Pygame Grid Pathfinding Project Documentation

1. Project Overview

1.1 Purpose

This project implements an interactive visualization of the A* pathfinding algorithm using Python and Pygame. The application allows users to create custom grid layouts, place obstacles, and observe the step-by-step process of finding the shortest path between two points[1][2].

1.2 Key Features

- Interactive grid creation with customizable dimensions
- Two input modes: manual setup or file-based map loading
- Real-time visualization of the A* algorithm execution
- Display of f, g, and h values for each cell during pathfinding
- Export and import functionality for grid layouts
- Zoom and pan controls for large grids (70x70 and above)
- Mouse-based barrier placement and removal
- Visual path reconstruction upon completion

1.3 Educational Value

The project serves as an educational tool for understanding pathfinding algorithms, demonstrating how A* evaluates nodes, calculates heuristics, and explores the search space to find optimal paths.

2. Technical Implementation

2.1 Core Technologies

Technology	Purpose
Python 3.x	Primary programming language
Pygame	Graphics rendering and event handling
PriorityQueue	Managing open set in A* algorithm
Math module	Calculating heuristic distances

Table 1: Technologies used in the project

2.2 Program Architecture

The application follows a modular design with clear separation of concerns:

- **Node Class**: Represents individual grid cells with state management (open, closed, barrier, start, end, path)
- **Grid Management**: Functions for creating, drawing, and updating the grid
- **A* Algorithm**: Implements the pathfinding logic with visualization callbacks
- **Input Handling**: Processes mouse clicks and keyboard commands
- **Rendering System**: Two modes (plain and zoom) for different grid sizes
- **File I/O**: Saves and loads grid layouts in text format

2.3 Coordinate System

The project uses a custom coordinate system where:

- Coordinates start at (1,1) positioned at the bottom-left corner
- The y-axis increases upward (mathematical convention)
- Internal array indexing is converted to/from this logical coordinate system
- This design improves user intuition when specifying positions

3. Algorithm Details

3.1 A* Pathfinding Algorithm

The A* algorithm finds the shortest path by evaluating nodes based on the cost function:

$$f(n) = g(n) + h(n)$$

Where:

- $g(n)$ = actual cost from start node to node n
- $h(n)$ = estimated cost from node n to goal (heuristic)
- $f(n)$ = total estimated cost of path through node n

3.2 Heuristic Function

The implementation uses the Manhattan distance heuristic:

$$h(p_1, p_2) = |x_1 - x_2| + |y_1 - y_2|$$

This heuristic is admissible (never overestimates) and appropriate for grid-based movement in four cardinal directions[3].

3.3 Algorithm Steps

1. Initialize open set with start node
2. While open set is not empty:
    a. Select node with lowest f value
    b. If node is goal, reconstruct and return path
    c. Mark current node as closed
    d. For each neighbor:
        i. Calculate tentative g score
        ii. If better than previous, update scores and add to open set

iii. Update node's g, h, and f values for visualization

3. If open set becomes empty, no path exists

3.4 Neighbor Exploration

Each node explores up to four neighbors (up, down, left, right). Diagonal movement is not permitted in this implementation, ensuring path optimality for Manhattan distance.

4. User Interface

4.1 Display Settings

Parameter	Value
Window Width	1180 pixels
Window Height	780 pixels
Grid Threshold	70x70 (switches to zoom mode)
Base Cell Size	10 pixels
Zoom Range	0.5x to 3.0x

Table 2: Display configuration parameters

4.2 Color Coding System

State	Color	RGB Value
Empty Cell	White	(255, 255, 255)
Start Node	Blue	(0, 0, 255)
End Node	Red	(255, 0, 0)
Barrier	Brown	(150, 75, 0)
Open Set	Green	(0, 255, 0)
Closed Set	Grey	(128, 128, 128)
Path	Pink	(255, 192, 203)
Grid Lines	Grey	(128, 128, 128)

Table 3: Color coding for different cell states

4.3 Control Scheme

Input	Action
Left Mouse Click	Place start/end/barrier
Right Mouse Click	Clear cell
SPACE	Run A* algorithm
S Key	Save map to map.txt
C Key	Clear entire grid
Q Key	Zoom in (large grids only)
E Key	Zoom out (large grids only)
Arrow Keys	Pan camera (large grids only)

Table 4: Keyboard and mouse controls

4.4 Visual Feedback

During algorithm execution, each cell displays:

- Top center: f value (total estimated cost)
- Bottom center: "g+h" breakdown showing individual components

This real-time display helps users understand how the algorithm evaluates different paths.

5. File Format Specification

5.1 Map File Structure

The map export/import system uses a text-based format:

- Each row of the grid corresponds to one line in the file
- Characters represent cell states:
  - '0' = empty cell
  - '1' = barrier
  - 's' = start position
  - 't' = terminal/end position
- Files are saved as map.txt by default

5.2 Example Map File

```
000000s000
0111111100
0100000100
0101110100
0100010100
0111010100
0000010100
0111110100
000000t100
0000000000
```

This format enables easy manual editing, version control, and sharing of grid layouts.

6. Implementation Highlights

6.1 Dual Rendering Modes

The project automatically switches between two rendering strategies:

Plain Mode (grids < 70x70):

- Cells are rendered directly to screen
- Simple mouse-to-grid coordinate conversion
- Efficient for smaller grids

Zoom Mode (grids ≥ 70x70):

- Grid rendered to off-screen surface
- Surface scaled and transformed for display
- Camera offset enables panning
- Maintains stable visualization during algorithm execution

6.2 Visualization Stability

Per user requirements, the zoom and camera position remain fixed during algorithm execution, allowing clear observation of cell value changes without disorienting motion.

6.3 Input Validation

The program validates user input for:

- Grid dimensions (positive integers)
- Start and end positions (within grid bounds)
- File existence and format (when loading maps)
- Invalid positions trigger re-prompts with helpful messages

6.4 Node Class Design

The Node class encapsulates:

- Position data (row, column, pixel coordinates)
- State management (color-based state representation)
- Neighbor relationships
- Algorithm values (g, h, f scores)
- Drawing logic with text overlay

This design enables clean separation between grid logic and visualization.

7. Usage Instructions

7.1 Starting the Program

1. Run the Python script
2. Choose input mode:
    - Enter 'y' to load from file (provide filename)
    - Enter 'n' for manual setup

7.2 Manual Setup Mode

1. Enter grid dimensions (rows columns, e.g., "30 40")
2. Enter start position (row column, using 1-based indexing)
3. Enter end position (row column, using 1-based indexing)
4. Use mouse to place additional barriers (left-click)
5. Press SPACE to run the algorithm

7.3 File Loading Mode

1. Provide the map filename (e.g., "map.txt")
2. Program loads grid with predefined start, end, and barriers
3. Optionally modify layout with mouse
4. Press SPACE to run the algorithm

7.4 Observing the Algorithm

- Green cells show the open set (nodes being considered)
- Grey cells show the closed set (already evaluated)
- Cell labels update in real-time showing f, g, and h values
- Pink cells appear when path is reconstructed
- For large grids, use Q/E to zoom and arrow keys to pan

8. Design Decisions

8.1 Bottom-Left Origin

The coordinate system places (1,1) at the bottom-left to align with mathematical conventions, making the interface more intuitive for users familiar with Cartesian coordinates.

8.2 Manhattan Distance Heuristic

Manhattan distance was chosen because:

- It matches the four-directional movement constraint
- It is admissible (guarantees optimal path)
- It is computationally efficient
- It provides good performance for grid-based pathfinding

8.3 Real-Time Value Display

Displaying g, h, and f values during execution serves educational purposes, allowing observers to understand the algorithm's decision-making process at each step.

8.4 Export Format Simplicity

The text-based export format was chosen for:

- Human readability and manual editing
- Easy version control with standard tools
- Simple parsing logic
- Cross-platform compatibility

9. Performance Considerations

9.1 Algorithm Complexity

The A* algorithm has time complexity of $O(b^d)$ where:

- b = branching factor (4 in this implementation)
- d = depth of solution

Space complexity is $O(b^d)$ for storing nodes in open and closed sets.

9.2 Visualization Overhead

The project includes a small delay (1ms) during algorithm execution to make the visualization observable. This can be adjusted for faster/slower demonstration speeds.

9.3 Large Grid Handling

For grids of 70x70 or larger, the zoom rendering mode:

- Reduces direct screen writes
- Enables efficient scaling transformations
- Maintains responsive user interaction
- Allows inspection of specific areas via pan/zoom

10. Learning Outcomes

Through this project, the following concepts were explored and implemented:

- **Algorithm Implementation**: Converting A* pseudocode into working Python code
- **Data Structures**: Using priority queues for efficient node selection
- **Event-Driven Programming**: Handling user input with Pygame's event system
- **Object-Oriented Design**: Structuring code with classes for maintainability
- **Coordinate Transformations**: Converting between logical and array indexing
- **Real-Time Visualization**: Updating display during algorithm execution
- **File I/O**: Serializing and deserializing grid states
- **User Interface Design**: Creating intuitive controls and visual feedback

11. Potential Extensions

Future enhancements could include:

- Additional pathfinding algorithms (Dijkstra, BFS, DFS) for comparison
- Diagonal movement with appropriate distance metrics
- Weighted cells with varying traversal costs
- Multiple start/end points
- Maze generation algorithms
- Performance metrics display (nodes explored, time elapsed)
- Animation speed control
- Undo/redo functionality for grid editing

12. Conclusion

This Pygame-based pathfinding visualizer successfully demonstrates the A* algorithm through an interactive, educational interface. The implementation balances code clarity with performance, provides flexible input options, and maintains visualization stability for effective learning. The project showcases practical application of algorithmic concepts, data structures, and software design principles in creating an educational tool.