QUIZ 2

Made by:
Haikal Athallarik - 5025221232
Revy Pramana - 5025221252
Jeremy James - 5025221139

Informatics Engineering
Design & Analysis of Algorithms H

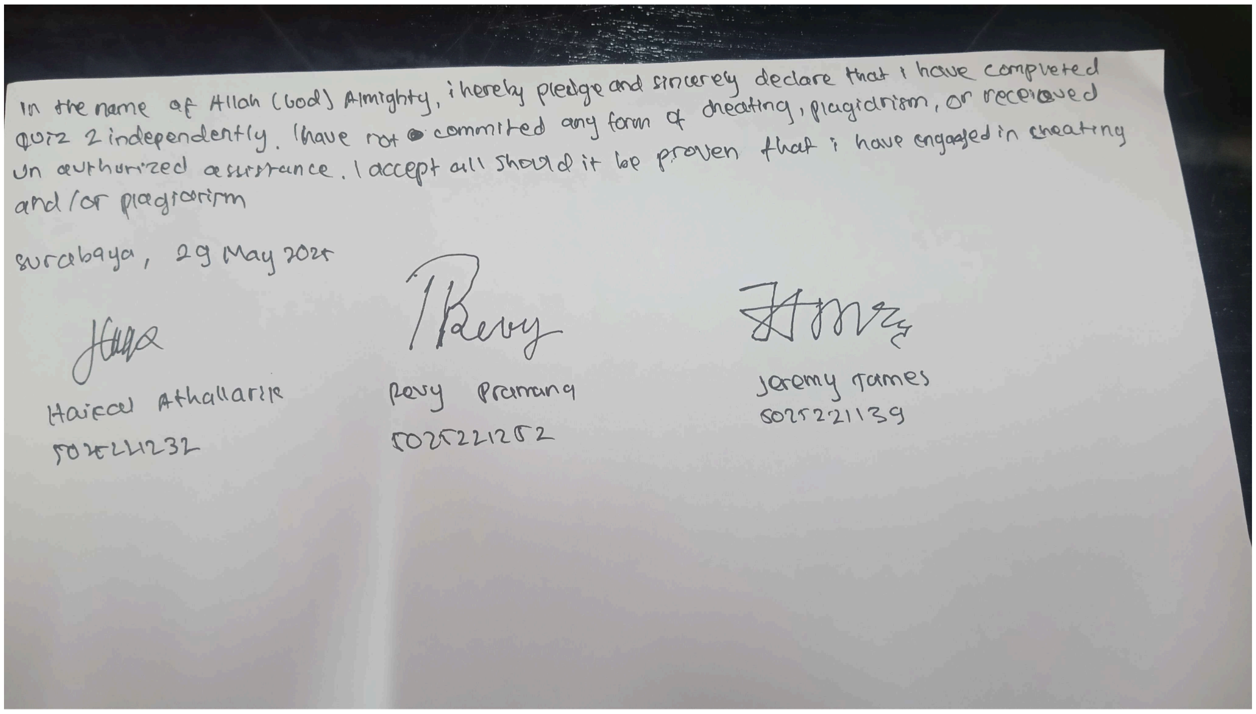
Institut Teknologi Sepuluh Nopember
Surabaya

Project Report: Shapeshifting Puzzle Game

Member Contribution

• Jeremy James—33.33%: Handled BFS Algorithm, Setting up tilemaps, assets, and project initialization.
• Haikal Athallarik—33.33%: Handled A*Star Algorithm, Main Menu Redesign, Integrating Sprites, and Wrote the final report.
• Revy Pramana—33.33%: DFS Algorithm, Main Menu Initialization, and Wrote the final report.

Pledge



In the name of Allah (God) Almighty, i hereby pledge and sincerey declare that i have completed quiz 2 independently. I have not commited any form of cheating, plagiarism, or received un authorized assistance. I accept all should it be proven that i have engaged in cheating and/or plagiarism

Surabaya, 29 May 2024

[illegible] Revy [illegible]

Haikal Athallarik
5025221232

Revy Pramana
5025221252

Jeremy James
5025221139

---

1. Project Design

1.1. **Overview**

The project is a simple shape shifting puzzle game designed to test real searching algorithms for player movements. The player can change forms (human, bat, rat), and each form has different walkable tiles on the game map. The game allows the user to select a pathfinding algorithm (BFS, DFS, or A*) at the start, which is then used to calculate the player's movement to a clicked tile. This project’s movement logic is inspired by the movement in games such as Dota 2, League of Legends, and Runescape.

1.2. **Core Components**

The game is built using Pygame and is structured into several core components:

- **Game Logic (**`core/game.py`**):** Initializes and runs the main game loop, handles updates, drawing, and manages other components like the game state, map renderer, and event handler. It loads the Tiled map and initializes the pathfinder and camera systems.
- **Game State (**`core/game_state.py`**):** Manages the player's current form, grid position, pixel position, target position, and the calculated path. It includes logic for changing forms, checking walkability based on the current form, and updating the player's smooth movement along a path.
- **Map Handling:**
    - **Map Loader (**`core/map/map_loader.py`**):** Loads Tiled JSON maps and, crucially, extracts tile properties for different forms (human, bat, rat) to create a `tile_properties_grid`. This grid determines walkability for each form based on layer and tileset properties in the Tiled map file. It handles default walkability and prioritizes more specific form-based properties if defined.
    - **Map Renderer (**`core/map/map_renderer.py`**):** Responsible for drawing the tile layers from the Tiled map data, the calculated path, and the player character onto the screen, taking into account the camera's viewport.
    - **Tileset Manager (**`core/map/tileset_manager.py`**):** Loads and manages tileset images, providing access to individual tile surfaces for rendering.
    - **Map Files (**`assets/maps/`**):** JSON files exported from the Tiled Map Editor (e.g., `TiledMap1.2_PAA.json`) define the game levels, tile layers, and custom properties for tiles and layers, including walkability for different forms.
- **Pathfinding (**`core/algorithms/pathfinder.py`**):** Implements BFS, DFS, and A* search algorithms. These algorithms find a path from a start to an end coordinate on the grid, considering the current player form's walkability rules. If the target is unreachable or unwalkable, BFS and A* are designed to find a path to the closest reachable tile.
- **Event Handling:**
    - **Event Handler (**`core/events/event_handler.py`**):** Processes Pygame events, delegating input handling to the `InputHandler`.
    - **Input Handler (**`core/events/input_handler.py`**):** Manages mouse clicks (to set a target position and trigger pathfinding) and key presses (to quit, reset, or change player form).
- **Camera (**`core/camera.py`**):** Implements a camera system that smoothly follows the player and handles the conversion between world coordinates and screen coordinates.
- **Settings (**`settings.py`**):** Contains constants for screen dimensions, FPS, sprite settings, grid size, and keybinds.
- **Main (**`main.py`**):** The entry point of the application. It initializes Pygame, displays a menu for the user to select a pathfinding algorithm (DFS, BFS, A*), shows brief information about the chosen algorithm, and then starts the game.

2. Algorithm Analysis

The game implements three pathfinding algorithms: Depth-First Search (DFS), Breadth-First Search (BFS), and A* Search. The user can select which algorithm to use via a startup menu.

2.1. Depth-First Search (DFS)

- **Implementation (`core/algorithms/pathfinder.py`):**
    - Uses a stack to explore paths, going as deep as possible along each branch before backtracking.
    - Considers form-specific walkability by checking the `form_walkable_key` (e.g., `human_walkable`) in the `tile_properties_grid`.
    - If the start or end position is not walkable for the current form, it returns an empty path.
    - It does **not** guarantee the shortest path and is not designed to find the "closest" reachable tile if the target is unwalkable/unreachable; it only finds a direct path if one exists and the target is walkable.
- **Characteristics (from `main.py` menu and general knowledge):**
    - **Pros:** Generally fast execution for finding *a* path, can have lower memory usage compared to BFS in some cases (especially for deep paths with few branches).
    - **Cons:** Does not guarantee the shortest path, can get stuck in very long paths before finding a solution, even if a shorter one exists. The specific implementation returns no path if the exact target tile is not walkable by the current form.

2.2. Breadth-First Search (BFS)

- **Implementation (`core/algorithms/pathfinder.py`):**
    - Uses a queue to explore all neighbors at the present depth prior to moving on to nodes at the next depth level.
    - Guarantees finding the shortest path in terms of the number of steps for unweighted graphs.
    - Consider form-specific walkability.
    - **Modified Behavior:** If the target `end` is unwalkable or unreachable, this BFS implementation attempts to find a path to the closest reachable tile to the `end` based on Manhattan distance.
- **Characteristics (from `main.py` menu and general knowledge):**
    - **Pros:** Guaranteed to find the shortest path (in terms of number of tiles).
    - **Cons:** Can be slower and consume more memory than DFS, especially in graphs with a high branching factor or large search spaces.

*2.3. A Search**

- **Implementation (`core/algorithms/pathfinder.py`):**
  - Uses a priority queue (min-heap) and a heuristic function (Manhattan distance) to guide the search towards the target.
  - Calculates `f_score = g_score + heuristic_score`, where `g_score` is the cost from the start and `h_score` is the estimated cost to the end.
  - Consider form-specific walkability.
  - **Modified Behavior:** Similar to the BFS implementation, if the target `end` is unwalkable or unreachable, this A* implementation attempts to find a path to the closest reachable tile to the `end`.
- **Characteristics (from `main.py` menu and general knowledge):**
  - **Pros:** Optimal path (shortest path, assuming an admissible and consistent heuristic) with generally better performance than BFS in many common scenarios because the heuristic guides the search more directly.
  - **Cons:** Slightly more complex to implement and calculate than BFS or DFS due to the heuristic and priority queue management. Performance depends on the quality of the heuristic.

2.4. **Form-Specific Walkability**

A key feature across all algorithms is their interaction with the tile_properties_grid. This grid, generated by map_loader.py, stores boolean flags like human_walkable, bat_walkable, and rat_walkable for each tile. The pathfinding functions dynamically use the current_form of the player to check the appropriate walkability flag (e.g., if current_form is "bat", it checks bat_walkable). This allows different forms to traverse different parts of the map.

---

3. Source Code Overview

The project is organized into several Python files and directories:

- `main.py`:
  - Entry point of the application.
  - Handles Pygame initialization.
  - Displays an interactive menu for selecting the pathfinding algorithm (DFS, BFS, A*).
  - Initializes and runs the `Game` object.
- `settings.py`:
  - Contains global constants for the game, such as screen dimensions, FPS, grid configuration, and keybinds.
  - Defines each sprite configuration for each Human, Bat, and Rat form

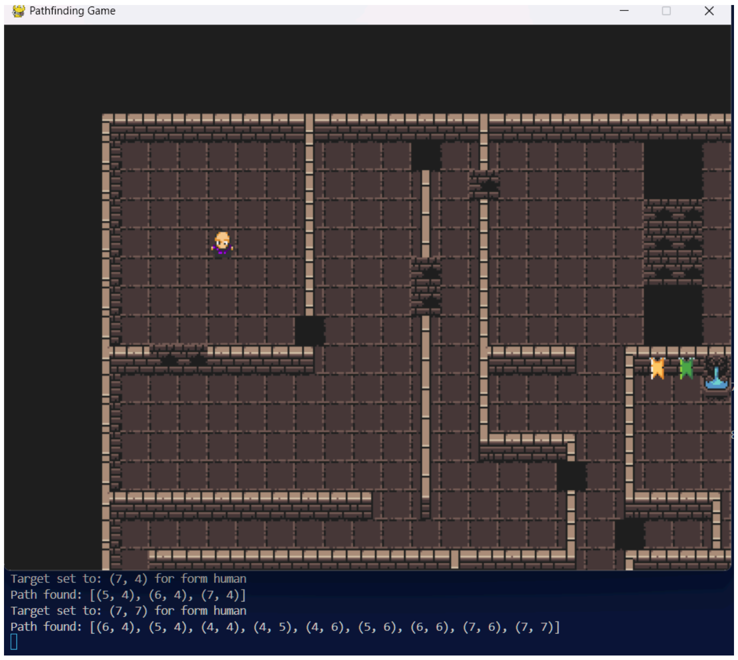
- `requirements.txt`:
  - Lists project dependencies: `numpy` and `pygame`.
- `README.md`:
  - Provides a project description, group member information, setup instructions, and asset credits.
- `core/` **Directory**: Contains the main game logic.
  - `game.py`:
    - Manages the main game loop, game state, rendering, and event processing.
    - Integrates various components like the map loader, tileset manager, pathfinder, and camera.
  - `game_state.py`:
    - Manages dynamic game data: player's current form, position (grid and pixel), target destination, and the calculated path.
    - Handles form switching logic and player movement along the path.
    - Includes `is_walkable` checks based on the `tile_properties_grid` and current form.
  - `camera.py`:
    - Implements a camera that follows the player smoothly and handles viewport calculations.
  - `algorithms/pathfinder.py`:
    - Contains the static methods for BFS, DFS, and A* pathfinding algorithms.
    - These methods take the start/end positions, the `tile_properties_grid`, and the `current_form` as input.
    - BFS and A* include fallback logic to find the closest reachable tile if the target is not directly reachable/walkable.
  - `entities/player.py`:
    - (Note: The content of this file appears to be largely commented out in the provided fullContent. It might have been intended for more complex player logic initially but is not actively used in the current state described by other files like `game_state.py` which handles player position and movement.)
  - `events/event_handler.py`:
    - Processes general Pygame events (like quit, resize).
    - Delegates input events to `InputHandler`.
  - `events/input_handler.py`:
    - Handles specific user inputs:
      - Mouse clicks for setting player target and initiating pathfinding.
      - Key presses for game actions (quit, reset) and form switching.
  - `map/map_loader.py`:
    - Loads Tiled JSON map files.

        - Crucially, processes layers and tilesets to build the `tile_properties_grid` which defines walkability for different player forms based on custom properties set in Tiled.
    - `map/map_renderer.py`:
        - Renders the game map (tile layers), the player's path, and the player character.
        - Uses the `TilesetManager` to get tile images and the `Camera` to adjust drawing positions.
        - Includes an optional debug mode to visualize non-walkable areas for the current form.
    - `map/tileset_manager.py`:
        - Loads tileset images specified in the Tiled map data.
        - Caches and provides individual tile surfaces (subsurfaces) based on their global tile ID (GID).
- `assets/maps/` **Directory**: Contains the Tiled map JSON files (e.g., `TiledMap1.1_PAA.json`, `TiledMap1.2_PAA.json`, `TiledMap1_PAA.json`).
- `assets/sprites/` **Directory:** (Referenced in `game_state.py`, `game_state.py`, and `README.md`)
    - Contains the player sprites used. The `README.md` credits "Sprites by IndigoFenix : https://opengameart.org/content/boundworlds-acolytes-human-and-monster, bagzie : https://opengameart.org/content/bat-sprite, Reemax : https://opengameart.org/content/lpc-rat-cat-and-dog"
- `assets/tilesets/` **Directory**: (Referenced in `tileset_manager.py` and `README.md`)
    - Contains the tileset images used by the Tiled maps. The `README.md` credits "Tileset by 0x72: https://0x72.itch.io/dungeontileset-ii".

---

4. Output Analysis

4.1 Depth First Search (DFS)



```
Target set to: (7, 4) for form human
Path found: [(5, 4), (6, 4), (7, 4)]
Target set to: (7, 7) for form human
Path found: [(6, 4), (5, 4), (4, 4), (4, 5), (4, 6), (5, 6), (6, 6), (7, 6), (7, 7)]
```

1. **Log Segment 1: Path to Target (7, 4)**

   Log Entry:
   Target set to: (7, 4) for form human
   Path found: [(5, 4), (6, 4), (7, 4)]

   - **Path Traversed:** The player moves from (4,4), (5,4), (6,4), (7,4).
   - **Analysis:** The path generated is a direct horizontal line to the right. This output is consistent with DFS behavior under circumstances where the most direct route (according to its neighbor exploration order) is also an open path. The algorithm, upon being called with start=(4,4) and end=(7,4), would explore neighbors of (4,4). If its predefined neighbor search order (e.g., Down, Right, Up, Left) encounters the "Right" direction (5,4) as a viable and unvisited tile, and this branch continues to be open towards the target, DFS will proceed along this branch. In this instance, the path found by DFS coincides with the optimal (shortest) path, which can occur if such a path is the first one fully explored by the algorithm's depth-first traversal.

2. **Log Segment 2: Path to Target (7, 7)**

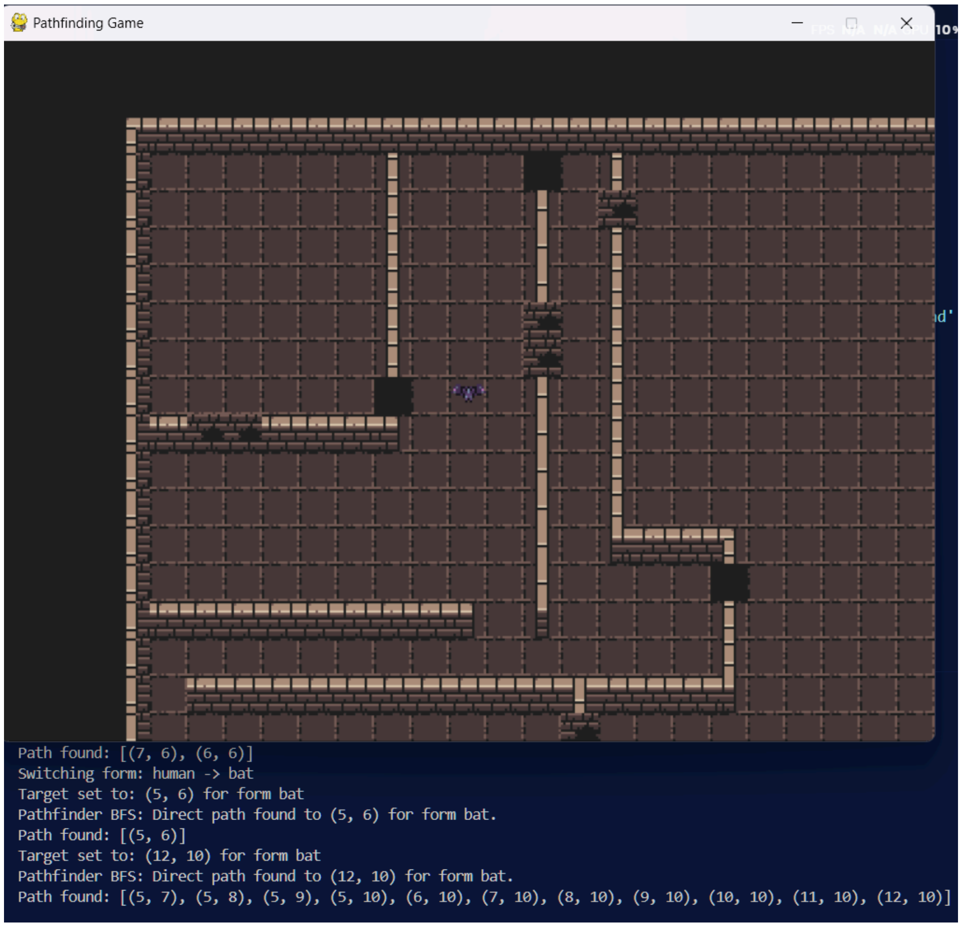
   Log Entry:
   Target set to: (7, 7) for form human
   Path found: [(6, 4), (5, 4), (4, 4), (4, 5), (4, 6), (5, 6), (6, 6), (7, 6), (7, 7)]

- **Player Starting Position:** Following the previous segment, the player is at (7,4).
- **Path Traversed:** The player moves from (7,4), (6,4), (5,4), (4,4), (4,5), (4,6), (5,6), (6,6), (7,6), (7,7).
- **Analysis:** This segment clearly demonstrates a characteristic non-optimal path generated by DFS. The target (7,7) is directly south of the starting position (7,4). A shortest path would be (7,4), (7,5), (7,6), (7,7). The generated path, however, first moves left, then down, then right, and finally down to the target.

  This behavior is explained by DFS's methodology:

  1. **Neighbor Exploration Order:** From (7,4), DFS attempts to explore neighbors in its predefined order. The path (6,4) being the first step indicates that attempts to move Down (7,5), Right (8,4), or Up (7,3) from (7,4) either encountered obstacles (non-walkable tiles for the 'human' form) or those branches were explored and led to dead ends (or did not reach the target before the (6,4) branch did).
  2. **Deep Dive:** Once DFS committed to the "Left" branch by moving to (6,4), it continued to explore this branch deeply: (6,4), (5,4), (4,4). This suggests that from (6,4) and (5,4), exploring further left was prioritized or other immediate options were blocked.
  3. **Path Discovery:** From (4,4), the algorithm found an open route downwards ((4,5), (4,6)), then to the right ((5,6), (6,6), (7,6)), and finally down to the target (7,7). This was the first complete path to the target that DFS discovered through its exhaustive depth-first exploration of the chosen initial branches.

4.2 Breadth First Search (BFS)



```
Path found: [(7, 6), (6, 6)]
Switching form: human -> bat
Target set to: (5, 6) for form bat
Pathfinder BFS: Direct path found to (5, 6) for form bat.
Path found: [(5, 6)]
Target set to: (12, 10) for form bat
Pathfinder BFS: Direct path found to (12, 10) for form bat.
Path found: [(5, 7), (5, 8), (5, 9), (5, 10), (6, 10), (7, 10), (8, 10), (9, 10), (10, 10), (11, 10), (12, 10)]
```

1. **Log Segment 1: Path to Target (8, 4)**

   Log Entry:
   Target set to: (8, 4) for form human
   Path found: [(5, 4), (6, 4), (7, 4), (8, 4)]

   - Inferred Player Starting Position: For the path [(5, 4), (6, 4), (7, 4), (8, 4)] to be generated, with (5,4) as the first step, the player's position immediately before this path calculation must have been (4,4).
   - Path Traversed: The player moves from (4,4) \rightarrow (5,4) \rightarrow (6,4) \rightarrow (7,4) \rightarrow (8,4).
   - Analysis: The path generated is a direct horizontal movement to the right across four tiles. This output is characteristic of BFS when a clear, unobstructed path exists. BFS explores the grid layer by layer, expanding outwards from the start node. From (4,4), to reach (8,4), BFS would identify this straight sequence of walkable tiles as the shortest possible route in terms of the number of moves. The console message "Pathfinder

BFS: Direct path found..." confirms that the target was directly reachable and walkable. This path is optimal.

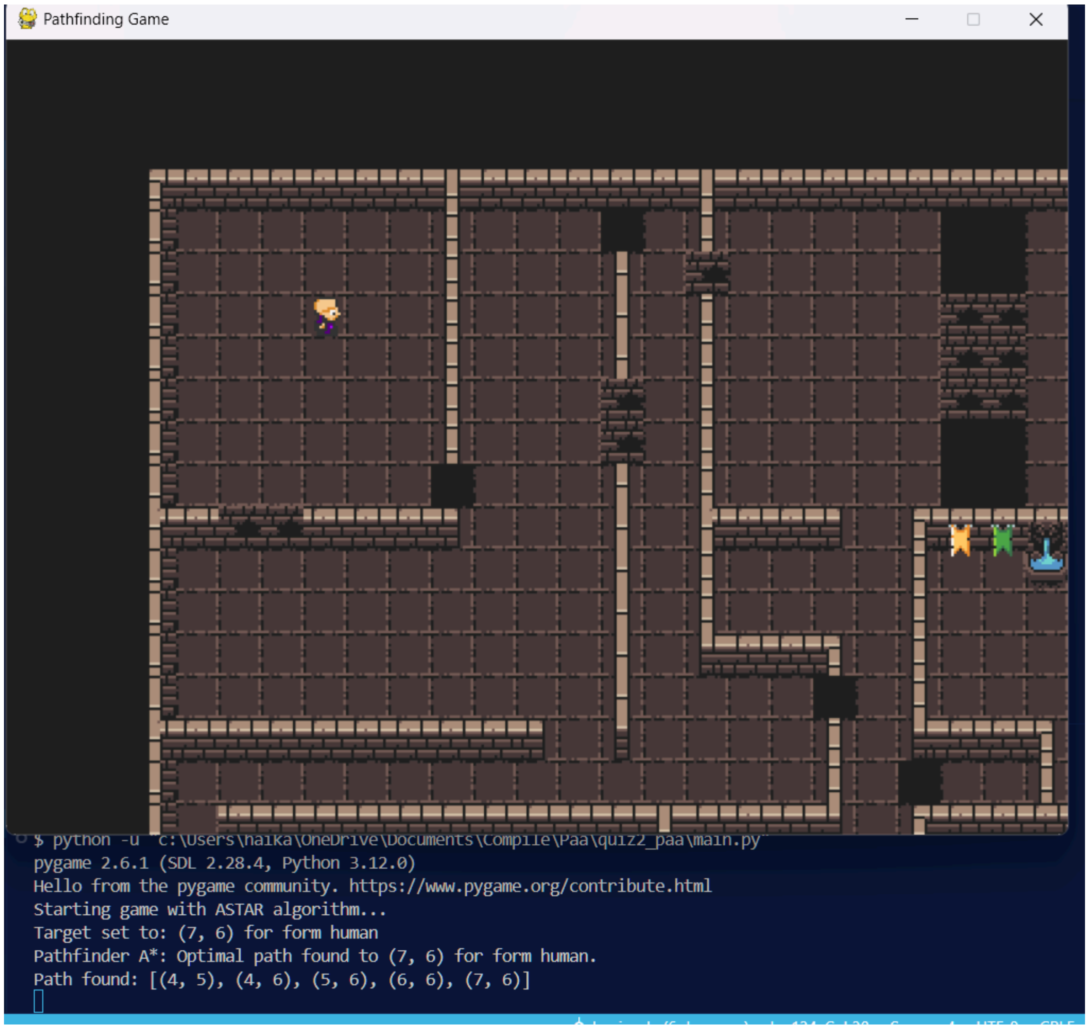
- Player's New Position after Segment 1: (8,4).

2. **Log Segment 2: Path to Target (6, 6)**

   Log Entry:
   Target set to: (6, 6) for form human
   Path found: [(8, 5), (8, 6), (7, 6), (6, 6)]

   - Player Starting Position: The player is at (8,4) (the end point of the previous segment).
   - Path Traversed: The player moves from (8,4), (8,5), (8,6), (7,6), (6,6).
   - Analysis: The path from (8,4) to (6,6) consists of two steps down and then two steps left, forming an "L" shape. This path involves four moves. BFS guarantees finding the shortest path in terms of tile count. The generation of this L-shaped path implies that it is one of the optimal routes (or the only optimal route) given the map's layout and 'human' form walkability. Alternative 4-step paths, such as moving left first then down (e.g., (8,4), (7,4), (6,4), (6,5), (6,6)), would be of equal length. The specific L-shape chosen by BFS depends on the order in which neighbors are added to and processed from its queue. The path (8,4), (8,5), (8,6), (7,6), (6,6) indicates that exploring downwards from (8,4) (to (8,5)) was part of the first optimal path segment identified by the algorithm's layer-by-layer expansion. The console again confirms a "Direct path found".

4.3 A * Star



```
$ python -u "c:\Users\naika\OneDrive\Documents\Compile\Paa\quiz2_paa\main.py"
pygame 2.6.1 (SDL 2.28.4, Python 3.12.0)
Hello from the pygame community. https://www.pygame.org/contribute.html
Starting game with ASTAR algorithm...
Target set to: (7, 6) for form human
Pathfinder A*: Optimal path found to (7, 6) for form human.
Path found: [(4, 5), (4, 6), (5, 6), (6, 6), (7, 6)]
```

1. **Log Segment 1: Path to Target (8, 6)**

   Log Entry:
   Target set to: (8, 6) for form human
   Path found: [(4, 5), (4, 6), (5, 6), (6, 6), (7, 6), (8, 6)]

   - Inferred Player Starting Position: For the path [(4, 5), (4, 6), ..., (8, 6)] to be generated with (4,5) as the first step, the player's position prior to this calculation must have been (4,4).
   - Path Traversed: The player moves from (4,4, (4,5), (4,6), (5,6), (6,6, (7,6, (8,6).
   - Analysis: The path from (4,4) to (8,6) consists of two steps down along column 4, followed by four steps right along row 6, forming an "L" shape. This path involves a total of six moves. A* Search utilizes a heuristic (Manhattan distance in this implementation) in conjunction with the actual cost from the start (g_score) to evaluate nodes using an f_score (f = g + h).

This guides the search preferentially towards nodes that are not only few steps away from the start but also estimated to be close to the target. The generated L-shaped path is an optimal (shortest) route. The console message"Pathfinder A*: Optimal path found..."` confirms this. The specific shape of the path indicates that this was the most promising route according to A*'s evaluation function, likely being as short as any other possible path (e.g., moving right first then down, if unblocked).

- Player's New Position after Segment 1: (8,6).

---

2. **Log Segment 2: Path to Target (8, 9)**

   Log Entry:
   Target set to: (8, 9) for form human
   Path found: [(8, 7), (8, 8), (8, 9)]

   - Player Starting Position: The player is at (8,6) (the endpoint of the previous segment).
   - Path Traversed: The player moves from (8,6), (8,7), (8,8), (8,9).
   - Analysis: The path from (8,6) to (8,9) is a direct vertical movement downwards along column 8, comprising three steps. This is unequivocally the shortest possible path. A*'s heuristic function (Manhattan distance) would strongly favor nodes along this direct column, as any deviation would increase the g_score (actual steps) and likely the h_score (estimated distance to target) or at least make the f_score less competitive than the direct route. The algorithm efficiently identifies this straight path as optimal.

---

5. Conclusion

In conclusion, This project successfully demonstrates the distinct characteristics of Depth-First Search (DFS), Breadth-First Search (BFS), and A* Search algorithms within a dynamic puzzle game environment where player forms dictate unique walkable paths. The analysis confirms that DFS quickly finds *a* path, often non-optimal and failing on unwalkable targets, while both BFS and A* consistently deliver the shortest path, with the added robustness of finding the closest reachable alternative if the direct target is invalid. A*, through its heuristic guidance, generally achieves this optimal path with greater efficiency than BFS. Ultimately, the game effectively showcases these fundamental algorithmic differences in pathfinding quality and behavior, fulfilling its objective as a practical testbed for these search techniques.