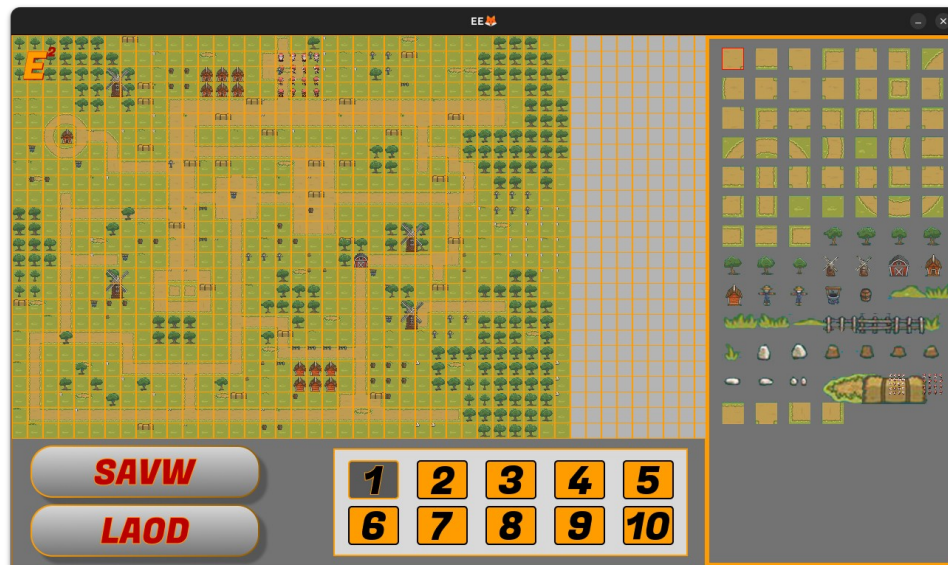


Figure 2: Creation of Environment using EE

Each number in these layers directly corresponds to an image in the image set managed by the EE application, which allows the game to load and display environmental elements correctly based on the agent's actions.

Each layer are now loaded and display by world_loader.py.

 The image shows a code editor window with a project explorer on the left and a code editor on the right. The project explorer shows a folder named 'agent' containing several files, including 'world_loader.py'. The code editor displays the contents of 'world_loader.py', which includes a function 'world_loader()' that loads data from a file or CSV and a function 'draw_world()' that renders the world data.


```

6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
def world_loader(): 2 usages
    world_data = []
    for layer_index in range(10): # Assuming you have 10 layers
        # Load from .EE file if exists
        file_name_ee = f'layers/layer_{layer_index + 1}.data.EE'
        if os.path.exists(file_name_ee): # Check if the file exists
            with open(file_name_ee, 'rb') as pickle_in: # Open file for reading
                current_layer = pickle.load(pickle_in) # Load layer data
                world_data.append(current_layer) # Update the world_data
        else:
            print(f"Layer {layer_index + 1} EE file does not exist. Checking CSV.")
            # Load from CSV file if EE file does not exist
            file_name_csv = f'layers/layer_{layer_index + 1}.data.csv'
            if not os.path.exists(file_name_ee) and os.path.exists(file_name_csv):
                with open(file_name_csv, 'r') as csv_in: # Open file for reading
                    csv_reader = csv.reader(csv_in)
                    current_layer = []
                    for row in csv_reader:
                        # Convert row from string to the appropriate type (if needed)
                        current_layer.append([int(tile) if tile else None for tile in row])
                    world_data.append(current_layer) # Update the world_data
            else:
                if not os.path.exists(file_name_csv):
                    print(f"Layer {layer_index + 1} CSV file does not exist. Skipping.")
    return world_data

def draw_world(world_data, image_list, TILE_SIZE, MAX_ROWS, MAX_COLS, player_layer, sur, scale, agent, ai): 2 usages
    for index, layer in enumerate(world_data): # Iterate over all layers
        for v in range(MAX_ROWS):

```

Figure 3: loading Environment from the layers

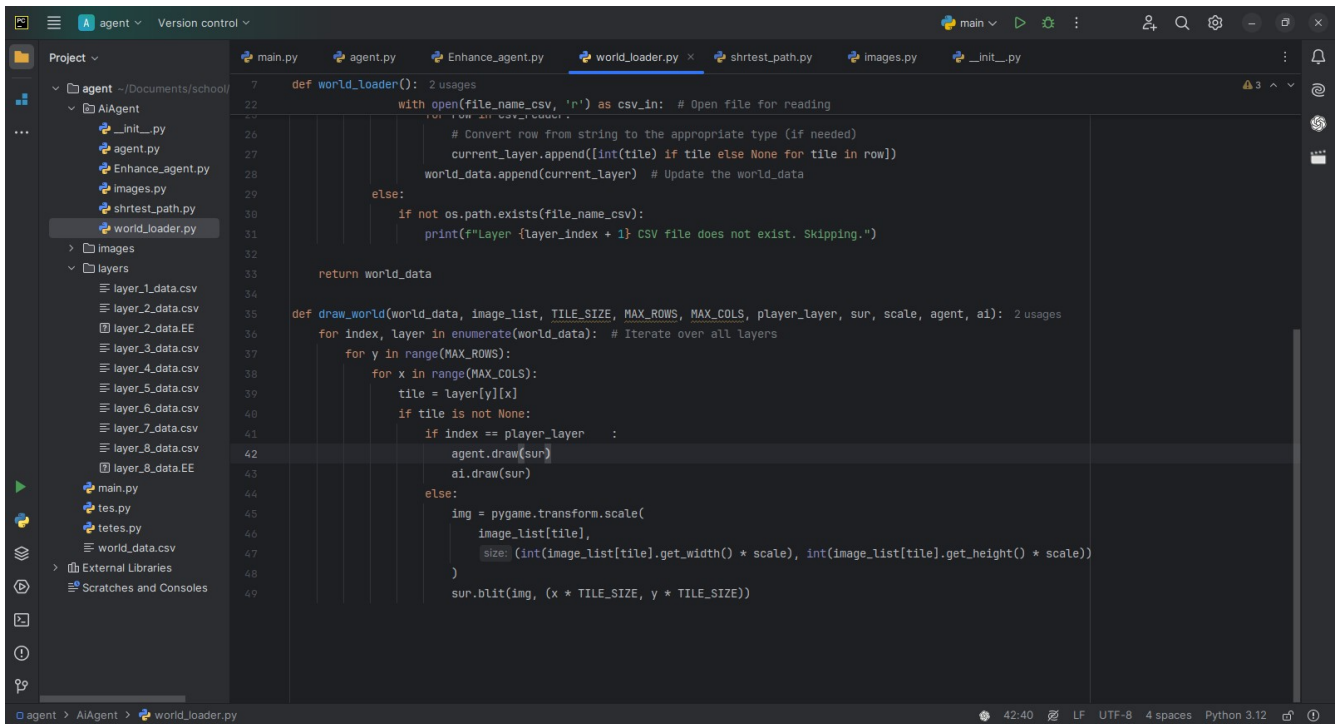


Figure 4: drawing world to the screen

Step2 - Agent Design:

The Agent class have been create with a set of properties and methods. The information need to create an Agent instance are :

- Image : The spring/ image that will be display on the screen for the visual of the Agent
- X and Y: Represent the column and row in the grid world where the agent image will be draw
- Scale : Represent the different/percentage change in size of the image
- Tile Size : Represent the constant that will be use to be multiply the row and col of the image to obtain it image location in pixel

The Agent have a set of method to control how it move, draw, check valid movement and update it position on the display.

```

class Agent:
    def valid_move(self, dir, x, y, world, allow_visited=False):
        # ... (omitted code) ...

    def move(self, world):
        directions = [1, 2, 3, 4]
        unvisited_moves = []
        visited_moves = []

        # Remove blocked or previous directions
        if self.previous_dir in directions:
            directions.remove(self.previous_dir)
        if self.block_dir:
            directions.remove(self.block_dir)

        # First, look for unvisited valid moves
        for direction in directions:
            path = self.valid_move(direction, self.x, self.y, world)
            if path:
                unvisited_moves.append(path)
            else:
                # Check if it's a valid move through a visited tile
                path = self.valid_move(direction, self.x, self.y, world, allow_visited=True)
                if path:
                    visited_moves.append(path)

        # If no unvisited moves, fallback to visited moves
        if not unvisited_moves and visited_moves:
            unvisited_moves = visited_moves

        # If no valid unvisited or visited moves, try backtracking
        if not unvisited_moves:
            path = self.valid_move(self.previous_dir, self.x, self.y, world, allow_visited=True)

        # If valid moves are found, choose one
        if unvisited_moves:
            movement = random.choice(unvisited_moves)
            self.moving = True
            if movement[0] < self.x:
                self.previous_dir = 3 # Moved to the right
                self.dir = 2
            elif movement[0] > self.x:
                self.previous_dir = 2 # Moved to the left
                self.dir = 3
            if movement[1] < self.y:
                self.previous_dir = 1 # Moved upward
                self.dir = 4
            elif movement[1] > self.y:
                self.previous_dir = 4 # Moved downward
                self.dir = 1

            # Update the agent's position and mark the tile as visited
            self.x, self.y = movement
            self.visited.append(movement)
        else:
            # No valid moves, set block_dir to prevent moving back immediately
            self.block_dir = self.dir

```

Figure 5: method for the movement of the Agent

```

import pygame
import random

class Agent:
    def __init__(self, image, x, y, scale, tile_size):
        self.x = x
        self.y = y
        self.tile_size = tile_size
        self.pos_x = self.x * tile_size
        self.pos_y = self.y * tile_size
        self.image = pygame.transform.scale(image, (int(image.get_width() * scale), int(image.get_height() * scale)))
        self.col = 0
        self.row = 0
        self.dir = 1
        self.block_dir = 0
        self.previous_dir = 0
        self.visited = [(x, y)] # Start with the current position as visited
        self.moving = True
        self.ATTAIN_OBJECTIVE = False
        self.width = self.image.get_width() // 4
        self.height = self.image.get_height() // 4
        # Extract the first image (top-left corner of the sprite sheet)
        self.current_image = self.image.subsurface((self.row, self.col, self.width, self.height))

    def draw(self, surface):
        # ... (omitted code) ...

    def verify_objective(self, world):
        # ... (omitted code) ...

    def is_moving(self):
        # ... (omitted code) ...

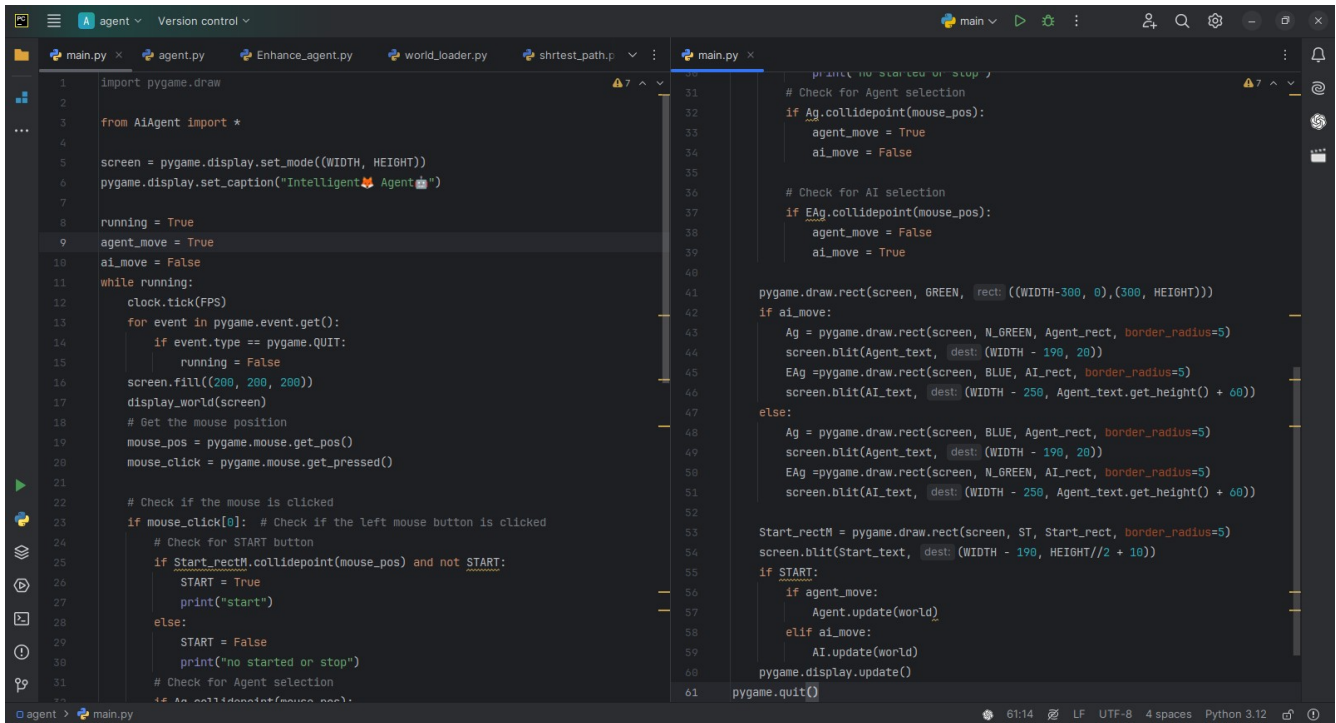
    def update(self, world):
        # ... (omitted code) ...

```

Figure 6: design of the Agent class

Step 3 - Agent Simulation Loop

This loop run the update function every 30 Frame For Per Second.

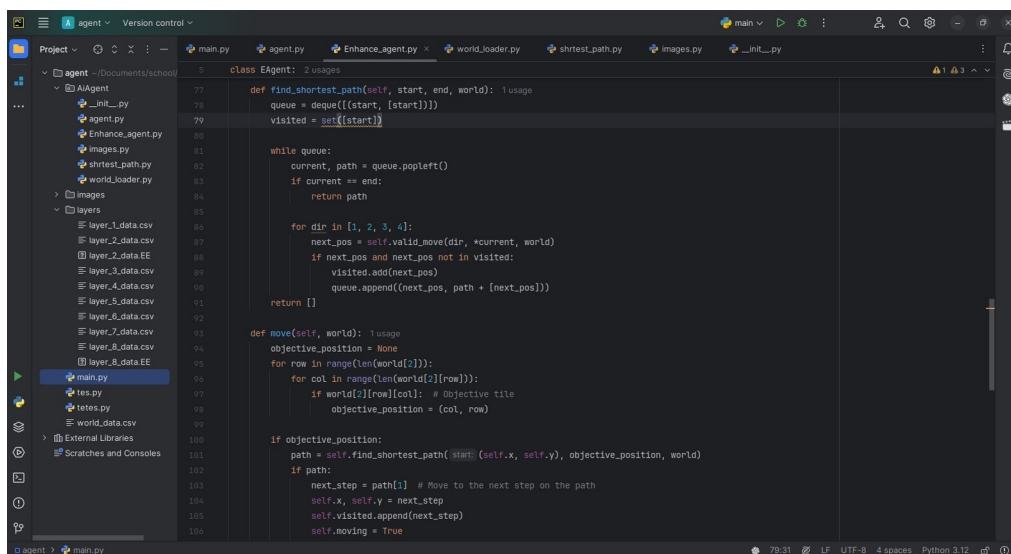


```
1 import pygame.draw
2
3 from AiAgent import *
4
5 screen = pygame.display.set_mode((WIDTH, HEIGHT))
6 pygame.display.set_caption("Intelligent Agent")
7
8 running = True
9 agent_move = True
10 ai_move = False
11 while running:
12     clock.tick(FPS)
13     for event in pygame.event.get():
14         if event.type == pygame.QUIT:
15             running = False
16     screen.fill((200, 200, 200))
17     display_world(screen)
18     # Get the mouse position
19     mouse_pos = pygame.mouse.get_pos()
20     mouse_click = pygame.mouse.get_pressed()
21
22     # Check if the mouse is clicked
23     if mouse_click[0]: # Check if the left mouse button is clicked
24         # Check for START button
25         if Start_rect.collidepoint(mouse_pos) and not START:
26             START = True
27             print("start")
28         else:
29             START = False
30             print("no started or stop")
31     # Check for Agent selection
32     if Ag.collidepoint(mouse_pos):
33         agent_move = True
34         ai_move = False
35
36     # Check for AI selection
37     if EA.collidepoint(mouse_pos):
38         agent_move = False
39         ai_move = True
40
41     pygame.draw.rect(screen, GREEN, rect((WIDTH-300, 0),(300, HEIGHT)))
42     if ai_move:
43         Ag = pygame.draw.rect(screen, N_GREEN, Agent_rect, border_radius=5)
44         screen.blit(Agent_text, dest: (WIDTH - 190, 20))
45         EA = pygame.draw.rect(screen, BLUE, AI_rect, border_radius=5)
46         screen.blit(AI_text, dest: (WIDTH - 250, Agent_text.get_height() + 60))
47     else:
48         Ag = pygame.draw.rect(screen, BLUE, Agent_rect, border_radius=5)
49         screen.blit(Agent_text, dest: (WIDTH - 190, 20))
50         EA = pygame.draw.rect(screen, N_GREEN, AI_rect, border_radius=5)
51         screen.blit(AI_text, dest: (WIDTH - 250, Agent_text.get_height() + 60))
52
53     Start_rect = pygame.draw.rect(screen, ST, Start_rect, border_radius=5)
54     screen.blit(Start_text, dest: (WIDTH - 190, HEIGHT//2 + 10))
55     if START:
56         if agent_move:
57             Agent.update(world)
58         elif ai_move:
59             AI.update(world)
60     pygame.display.update()
61 pygame.quit()
```

Figure 7: Main loop simulating the agent

Step 4 - Path-finding Enhancement:

Using BFS to enhanced to find the short-es path and move in and efficient moaner:



```
77 class EAAgent: 2 usages
78     def find_shortest_path(self, start, end, world): 1 usage
79         queue = deque([(start, [start])])
80         visited = set([start])
81
82         while queue:
83             current, path = queue.popleft()
84             if current == end:
85                 return path
86
87             for dir in [1, 2, 3, 4]:
88                 next_pos = self.valid_move(dir, current, world)
89                 if next_pos and next_pos not in visited:
90                     visited.add(next_pos)
91                     queue.append((next_pos, path + [next_pos]))
92
93             return []
94
95     def move(self, world): 1 usage
96         objective_position = None
97         for row in range(len(world[2])):
98             for col in range(len(world[2][row])):
99                 if world[2][row][col]: # Objective tile
100                     objective_position = (col, row)
101
102         if objective_position:
103             path = self.find_shortest_path(self, (self.x, self.y), objective_position, world)
104             if path:
105                 next_step = path[1] # Move to the next step on the path
106                 self.x, self.y = next_step
107                 self.visited.append(next_step)
108                 self.moving = True
```

Figure 8: shortes path module using BFS

Comparison of Original and Enhanced Pathfinding Methods(BFS)

The original and enhanced methods each approach pathfinding with specific goals in mind. Here’s a breakdown of their differences:

1. Original Pathfinding:

- **Randomized Selection of Movement Directions:** It uses random choices within valid movement directions, only verifying whether a path is visited or unvisited.
- **Efficiency:** This method is simpler but less efficient because it may backtrack or revisit areas without a strong preference for optimal paths.
- **Usage:** Works well in smaller, less complex environments where an exhaustive search is feasible.

2. Enhanced Pathfinding:

- **Layer-Dependent Optimization:** This version takes full advantage of the multi-layered EE environment. It checks each direction based on obstacles, paths, and previously visited points and uses a fallback to expand the search.
- **Breadth-First Search (BFS):** The enhancement includes BFS-based algorithms to find the shortest path. BFS is ideal for finding the shortest path in grid-like environments, and it reduces redundant checks.
- **Backtracking with Fallbacks:** If no direct path is found, it attempts to backtrack while considering all available paths in unvisited and visited lists to ensure all options are exhausted.
- **Use Case:** Ideal for larger environments with multiple obstacles and layers, making it more adaptable to complex grid designs.

Comparison of Methods

| Feature | Original Code | Enhanced Code (BFS) |
|-------------------|---|---|
| Movement Strategy | Random, favors unvisited tiles | Prioritizes shortest path to objective using BFS |
| Efficiency | Inefficient with increased backtracking | More efficient, minimizes revisits and random moves |
| Pathfinding | Random exploration with fallbacks | BFS for shortest path to objective or fallback |
| Best Use Case | Small grid, no specific target | Large grid, target objective within agent's reach |

The enhanced code improves pathfinding efficiency by eliminating random moves when a path to the target is available, thus making the agent faster and more effective in reaching its objective. The original method’s fallback to random movement still remains, providing flexibility if the agent cannot directly reach the target.

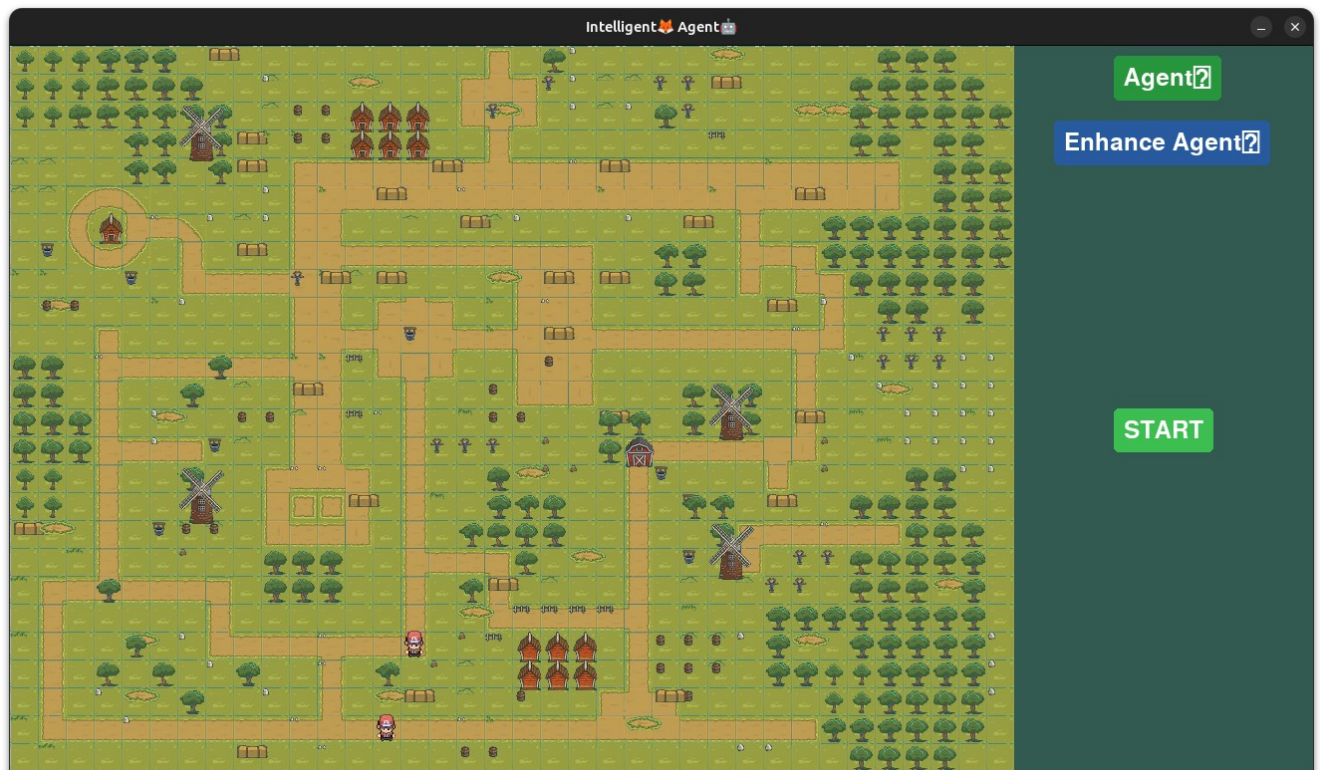


Figure 9: Agent And EnhanceAgent moving in the world