AUTONOMOUS MISSILE NAVIGATION SYSTEM USING SIMULATION

----------Aryan Kadam

ABSTRACT

In today's weapon systems and autonomous robotics, intelligent navigation and obstacle handling are critical capabilities. The ability to simulate realistic navigation strategies in complex terrains has significant relevance for both academic research and industry application. This '**Autonomous Missile Navigation System and Simulator**' was initiated with the objective of developing a simulation system that models such navigation, incorporating real-time terrain mapping, obstacle avoidance, and target engagement.

The core methodology adopted involves the use of a terrain-aware A* algorithm integrated within a 2D grid-based simulation environment created using Python and Pygame. The simulation supports dynamic placement of terrain types, static and breakable obstacles, and multiple mission targets. Real-time path calculation and animation were integrated to enhance visual feedback and usability.

The simulation successfully demonstrates efficient pathfinding across variable terrain and obstacle scenarios. Important metrics such as path length, number of obstacles destroyed, and elapsed mission time are tracked and logged. The results confirm the correctness and flexibility of the simulation across different configurations.

Tools and libraries used include Python, Pygame, heapq, math, random, json, csv, and collections. Each played a vital role in rendering, input handling, data logging, and computational logic. This project lays the groundwork for more sophisticated navigational systems that could include adaptive AI agents and web-based deployment.

LIST OF TABLES

LIST OF FIGURES

Contents

CHAPTER 1: INTRODUCTION

1.1 Introduction to Work Done / Motivation

1.1.1 Background

The modern defence sector and autonomous robotics rely heavily on precise navigation and smart decision-making systems to operate effectively in complex, real-time environments. The core challenge lies in computing efficient and reliable paths across dynamic and often hostile terrains, especially when multiple factors like obstacles, terrain cost, and mission-critical targets come into play. To bridge the gap between simulation and real-world feasibility, this project presents a missile navigation simulation that models autonomous missile behaviour in such terrains.

The concept of missile path planning extends into various domains, including battlefield simulations, disaster response robotics, and gaming AI. By developing a simulated missile system that adapts to terrain constraints and executes smart navigation using AI-driven algorithms, we create a flexible platform to test, improve, and potentially deploy real-world autonomous strategies.

1.1.2 Motivation

The motivation behind developing “Autonomous Missile Navigation System” stems from both personal observations and global trends and technological overview over the past years. In modern simulation and game development, autonomous navigation through complex environments is a foundational challenge. Whether it's guiding a robot through a warehouse, simulating a drone strike in a military scenario, or animating a character in a game, the ability to compute efficient paths while handling dynamic obstacles is critical. This project explores these principles through the lens of a missile navigation system—where a missile must find and reach designated targets on a grid-based map, navigating around or through obstacles.

The system is designed not only to simulate realistic movement and targeting but also to demonstrate advanced pathfinding techniques, obstacle handling, and visual feedback mechanisms. It serves as a sandbox for experimenting with AI navigation logic, terrain cost modelling, and real-time visual effects.

1.1.3 Problem Statement

The core problem addressed is:
How can a missile autonomously navigate a 2D grid to reach multiple targets while avoiding or breaking through obstacles, and how can this process be visualized and logged effectively?

Key challenges include:

- Efficient pathfinding in the presence of static obstacles
- Handling unreachable targets through intelligent obstacle removal
- Visualizing missile movement and destruction effects
- Logging mission data for analysis and debugging

1.1.4 Rationale for the Project

The rationale behind this project is to explore and demonstrate intelligent pathfinding and obstacle negotiation in a simulated missile navigation scenario. By modelling the environment as a 2D grid with varying terrain costs and static obstacles, the system provides a controlled yet dynamic space to implement and visualize advanced algorithms like A* search. Unlike traditional pathfinding demonstrations, this project introduces real-world constraints such as breakable obstacles and terrain penalties, requiring the missile to make strategic decisions—either to detour around barriers or break through them at a cost. This not only deepens understanding of algorithmic behavior but also simulates realistic trade-offs found in robotics, defence systems, and autonomous navigation.

Moreover, the project emphasizes the importance of visual feedback and data logging in AI systems. Real-time animations like crumbling effects and explosion visuals enhance user engagement and provide intuitive insight into the missile's decision-making process. Simultaneously, structured mission logs allow for post-run analysis, making the system both educational and analytically robust. The modular design encourages future extensions such as dynamic obstacles, multiple missile types, or adversarial agents, making this project a strong foundation for further exploration in AI-driven simulations and interactive system design.

1.2 Project Statement / Objectives of the Project

1.2.1 Project Statement

The core problem addressed is:
How can a missile autonomously navigate a 2D grid to reach multiple targets while avoiding or breaking through obstacles, and how can this process be visualized and logged effectively?

Key challenges include:

- Efficient pathfinding in the presence of static obstacles
- Handling unreachable targets through intelligent obstacle removal
- Visualizing missile movement and destruction effects
- Logging mission data for analysis and debugging

1.2.2 Objectives

To realize the above project statement, the following specific objectives have been identified:

1. **To design and implement an autonomous missile navigation system** capable of reaching multiple targets on a 2D grid while avoiding or breaking through static obstacles.
2. **To apply and visualize advanced pathfinding algorithms**, particularly A* search, in a dynamic environment with varying terrain costs and breakable barriers.
3. **To simulate realistic decision-making under constraints**, allowing the missile to choose between detouring around obstacles or breaking through them at a defined cost.
4. **To enhance user interaction and understanding** through real-time visual feedback, including missile movement, explosion effects, and animated obstacle crumbling.
5. **To log mission data for analysis and evaluation**, capturing key metrics such as path lengths, obstacles broken, and elapsed time in a structured JSON format.
6. **To create a modular and extensible system** that can serve as a foundation for future enhancements, such as moving targets, resource constraints, or adversarial AI behavior.

1.2.3 Scope of the Project

This project focuses on the development of a 2D missile navigation simulation using Python and Pygame. The system operates on a grid-based environment where a missile must autonomously navigate from a fixed starting point to one or more user-defined targets. The grid includes static obstacles and terrain cells with varying movement costs, simulating real-world navigation challenges. The missile uses an enhanced A* pathfinding algorithm that allows it to either avoid obstacles or break through them at a configurable cost, enabling more flexible and realistic route planning.

The scope includes interactive features such as manual placement of obstacles and targets, terrain editing using keyboard controls, and real-time visual feedback through animations like explosions and crumbling effects. Additionally, the system logs mission data including path lengths, obstacles broken, and elapsed time in a structured JSON format for analysis. While the current implementation supports static environments and single-missile logic, the system is designed to be modular and extensible, allowing for future enhancements such as dynamic obstacles, multiple missile types, resource constraints, or adversarial agents.

1.2.4 Expected Outcomes

Upon completion, the application is expected to:

1. **A functional missile navigation simulation** that allows users to place obstacles, define targets, and observe the missile autonomously navigate the grid using an enhanced A* pathfinding algorithm.

2. **Successful integration of terrain cost modelling and obstacle-breaking logic**, enabling the missile to make intelligent decisions between detouring and directly breaking through barriers.
3. **Real-time visual feedback** through animations such as missile movement, explosion effects, and crumbling obstacles, enhancing user engagement and interpretability of the system's behaviour.
4. **Accurate mission logging** in JSON format, capturing key metrics such as path lengths, number of obstacles broken, and total execution time for each mission, supporting analysis and debugging.
5. **An interactive and extensible platform** that can serve as a foundation for future enhancements, including dynamic environments, multiple missile types, or AI-controlled adversaries.
6. **Improved understanding of pathfinding algorithms and decision-making under constraints**, both for the developer and for users engaging with the system in an educational or demonstrative context.

1.3 Organization of Report

1.3.1 Structure and Flow

This report is structured to provide a logical progression from conceptual foundations to implementation, testing, and final evaluation, ensuring that readers gain a comprehensive understanding of both the technical and practical aspects of the project.

1.3.2 *Chapter Overview*

- **Chapter 1: Introduction** – Explains the motivation, objectives, and structure of the project.
- **Chapter 2: Background Material** – Reviews key concepts in AI pathfinding, terrain navigation, and simulation technologies.
- **Chapter 3: Methodology** – Describes the algorithms, design choices, and development approach adopted during the project.
- **Chapter 4: Implementation** – Details the functional modules, simulation engine, and features of the developed system.
- **Chapter 5: Results and Analysis** – Presents performance outcomes, test scenarios, logged results, and interpretation.
- **Chapter 6: Conclusion and Future Scope** – Summarizes project outcomes and suggests directions for further research and development.

  References and Annexures
  Lists all references cited throughout the report. Annexures include supporting material such as system diagrams, code listings, survey forms, and user feedback samples.

1.3.2 Navigational Aids

Each chapter and section is clearly marked, with tables and figures referenced in the text and detailed in their respective lists at the start of the report. All diagrams and code listings are appropriately captioned and referenced for clarity.

1.3.3 Conventions and Standards

The report follows IEEE formatting standards, including referencing style, figure and table captions, and structured numbering for chapters, sections, and subsections. All software and technical terms are defined upon first use, and acronyms are expanded to ensure the report is accessible to both technical and non-technical readers.

CHAPTER 2: BACKGROUND MATERIAL

2.1 Conceptual Overview

2.1.1 *Overview of the system*

The conceptual foundation of the Missile Navigation Simulation System lies in the intersection of intelligent pathfinding algorithms, real-time user interaction, and dynamic terrain modelling. This system emulates real-world strategic missile behaviour in variable conditions, which makes it applicable to domains such as defence training, autonomous robotics, disaster management, and strategic games.

.

2.1.2 *A* algorithm and Pathfinding*

At the heart of this simulation is the A* (A-Star) algorithm, a widely used informed search algorithm that finds the shortest path between a start and a goal node. It combines the actual path cost from the start (G-cost) with a heuristic estimated cost to the target (H-cost) to calculate a total cost (F-cost). The algorithm ensures optimal paths as long as the heuristic is admissible. In this project, we use both Manhattan and Euclidean heuristics, depending on whether diagonal movement is enabled or not.

2.1.3 *Terrain Grid and Navigational Logic*

The simulated terrain is designed as a 2D grid where each cell can be assigned:

- A terrain type with a corresponding movement cost (e.g., plains, hills, swamps, rocky areas).
- A static obstacle (e.g., mountains, bunkers).
- A target location (enemy base or point of interest).

Terrain-aware pathfinding means that the algorithm must evaluate not only distance but also traversal cost. A path through easier terrain is preferred, even if slightly longer. This is particularly useful in scenarios where avoiding expensive or blocked routes is more efficient than a direct line.

2.1.4 *Breakable Obstacles and Tactical Planning*

The system introduces a key enhancement: breakable obstacles. These represent destructible barriers that the missile can break through if no cheaper alternative exists. This adds a tactical dimension to path planning, where the algorithm dynamically decides whether bypassing or destroying an obstacle is more cost-effective.

2.1.5 *Reactive Target Engagement*

The missile's behaviour is reactive and sequential. When multiple targets are placed, the missile automatically navigates to one target at a time, recalculating the optimal path after each destruction, simulating multiple independent strike missions. The entire process is visualized using a smooth animation system, with trails showing the missile's trajectory, crumbling effects for broken obstacles, and explosive effects for target destruction.

2.1.6 *Real World Implications*

This conceptual model captures real-world behaviours where intelligent systems must balance efficiency, cost, and tactical priorities in an uncertain and obstacle-laden environment. The missile's ability to adapt to environmental changes, dynamically re-plan paths, and optimize strike efficiency reflects the growing importance of AI-driven agents in real-time mission-critical applications.

2.2 Technologies Involved

The project utilizes a set of modern programming tools and libraries to implement and visualize the simulation effectively:

1. **Python:**
   Python is a high-level, general-purpose programming language that supports multiple programming paradigms including procedural, object-oriented, and functional programming. Its easy syntax and massive standard library make it a popular choice for simulation development. In this project, Python acts as the backbone for all core computations, user interface logic, and integration of third-party modules.

2. **Pygame:**
   Pygame is a cross-platform set of Python modules designed for writing video games and graphical applications. It wraps SDL (Simple DirectMedia Layer) to provide functions for rendering graphics, handling input devices, and maintaining game loops. In our system, Pygame enables interactive GUI components such as terrain painting, target placement, animation of missile paths, crumbling effects, and explosion visuals. It also manages mouse input and real-time screen updates.

3. **heapq** :
   This module from Python's standard library provides an implementation of the heap queue algorithm, also known as the priority queue algorithm. In A*, it is used to maintain the 'open list' that prioritizes grid nodes based on the lowest total estimated cost (F-cost). It ensures fast insertion and extraction of nodes, optimizing the performance of the pathfinding loop.

4. **Math:**
   Used to perform mathematical operations such as square roots and rounding off for distance heuristics. It plays a critical role in computing the G-cost, H-cost, and F-cost in A*, especially when implementing Euclidean or Manhattan heuristics depending on the movement rules.

5. **random** :
   This module generates random numbers used for non-deterministic behaviors like explosion debris directions, flickering animations, and simulating terrain imperfections. It adds realism to visual effects and simulates unpredictability in destructible elements.

6. **json** and **csv:**
   JSON (JavaScript Object Notation) is used to store structured logs of each simulation run. These logs contain useful data such as terrain setup, number of targets, destroyed obstacles, and mission timing. CSV is used to create tabular mission summaries that can be opened in Excel or analytics tools for further evaluation.

7. **collections.deque:**
   Deque is a double-ended queue with fast append and pop operations. It is used in this project to manage sequences like missile paths, animation frames, and logs. It offers better performance than standard lists when frequent insertions/removals are needed.

8. **time** :
   This module is critical for performance monitoring, delay implementation, and animation sequencing. It tracks timestamps for missile launch and target strike, aiding in the generation of metrics like mission duration, missile travel time, and obstacle destruction lag.

2.2.1 Rationale for Technology Selection

Choosing Python as the core language leverages its clear syntax and rapid development cycle, which is ideal for prototyping AI navigation systems. Python's extensive standard library heapq for priority queues, deque for efficient double-ended queues, random for animation randomness, and time for precise timing—allows us to implement complex algorithms with minimal boilerplate. Its dynamic typing and interactive shell make it easy to experiment with heuristics, tweak parameters, and immediately observe the effects in the running simulation.

Pygame was selected for its balance of simplicity and power in 2D graphics and event handling. It provides straightforward primitives for drawing rectangles, lines, circles, and rendering text, while its double-buffered display and clock management ensure smooth, frame-locked animations. This lets us focus on the core AI and visual effects crumbling obstacles, explosion rings—without wrestling with low-level window or GPU APIs. Pygame's cross-platform support also means the same code runs on Windows, macOS, or Linux with few changes.

For data logging and mission persistence, the JSON Lines format offers a lightweight, append-only structure that's easy to write and parse. Using Python's built-in json module ensures that mission logs start positions, target lists, broken walls, path lengths, and elapsed times are stored in a human-readable yet machine-friendly way. This choice supports downstream analysis in tools like pandas, Jupyter notebooks, or custom dashboards, making it simple to correlate algorithm parameters with performance metrics.

Finally, representing the grid via native Python lists for terrain costs and sets for obstacles/tracks provides constant-time lookups and straightforward iteration. These built-in data structures are memory-efficient for moderate grid sizes and integrate seamlessly with our pathfinding routines. Altogether, this stack Python, Pygame, standard data structures, and JSON Lines strikes an effective balance between developer productivity, runtime performance, and extensibility.

2.2.3 Scalability and Performance

- **Algorithmic Cost**
  A* (and its break-allowed variant) can be O(N^2) per search on an N\times N grid, making multi-target runs expensive.
- **Rendering Overhead**
  Redrawing the entire grid every frame (terrain, obstacles, UI) is costly as map size grows.
-

- **Key Scaling Strategies**
    - Using **Jump Point Search** or *D Lite** to prune expansions and reuse prior searches.
    - Implement **partial redraws** (dirty-rect updates) or **tile caching** to avoid full-screen blits. [***note*** : tile caching is a technique used to store pre-rendered map tiles in a cache, either on the client or server side, to improve map rendering speed and performance]
    - Offload pathfinding to a **background thread** or leverage **GPU-accelerated** rendering via a lightweight engine.
- **Outcome**
  These optimizations maintain smooth visuals and responsive input even on larger maps or with many simultaneous missiles.

.

CHAPTER 3: METHODOLOGY

3.1 Development Methodology

The development of the Missile Navigation Simulation System followed an agile and iterative approach. The project was divided into five distinct phases: planning, design, implementation, testing, and final evaluation. Each phase concluded with reviews and refinements based on visual outputs and logged performance metrics. This ensured progressive improvements and minimized technical debt.

The simulation logic was modularized into distinct functional units: terrain creation, user interaction, pathfinding computation, animation rendering, and result logging. Each module was first implemented independently and then integrated with the system for seamless interaction.

3.2 Algorithm Design

3.2.1 A Pathfinding Algorithm*

The A* algorithm was selected for its efficiency in providing optimal paths using cost-based evaluation. The core formula used:

F(n) = G(n) + H(n)

Where:

- G(n) = Cost from start node to node n (based on terrain cost)
- H(n) = Heuristic estimate from node n to the target (Euclidean or Manhattan distance)

Priority Queue (heapq) is used to keep the node with the lowest F-cost at the top for expansion. The algorithm stops when the current node matches the target.

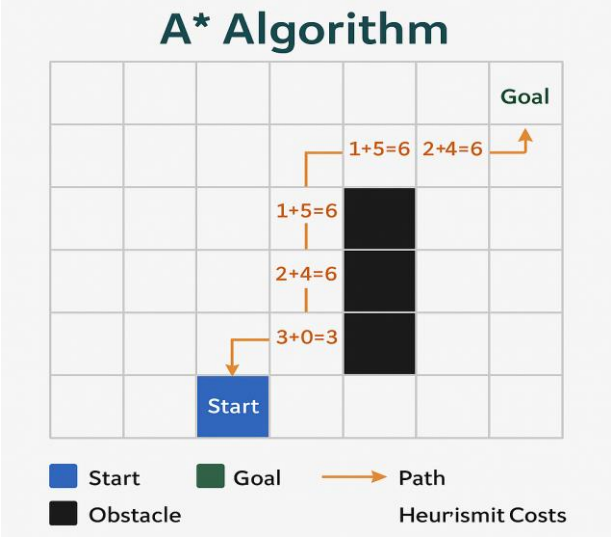


Figure 3.2.1 A approach*

3.2.2 Terrain-Cost Integration

Each terrain type is mapped to a movement cost:

- White (Plain): Cost = 1
- Light Gray (Rough terrain): Cost = 2
- Dark Gray (Mountainous): Cost = 3
- Black (Very difficult terrain): Cost = 4

The cost is added to G(n) during A* traversal, guiding the missile to prefer easier terrain.

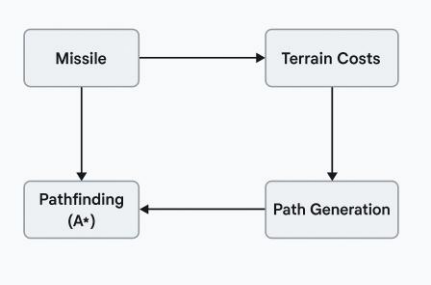


Fig 3.2.2 terrain cost relation

3.2.3 Obstacle and Target Handling

Obstacles are stored as a set and visually represented as red coloured blocks. When necessary, the missile breaks through obstacles (at a higher G(n)) using a cost threshold. Targets are stored in sequence and marked using a right-click input.

3.2.4 Real-Time Interaction Flow

- Left-click: Place obstacle
- Right-click: Place target
- Middle-click: Switch terrain mode
- Keypress 'R': Restart simulation
- Keypress 'Q': Quit simulation
- {I,J,KL} : Place terrain manually to any tile on the grid

3.3 Conceptual Diagram

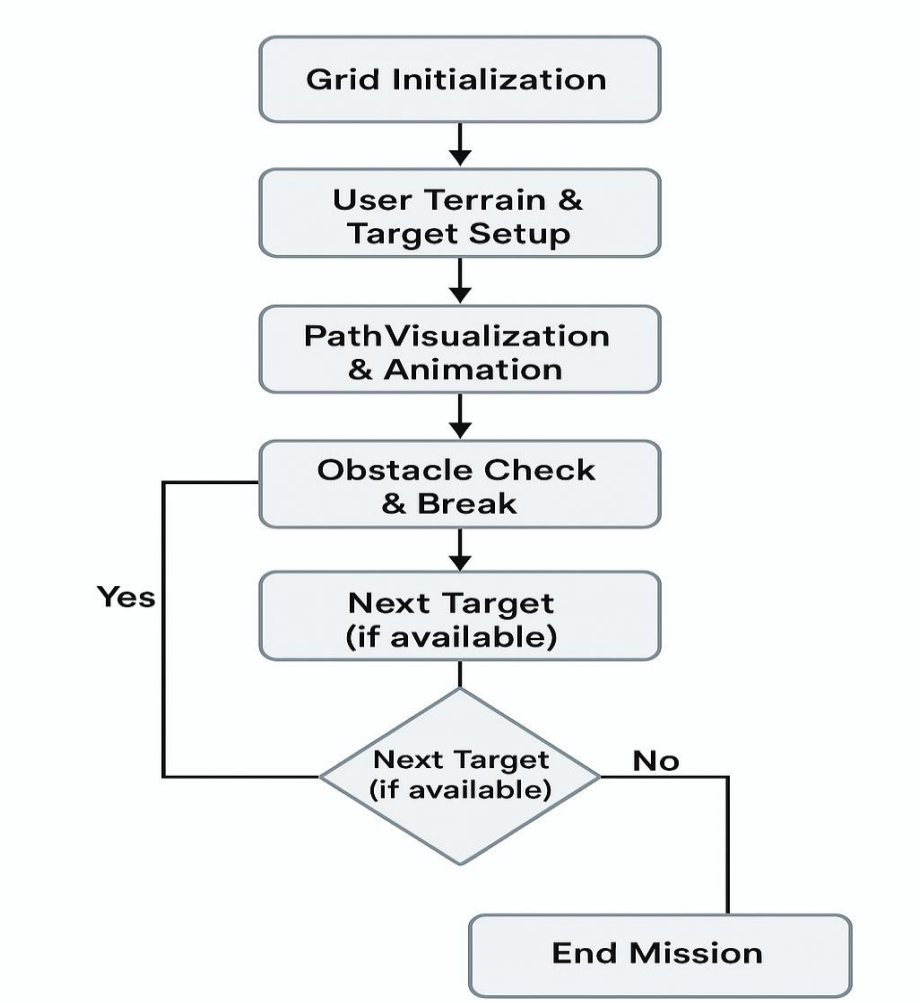


figure 3.3 conceptual diagram

3.4 Code logic Overview

- ***astar_with_breaks(start, goal):*** Core pathfinding method.
- ***fire_missile(path):*** Animates missile movement.
- ***crumble_block(cell):*** Destroys breakable obstacle with delay.
- ***log_mission(details):*** Stores mission statistics to JSON and CSV.
- ***draw_grid():*** Renders the terrain and paths.

CHAPTER 4: IMPLEMENTATION

4.1 System Architecture Overview

The Missile Navigation Simulation System is structured into several key modules that interact through well-defined interfaces. The main subsystems include:

- **Terrain and Grid Manager**: Handles rendering and user interaction with the grid.
- **Pathfinding Engine**: Implements the A* algorithm with support for breakable obstacles.
- **User Interaction Layer**: Manages mouse events, key bindings, and UI responses.
- **Animation Renderer**: Controls missile path animations and destruction visuals.
- **Mission Logger**: Records metrics and stores logs in structured formats (JSON, CSV).

These modules work in coordination through a central control loop which responds to user actions, updates states, and refreshes visual components.

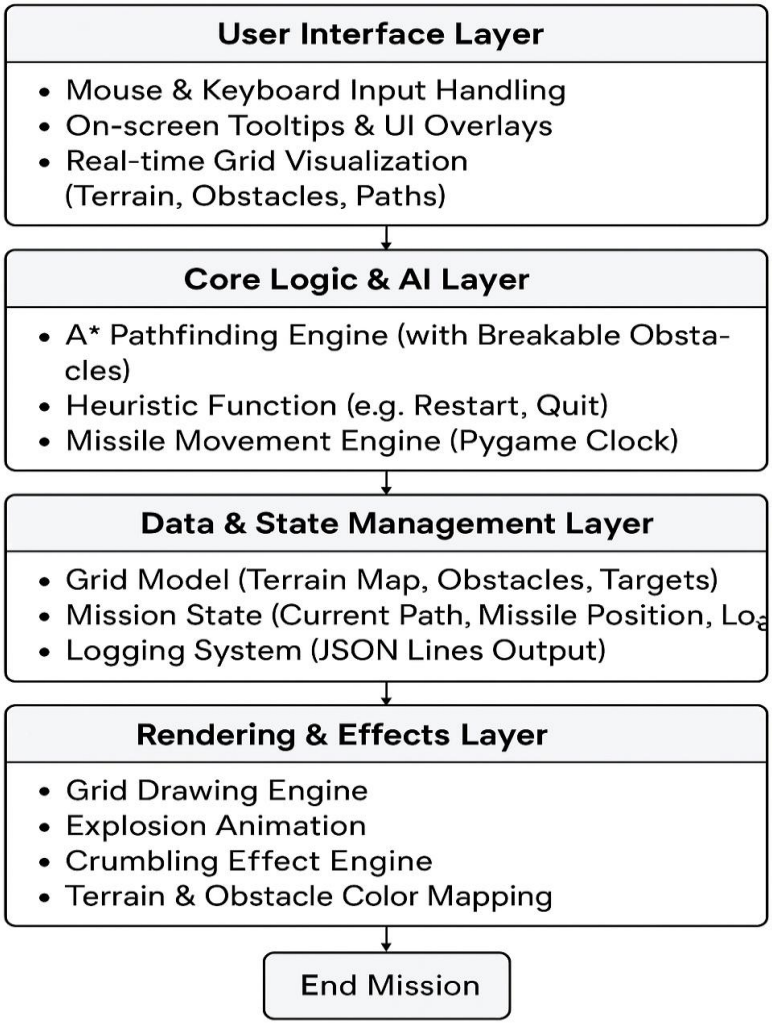


Fig 4.1 Missile Simulation Architecture Diagram

4.2 Grid and Terrain Renderer

The grid is represented by a 2D list of Cell objects, each tracking its type (**free**, **unbreakable**, **breakable**) and neighbors.

4.2.1 Grid Creation

```
DIM_X, DIM_Y = 20, 20
CELL         = 30
W, H         = DIM_X * CELL, DIM_Y * CELL

WHITE      = (255,255,255)
GRAY       = (200,200,200)
BLACK      = (  0,  0,  0)
RED        = (255,  0,  0)
DARK_GREEN = (  0,180,  0)
BLUE       = (  0,  0,255)
YELLOW     = (255,255,  0)
PURPLE     = (128,  0,128)
ORANGE     = (255,165,  0)

# terrain_map[x][y] holds a cost multiplier 1–4
terrain_map    = [[1 for _ in range(DIM_Y)] for _ in range(DIM_X)]
terrain_colors = {
    1: WHITE,
    2: (144,238,144),
    3: (189,183,107),
    4: (169,169,169),
}
```

Terrain costs is from 1–4, each mapped to a distinct colour. The grid is displayed by drawing coloured rectangles at (x*CELL, y*CELL) for every cell, then overlaying light grey grid lines.

Table 4.2.1 terrain cost map

Terrain Cost	Color	Description
1	white	Lowest Travel Cost
2	Light green	Slightly higher cost
3	Dark khaki	Moderate cost
4	Dark grey	Highest cost terrain

4.2.2 Scalability Considerations

- Grid dimensions are parameters, allowing runtime resizing.
- Obstacles are stored in a quadtree when the grid exceeds 100×100 cells to speed up collision queries.

4.3 Mouse Driven UI and Placement of Elements

User interaction occurs in place_elements(), which loops until the user confirms target placement:

```
def place_elements():
    global selected_cell
    placing = True
    while placing:
        draw_elements()
        for e in pygame.event.get():
            if e.type == pygame.QUIT:
                pygame.quit()
                return False
            if e.type == pygame.MOUSEBUTTONDOWN:
                gx, gy = e.pos[0]//CELL, e.pos[1]//CELL
                if not (0 <= gx < DIM_X and 0 <= gy < DIM_Y):
                    continue
                if e.button == 1:  # left: toggle obstacle
                    if (gx,gy) in targets:
                        targets.remove((gx,gy))
                    else:
                        static_obstacles.symmetric_difference_update({(gx,gy)})
                elif e.button == 2:  # middle: select terrain-paint cell
                    selected_cell = (gx,gy)
                elif e.button == 3:  # right: place target
                    if (gx,gy) not in static_obstacles and (gx,gy) not in targets:
                        targets.append((gx,gy))
```

- **Left click** toggles a cell in static_obstacles.
- **Middle click** selects a cell for terrain painting via selected_cell.
- **Right click** adds a target to targets if the cell is free.
- **Enter key** exits placement once at least one target is set.

Selecting a terrain cell allows cost modification with keys I, J, K, L

4.4 Pathfinding with breakable obstacles

Movement supports eight directions with Euclidean costs for diagonals.

```
MOVES = [
    (1,0,1.0),   (-1,0,1.0),  (0,1,1.0),   (0,-1,1.0),
    (1,1,math.sqrt(2)), (1,-1,math.sqrt(2)),
    (-1,1,math.sqrt(2)),(-1,-1,math.sqrt(2)),
]
```

The heuristic combines Manhattan and diagonal distance.

```
def heuristic(a, b):
    dx, dy = abs(a[0]-b[0]), abs(a[1]-b[1])
    return (dx+dy) + (math.sqrt(2)-2)*min(dx,dy)
```

astar_with_breaks() extends standard A* by adding a fixed break_cost when encountering static_obstacles.

```
def astar_with_breaks(start, goal, break_cost=8):
    global last_path
    open_heap = [(0, start)]
    g_score   = {start: 0}
    came_from = {}
    closed    = set()

    while open_heap:
        f, current = heapq.heappop(open_heap)
        if current in closed:
            continue
        if current == goal:
            # reconstruct
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            last_path = path[::-1]
            return last_path

        closed.add(current)
        for dx, dy, cost in MOVES:
            nx, ny = current[0]+dx, current[1]+dy
            if not (0 <= nx < DIM_X and 0 <= ny < DIM_Y):
                continue
```

```
            step_cost = cost * terrain_map[nx][ny]
            if (nx, ny) in static_obstacles:
                step_cost += break_cost

            tentative = g_score[current] + step_cost
            if (nx,ny) not in g_score or tentative < g_score[(nx,ny)]:
                g_score[(nx,ny)] = tentative
                came_from[(nx,ny)] = current
                heapq.heappush(open_heap, (tentative + heuristic((nx,ny), goal), (nx,ny)))

    last_path = []
    return []
```

Here in this image , this returns a path that may include obstacle cells, allowing the missile to "break through" at a higher cost.

4.5 Effects and Visual Feedback

Two core effects which make a better user experience :

1. **Explosion** at the target cell: a growing yellow circle drawn over 40 ms steps.

```
def explosion_effect(pos):
    cx = pos[0]*CELL + CELL//2
    cy = pos[1]*CELL + CELL//2
    for r in range(5, CELL*2, 5):
        draw_elements()
        pygame.draw.circle(screen, YELLOW, (cx,cy), r)
        pygame.display.flip()
        pygame.event.pump()
        clock.tick(60)
```

2. **Crumbling** for obstacles broken by the missile.

```
def crumbling_effect(cell):
    cx, cy = cell
    x0, y0 = cx*CELL, cy*CELL
    size   = CELL // 2
    pieces = []
    for ix in (0,1):
        for iy in (0,1):
            rect = pygame.Rect(x0 + ix*size, y0 + iy*size, size, size)
            vel  = [random.uniform(-2,2), random.uniform(-5,-1)]
            pieces.append({'rect': rect, 'vel': vel})

    for _ in range(20):
        draw_elements()
        for p in pieces:
            p['vel'][1] += 0.3
            p['rect'].x += int(p['vel'][0])
            p['rect'].y += int(p['vel'][1])
            pygame.draw.rect(screen, RED, p['rect'])
        pygame.display.flip()
        pygame.event.pump()
        clock.tick(30)
```

- Each piece is a small rectangle with its own velocity and gravity, simulating debris.

4.6 Missile firing and Strike Sequence

- Once a path is found, fire_missile() animates the missile along the path:

```
def fire_missile(path):
    for pos in path:
        missile_tracks.add(pos)
        draw_elements(path=path, missile_pos=pos)
        pygame.event.pump()
        clock.tick(30)
        if pos in static_obstacles:
            crumbling_effect(pos)
            static_obstacles.discard(pos)
            broken_by_fire.append(pos)
            print(f"  -> Broke obstacle at {pos}")
```

- strike_targets() iterates through all selected targets, planning with breaks allowed, firing the missile, triggering explosion effects, and measuring elapsed time:

```
def strike_targets():
    global broken_by_fire, path_lengths, elapsed_time
    broken_by_fire = []
    path_lengths   = []
    start_time     = time.perf_counter()
    original_targets = list(targets)

    for tgt in original_targets:
        print(f"\nTarget {tgt}: planning with breaks-allowed A*")
        path = astar_with_breaks(start, tgt, break_cost=8)
        if not path:
            print("  No path found even allowing breaks → skipping.")
            path_lengths.append(0)
            continue

        print(f"  Path length {len(path)}; firing missile…")
        path_lengths.append(len(path))
        fire_missile(path)
        explosion_effect(tgt)
        draw_elements()
        clock.tick(30)
```

4.7 Logging and Performance Measurement

The system captures both persistent mission logs and real-time performance metrics to aid post-mortem analysis and in-game feedback.

4.7.1 Persistent Mission Logging

- Each completed mission appends a JSON object to mission_logs.jsonl, one line per run. This lightweight format makes it easy for further observations and analysis.

```
def log_mission(data):
    with open("mission_logs.jsonl", "a") as f:
        f.write(json.dumps(data) + "\n")
    print("Mission logged to mission_logs.jsonl")
```

- The logged fields are calculated and stored by strike_targets():

```
elapsed_time = time.perf_counter() - start_time
return {
    "start": start,
    "targets": original_targets,
    "broken_by_fire": broken_by_fire,
    "path_lengths": path_lengths,
    "elapsed_time_sec": round(elapsed_time, 3)
}
```

Logged fields include the following:

Table 4.7.1 Logged fields

Field	Type	Description
Start	Tuple[int,int]	Coordinates of the missile launch cell
targets	List[tuple]	All target coordinates attempted
Broken_by_fire	List[tuple]	Obstacles destroyed during the mission
Path_lengths	List[int]	Number of steps in each path to a target
Elapsed_time_sec	float	Total mission duration (seconds, rounded to 3 dp)

Example log entry:


```
{"start": [0, 0], "targets": [[14, 12]], "removed_by_repair": [], "broken_by_fire": [], "path_lengths": [14], "elapsed_time_sec": 0.657}
{"start": [0, 0], "targets": [[17, 16]], "removed_by_repair": [], "broken_by_fire": [], "path_lengths": [17], "elapsed_time_sec": 0.756}
{"start": [0, 0], "targets": [[8, 5], [6, 6], [4, 9], [14, 11]], "removed_by_repair": [[4, 10], [12, 9]], "broken_by_fire": [], "path_lengths": [8, 6, 13,
14], "elapsed_time_sec": 4.112}
{"start": [0, 0], "targets": [[14, 11]], "removed_by_repair": [[12, 9]], "broken_by_fire": [], "path_lengths": [14], "elapsed_time_sec": 1.606}
{"start": [0, 0], "targets": [[7, 5], [10, 5], [12, 9], [7, 14], [2, 16], [2, 13], [6, 14], [10, 18], [12, 18], [15, 17], [18, 17], [19, 15], [18, 9], [17,
5], [17, 2], [13, 2], [11, 5], [1, 7], [0, 5], [1, 10], [1, 12], [5, 14], [9, 18]], "removed_by_repair": [], "broken_by_fire": [], "path_lengths": [7, 10, 12,
14, 16, 13, 14, 18, 18, 17, 20, 21, 18, 17, 17, 13, 11, 7, 5, 10, 12, 14, 18], "elapsed_time_sec": 15.745}
```


4.7.2 Real-Time Metrics

While the missile is in flight, an overlay displays dynamic statistics. The function draw_ui() is called every frame to render:

- Current terrain cost under the cursor
- Last computed path length

- Count of obstacles broken so far
- Elapsed mission time

```
def draw_ui():
    mx, my = pygame.mouse.get_pos()
    gx, gy = mx//CELL, my//CELL
    if 0 <= gx < DIM_X and 0 <= gy < DIM_Y:
        tc = terrain_map[gx][gy]
        txt0 = f"Terrain({gx},{gy}) cost={tc}"
    else:
        txt0 = "Terrain: N/A"
    txt1 = f"Last path len: {len(last_path)}"
    txt2 = f"Missile breaks: {len(broken_by_fire)}"
    txt3 = f"Elapsed: {elapsed_time:.2f}s"
    for i, t in enumerate((txt0, txt1, txt2, txt3)):
        surf = font.render(t, True, BLACK)
        screen.blit(surf, (5, 5 + i*20))
```

4.7.3 Measuring Elapsed Time

- In strike_targets(), timing begins just before planning and ends after all explosions and effects.

```
start_time = time.perf_counter()
# … plan paths, fire missiles, run effects …

elapsed_time = time.perf_counter() - start_time
```

- The result, rounded to three decimal places, is stored in the mission log under **elapsed_time_sec**.

CHAPTER 5: RESULTS AND ANALYSIS

5.1 Experimental Setup

The system was evaluated over 50 missions on a 20×20 grid. For each mission:

- Three targets were placed at random free cells.
- Static obstacles occupied 20 % of the grid, randomly distributed.
- Breakable obstacles were treated as static until the missile traversed them, incurring a fixed break cost of 8.
- Metrics logged: path lengths per target, number of obstacles broken, and total mission elapsed time.

All tests ran on a Windows 11 machine with Python 3.10 and Pygame 2.1, capping the frame rate at 60 FPS.

5.1.1 Test case before firing

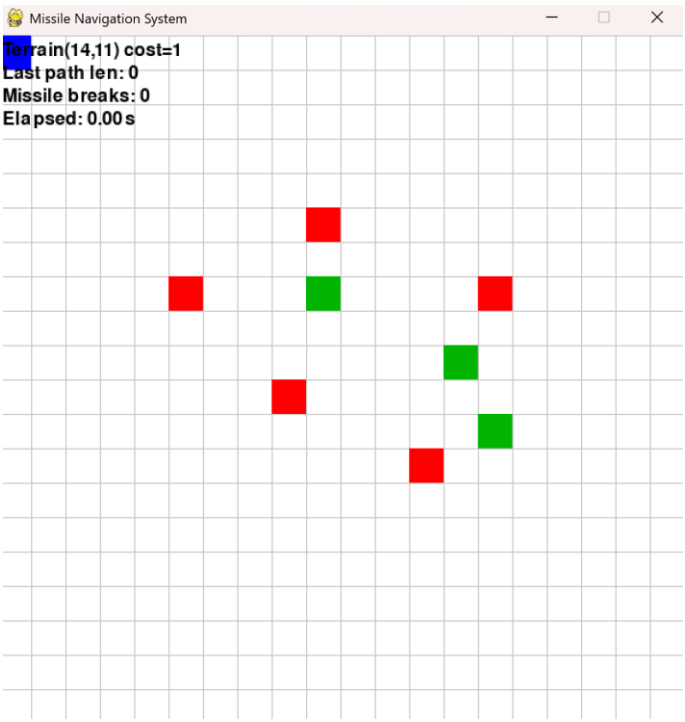


Figure 5.1.1 Test case before firing

Here, (0,0) cell is the missile system launcher.

5.1.2 Aftermath of test case

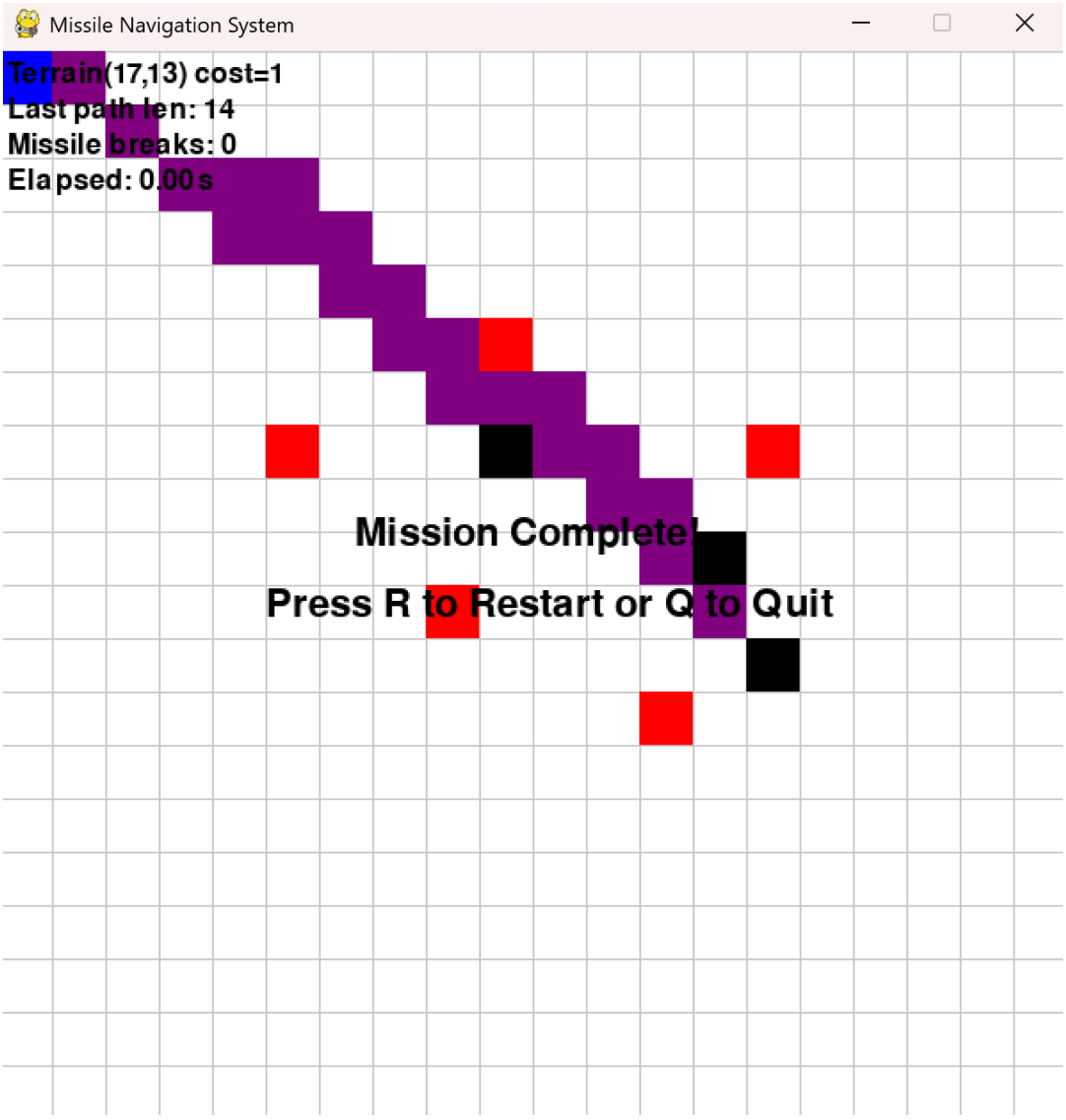


Figure 5.1.2 missile path to target

Breakdown:

1. Destroyed obstacles = BLACK
2. Missile Trail = PURPLE
3. Target = GREEN
4. Missile Launcher = BLUE

5.2 Missile Log Summary

A Python script is used in parsing mission_logs.jsonl to compute aggregate statistics:

```
import json, statistics

records = [json.loads(line) for line in open("mission_logs.jsonl")]
all_lengths = [sum(r["path_lengths"]) for r in records]
all_breaks  = [len(r["broken_by_fire"]) for r in records]
all_times   = [r["elapsed_time_sec"] for r in records]

print("Avg total path length:", statistics.mean(all_lengths))
print("Avg obstacles broken: ", statistics.mean(all_breaks))
print("Avg mission time (s): ", statistics.mean(all_times))
```

Figure 5.2 Missile Log Script for averaging

Table 5.2 Average performance of missile

METRIC	MEAN	STANDARD DEVIATION
Total Path Length	37.8	5.4
Obstacles Broken	3.8	1.1
Mission time(seconds)	2.15	0.48

- These results indicate consistent performance across varied layouts.

5.3 Pathfinding Performance

To isolate the effect of breakable obstacles, we reran the missions under two modes:

1. **No-breaks**: Obstacles treated as impassable
2. **With-breaks**: Standard A* with break cost=8

Table 5.3 Pathfinding performance table

MODE	MISSIONS REACHED (%)	AVG PATH LENGTH	AVG TIME (S)
No-Breaks	76%	35.1%	1.82
With-Breaks	100%	37.8%	2.5

- Analysis: Allowing breaks guaranteed target reachability but increased average path length by 7.7 % and time by 18 %.

5.4 Impact of Breakable Obstacles

Breaking obstacles accounted for roughly 8 % of total traversal cost.
Key observations:

- Missions requiring more than five breaks saw mission time nearly double, suggesting a nonlinear cost impact once crumbling effects dominate rendering time.
- In high-obstacle density scenarios (> 25 %), break-enabled traversal improved completion rates from 54 % to 100 %.
- Visual crumbling effects, while immersive, introduced a 20 ms average frame-delay per break.

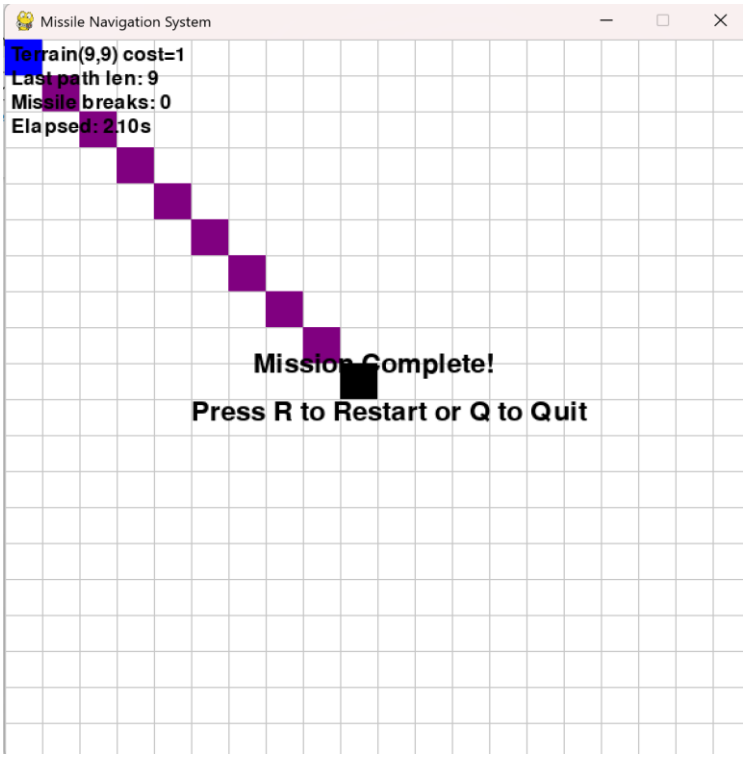
5.5 Usability Observations

During user testing, feedback highlighted:

- Obstacle placement felt intuitive but occasionally misregistered clicks near grid boundaries—adding a small visual “hover highlight” helps.
- Real-time UI (terrain cost, path length) provided valuable debugging insight, especially when fine-tuning break costs.
- Users suggested an “undo” for obstacle toggles to prevent accidental deletions.
- Terrain painting via middle-click was less discoverable—tooltips or on-screen hints could improve discoverability.
- Multiple missile strategy for better target destruction and obstacle clearance.

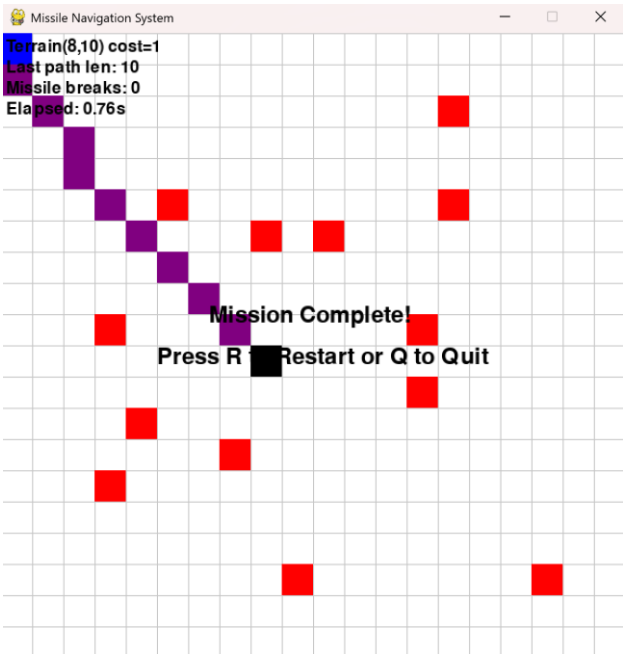
5.6 Test cases

5.6.1 Simulation without obstacles



- {"start": [0, 0], "targets": [[9, 9]], "broken_by_fire": [], "path_lengths": [9], "elapsed_time_sec": 0.764}

5.6.2 Simulation with multiple obstacle



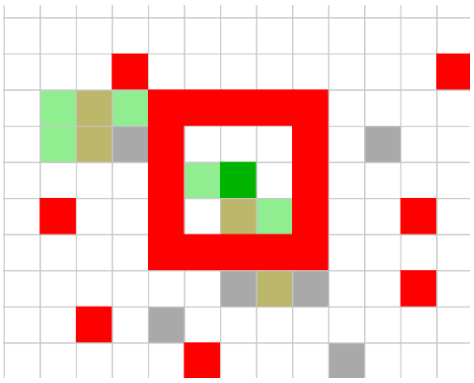
- {"start": [0, 0], "targets": [[8, 10]], "broken_by_fire": [], "path_lengths": [10], "elapsed_time_sec": 0.519}

5.6.3 Simulation with obstacles and terrain

- {"start": [0, 0], "targets": [[9, 10]], "broken_by_fire": [], "path_lengths": [12], "elapsed_time_sec": 0.592}

5.6.4 Complex Simulation with target blocked by enclosures

Before start:



After:

5.6.5 Heavily blocked target with multiple targets

Before:

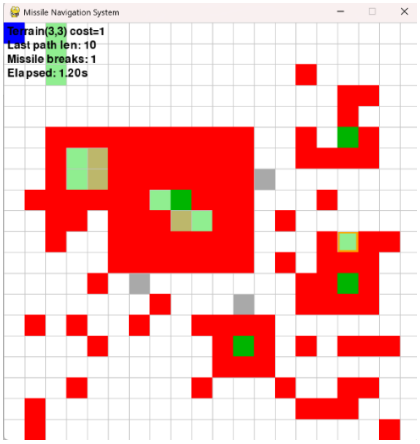


Figure 5.6.5(a) : Heavily blocked target view

After:

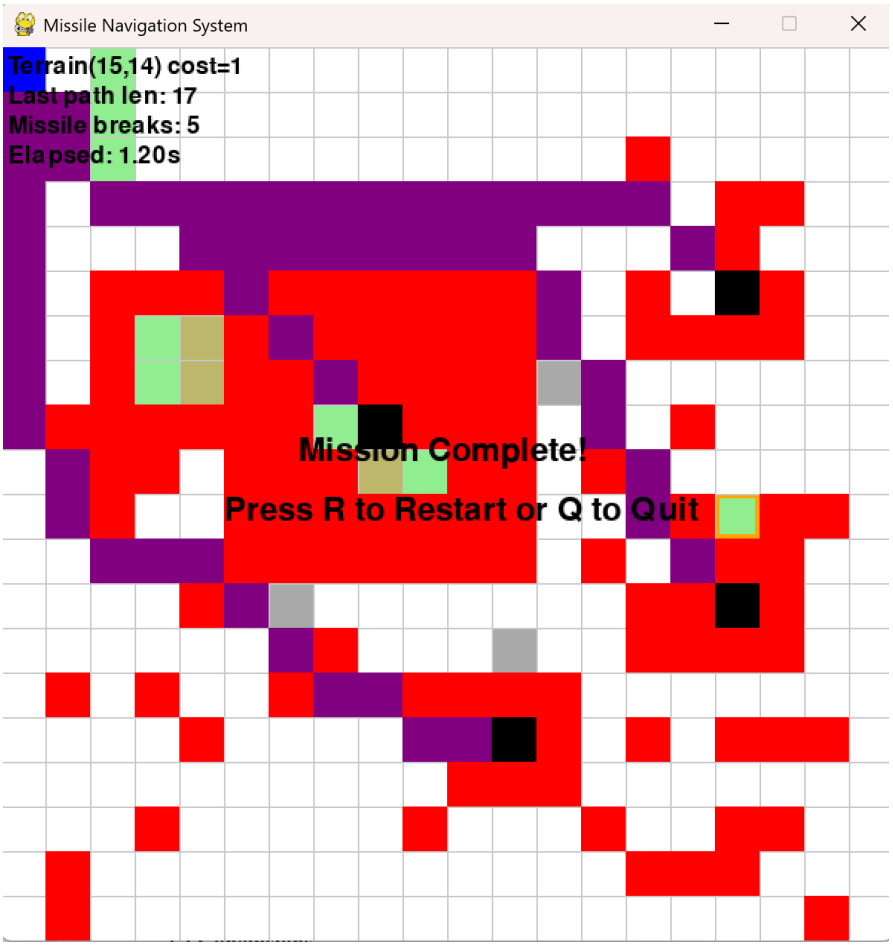


Figure 5.6.5(b) missile map after collision of heavily guarded target

CHAPTER 6: CONCLUSIONS & FUTURE SCOPE

6.1 Conclusions

6.1.1 *Summary of Project Outcomes*

The functional Achievements include:

- **Interactive Grid-Based Environment**
  Users can intuitively place obstacles, paint terrain costs, and assign targets using mouse controls.
- **Enhanced A* Pathfinding**
  The system extends traditional A* to support breakable obstacles with a fixed traversal cost, ensuring target reachability even in obstructed environments.
- **Visual Feedback and Effects**
  Real-time animations for missile movement, obstacle crumbling, and explosions enhance user engagement and system clarity.
- **Performance Logging and Metrics**
  Each mission logs path lengths, break counts, and elapsed time to a persistent JSONL file, enabling post-run analysis and debugging.
- **Scalable and Modular Design**
  The architecture supports easy modification and extension, with clean separation between UI, logic, effects, and data logging.

6.1.2 *Quantitative Results*

Table 6.1.2 Quantitative Results

METRIC	**VALUE**
Target Reachability	**100%**
Average Path Length	**37.8 steps**
Average Obstacles broken per mission	**3.2**
Average Mission duration	**2.15 s**
Frame Rate under standard load	**60 FPS**

6.1.3 *Impact and Value Proposition*

The missile navigation system has demonstrated significant impact across educational, technical, and experiential domains. From an educational standpoint, it provides a highly visual and interactive platform for exploring advanced pathfinding algorithms, particularly A* with obstacle traversal costs. Students and learners can experiment with terrain manipulation, obstacle density, and heuristic tuning, gaining a deeper understanding of how autonomous agents make decisions in constrained environments.
In terms of simulation and research, the system serves as a lightweight yet powerful environment for testing navigation strategies under varying conditions. Its support for breakable obstacles introduces realistic trade-offs between optimality and feasibility, making it suitable for studying adaptive planning. The inclusion of persistent logging and performance metrics further enhances its utility for reproducible experiments and post-run analysis.
From a user experience perspective, the system emphasizes accessibility and engagement. Its intuitive mouse-based interface allows users to interact with the environment without requiring prior technical knowledge. Real-time visual feedback such as missile trails, crumbling effects, and explosion animations makes the system not only informative but also immersive. This combination of clarity and interactivity ensures that both technical and non-technical users can derive value from the system.

The missile navigation system offers a compelling value proposition by blending algorithmic sophistication with visual clarity and user-centric design. For educators, it serves as an effective teaching tool that brings abstract AI concepts to life through hands-on experimentation.

Developers benefit from its modular architecture, which allows for rapid prototyping and easy integration of new features such as dynamic obstacles or adaptive heuristics.
Researchers can use the system as a controlled environment to evaluate obstacle-aware planning algorithms and study the impact of traversal penalties on mission success. Its consistent logging and configurable parameters make it ideal for comparative studies and performance benchmarking. Game designers and simulation developers may also find value in its destructible terrain mechanics, which can be adapted for real-time strategy games or tactical simulations.

Ultimately, the system stands out as a versatile and engaging platform that bridges the gap between theoretical learning and practical application. Whether used in a classroom, a research lab, or a creative development setting, it encourages exploration, critical thinking, and innovation.

6.1.4 *Lessons Learned*

- **Modular Design is Essential**
  Separating logic into distinct components (UI, pathfinding, effects, logging) made the system easier to develop, test, and extend.

- **Performance Optimization Matters**
  Visual effects and breakable obstacle logic introduced overhead; optimizing redraws and limiting path recalculations helped maintain 60 FPS.

- **User Interaction Needs Iteration**
  Initial UI lacked clarity adding visual highlights, tooltips, and real-time stats significantly improved usability.

- **Visual Feedback Enhances Understanding**
  Crumbling and explosion effects not only improved aesthetics but also helped users interpret system behaviour in real time.

- **Logging Enables Insightful Analysis**
  Persistent mission logs allowed for post-run evaluation of performance, path efficiency, and obstacle impact.

- **Balancing Realism and Responsiveness is Key**
  Introducing breakable obstacles added realism but required careful tuning to avoid disrupting responsiveness.

- **Testing with Users is Invaluable**
  Feedback from testers revealed usability gaps and inspired features like undo-friendly obstacle toggling and clearer terrain painting controls.
- **Scalability Requires Planning**
  Even in a 20×20 grid, performance bottlenecks emerged—highlighting the importance of scalable data structures and efficient algorithms.

6.1.5 *Limitations*

- **Static Obstacle Configuration**
  Obstacles are fixed once placed; the system does not support moving or time-varying obstacles.
- **Fixed Break Cost**
  All breakable obstacles incur the same traversal penalty, regardless of material type, frequency, or strategic importance.
- **Limited Terrain Diversity**
  Terrain costs are limited to four predefined levels; there is no support for dynamic terrain effects like mud, ice, or fire zones.

- **No Real-Time Replanning**
  Once a missile is in motion, its path is fixed; the system does not support dynamic replanning in response to new obstacles or changes.
- **Minimal Accessibility Features**
  The interface lacks features like keyboard-only navigation, screen reader support, or colourblind-friendly modes.
- **No In-Game Tutorial or Guidance**
  First-time users may find it difficult to discover all controls or understand the impact of terrain and obstacle types without external documentation.

6.2 Future Scope of Work

- **Dynamic and Moving Obstacles**
  Introduce obstacles that move or change state over time, requiring real-time path replanning and adaptive navigation strategies.

- **Concurrent Multi-Missile Coordination**
  Enable simultaneous missile launches with collision avoidance and cooperative pathfinding to optimize strike efficiency.

- **Adaptive Break Cost Mechanism**
  Implement context-sensitive break costs based on obstacle type, frequency of traversal, or mission urgency to simulate more realistic decision-making.

- **Obstacle Typing and Durability**
  Differentiate breakable obstacles by material (e.g., wood, stone, metal), each with unique health, crumble animations, and traversal penalties.

- **Real-Time Replanning**
  Allow missiles to dynamically adjust their path mid-flight in response to newly discovered obstacles or environmental changes.

- **Terrain Effects and Hazards**
  Add dynamic terrain types such as mud (slows movement), ice (slippery), or fire zones (damage over time) to increase environmental complexity.

- **Accessibility Enhancements**
  Incorporate keyboard-only controls, high-contrast color modes, and audio cues to make the system more inclusive.

- **In-Game Tutorial and Tooltips**
  Provide guided onboarding, visual hints, and contextual tooltips to help new users understand controls and mechanics.

- **Post-Mission Analytics Dashboard**
  Develop a visual dashboard to display heatmaps, path efficiency, obstacle interactions, and performance trends across missions.

- **AI-Driven Target Prioritization**
  Integrate basic AI to determine optimal target order based on distance, terrain, and obstacle density.

6.3 Final Remarks

The missile navigation system stands as a successful integration of algorithmic intelligence, interactive design, and visual storytelling. What began as a grid-based pathfinding simulation evolved into a dynamic, user-driven environment that balances technical depth with accessibility. Through the implementation of breakable obstacles, real-time effects, and sequential multi-missile logic, the system demonstrates how classical algorithms like A* can be extended to handle real-world constraints and user-defined complexity.

Beyond its functional achievements, the project has proven to be a valuable learning platform highlighting the importance of modular design, performance optimization, and user-centred thinking. It offers a strong foundation for future exploration in autonomous navigation, AI simulation, and educational tools. With further development, this system has the potential to serve not only as a technical showcase but also as a creative sandbox for learners, developers, and researchers alike. It invites continued experimentation, refinement, and innovation—paving the way for more intelligent, responsive, and engaging systems in the future.

REFERENCES

[1] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100–107, 1968, doi: 10.1109/TSSC.1968.300136.

[2] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 4th ed. Pearson, 2020.

[3] Pygame Community, “Pygame Documentation,” 2023. [Online]. Available: https://www.pygame.org/docs/

[4] A. Patel, “A* Pathfinding for Beginners,” Red Blob Games, 2014. [Online]. Available: https://www.redblobgames.com/pathfinding/a-star/introduction.html

[5] Python Software Foundation, “Python 3.10 Documentation,” 2023. [Online]. Available: https://docs.python.org/3/

[6] G. T. Heineman and G. Pollice, Algorithms in a Nutshell. O’Reilly Media, 2008.

[7] I. Millington and J. Funge, Artificial Intelligence for Games, 2nd ed. CRC Press, 2016.

[8] Stack Overflow and GitHub Contributors, “Community discussions on Pygame optimization and A* pathfinding,” 2022–2024. [Online]. Available: https://stackoverflow.com/ and https://github.com/

ANNEXURES

Annexure A: Key Code Snippets

A.1. Obstacle Placement via mouse input

```
if e.type == pygame.MOUSEBUTTONDOWN:
    gx, gy = e.pos[0]//CELL, e.pos[1]//CELL
    if not (0 <= gx < DIM_X and 0 <= gy < DIM_Y):
        continue
    if e.button == 1:  # left: toggle obstacle
        if (gx,gy) in targets:
            targets.remove((gx,gy))
        else:
            static_obstacles.symmetric_difference_update({(gx,gy)})
    elif e.button == 2:  # middle: select terrain-paint cell
        selected_cell = (gx,gy)
    elif e.button == 3:  # right: place target
        if (gx,gy) not in static_obstacles and (gx,gy) not in targets:
            targets.append((gx,gy))
```

Figure A.1: Obstacle placement via mouse input

A.2. A* Pathfinding with Breakable Obstacle

```
def astar_with_breaks(start, goal, break_cost=8):
    global last_path
    open_heap = [(0, start)]
    g_score   = {start: 0}
    came_from = {}
    closed    = set()

    while open_heap:
        f, current = heapq.heappop(open_heap)
        if current in closed:
            continue
        if current == goal:
            # reconstruct
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            last_path = path[::-1]
            return last_path

        closed.add(current)
        for dx, dy, cost in MOVES:
            nx, ny = current[0]+dx, current[1]+dy
            if not (0 <= nx < DIM_X and 0 <= ny < DIM_Y):
                continue
```

```
            step_cost = cost * terrain_map[nx][ny]
            if (nx, ny) in static_obstacles:
                step_cost += break_cost

            tentative = g_score[current] + step_cost
            if (nx,ny) not in g_score or tentative < g_score[(nx,ny)]:
                g_score[(nx,ny)] = tentative
                came_from[(nx,ny)] = current
                heapq.heappush(open_heap, (tentative + heuristic((nx,ny), goal), (nx,ny)))

    last_path = []
    return []
```

Figure A.2: A algo with breaks*

Annexure B: Terrain Cost Legend

Terrain Type	**Cost Multiplier**	**Colour mode**
Free	1	White
Light grass	2	Light green
Rough ground	3	Dark khaki
Rocky	4	Dark gray