Mario Trap Avoidance

A 3D Pathfinding & Adversarial AI Visualizer

Aditya Jha (ID: 12340090)
Ayush Khelwal (ID: 12340430)
Hirannya Mhaisbadwe (ID: 12340950)
Rounak Kumar Gupta (ID: 12341840)

November 16, 2025

Abstract

Mario Trap Avoidance is an interactive web application that combines a game-like interface with classic Artificial Intelligence (AI) techniques. The project visualizes pathfinding and decision-making in a 3D grid world, where a Mario-like agent must reach a goal while avoiding bombs and zombies, and exploiting boosters. The system uses `three.js` for 3D rendering, a logical grid to represent the environment, and a mix of search and adversarial techniques (A* and Minimax with heuristics) to drive the agent and enemies. The tool is designed as an educational visualizer to help students understand how search algorithms and evaluation functions work in dynamic, adversarial environments.

1 Introduction

Pathfinding and decision-making are core topics in Artificial Intelligence. They appear in robotics, games, navigation systems, and many other domains. However, students often encounter these ideas only as pseudocode or abstract graphs, which can be hard to connect to real-world behaviour.

The *Mario Trap Avoidance* project aims to bridge this gap by embedding AI ideas in a playful, interactive setting. The user observes a Mario-like character navigating a grid filled with:

- **Bombs** (static traps / obstacles),
- **Zombies** (moving adversaries),
- **Boosters** (positive collectibles that give lives and shields),
- a **Goal** cell that Mario must reach.

The environment is rendered in 3D using `three.js`, and the agent's behaviour is controlled by AI logic. By holding the spacebar, the user can switch Mario into an "Optimal AI" mode that uses a Minimax-based decision process guided by a heuristic evaluation. Zombies in turn use A* search to move towards Mario.

The main goal of the project is educational: to make search and adversarial reasoning *visible*, *intuitive*, and *interactive*.

2 Problem Description

2.1 Environment and State Space

The environment is a finite 10×10 grid G:

$$G = \{(x, y) \mid x, y \in \{0, 1, \ldots, 9\}\}.$$

Each cell may contain one of several entities:

- **Empty**: free space that can be traversed.
- **Start**: Mario's initial position at $(0, 0)$.
- **Goal**: target cell at $(9, 9)$.
- **Bomb** (trap): impassable obstacle.
- **Booster**: collectible power-up.
- **Zombie**: mobile enemy.

In the code, these are represented as symbolic constants:

$$\texttt{EMPTY} = 0,\ \texttt{BOMB} = 3,\ \texttt{BOOSTER} = 4,\ \texttt{ZOMBIE} = 5.$$

The logical game state at any time includes:

- Mario's position (x_M, y_M) and attributes: current lives and number of boosters held.
- Positions of all zombies $\{(x_{Z_i}, y_{Z_i})\}$.
- Positions of all remaining boosters.
- The static pattern of bombs.
- Whose turn it is (Mario turn vs. Zombies turn).

2.2 Actions and Dynamics

Mario and zombies move in discrete turns, sharing a common set of *movement primitives.* From a cell (x, y), an agent can move to one of the 8 neighbours:

$$(x + \Delta x, y + \Delta y),\ \Delta x, \Delta y \in \{-1, 0, 1\}$$

excluding the trivial $(0, 0)$ move. Diagonal moves are allowed, as long as the target cell is within the grid and not occupied by a bomb.

The dynamics can be summarized as:

- **Mario's turn:**
  - If the user is not pressing space, Mario picks a random valid move.
  - If the user holds the spacebar, Mario uses an "Optimal AI" mode based on Minimax search with a heuristic evaluation function.
  - After moving, collisions with zombies, boosters, or the goal are resolved.
- **Zombies' turn:**
  - Zombies move towards Mario using an A*-based move selection.
  - To keep difficulty reasonable, their turn is effectively slowed: they skip some turns based on a fixed rule (e.g. every 4th turn).
  - Conflicts (multiple zombies attempting to enter the same cell) are resolved by a conflict-handling routine; some zombies may stay put or choose alternate moves.

2.3 Win/Loss Conditions

Win condition: Mario reaches the goal cell $(9, 9)$ without running out of lives. The game displays a win message and stops.

Loss condition: Mario's life count becomes zero. This happens if he is caught by zombies without having boosters to protect him. In that case the game shows a loss message and ends.

3 Game Mechanics and Rules

3.1 Difficulty Levels

The user selects one of three difficulty levels from the dashboard:

- **Easy**: 1 zombie, 1 booster.
- **Medium**: 2 zombies, 2 boosters.
- **Hard**: 3 zombies, 3 boosters.

The number of bombs is controlled by a density parameter `BOMB_DENSITY` (e.g. 0.2 for a 20% chance that a non-start, non-goal cell becomes a bomb).

3.2 Lives and Boosters

Unlike a simple one-hit game, Mario has:

- **Lives** (`marioLives`): if this drops to zero, the game ends in a loss.
- **Boosters** (`marioBoosters`): these act like shields.

The interactions are:

- **Collecting a booster:**
  - Mario's booster count increases by one.
  - Mario also gains one life.
  - The booster model and light are removed from the scene and grid.
- **Being caught by a zombie:**
  - If Mario has at least one booster:
    - A booster is consumed.
    - The zombie is destroyed (removed from grid and scene).
    - Mario neither moves nor loses a life.
  - If Mario has no booster:
    - Mario loses one life.
    - If lives reach zero, the game terminates in defeat.
    - If lives remain, both Mario and the zombie are respawned: Mario returns to the start cell; the zombie is placed at a random safe cell far enough from the start.

3.3 Turn Order and Statistics

The game alternates strictly:

Mario Turn → Zombies Turn → Mario Turn → ·

A turn counter (`turnsTaken`) tracks how many Mario turns have occurred. The dashboard displays:

- Game status (Idle, Playing, Finished),
- Lives remaining,
- Boosters collected,
- Mario AI mode (Random vs. Optimal),
- Turns taken so far.

4 System Architecture and Implementation

4.1 Technology Stack

The implementation is fully web-based:

- **HTML**: Structure of the splash page and main game page.
- **Tailwind CSS**: Utility-first styling of the dashboard, legend, and statistics panel.
- **JavaScript (ES6 modules)**: Implements game logic, AI, and UI.
- **Three.js**: Handles 3D scene construction, lighting, camera, and animations.

4.2 3D Scene Setup

The code creates a `THREE.Scene` with:

- A perspective camera looking down on the grid.
- A hemisphere light and directional light to softly illuminate the scene and cast shadows.
- Orbit controls so the user can rotate and zoom the camera.

The grid is represented visually by a `gridGroup`: each logical cell is a cube (slightly transparent for empty cells) with an optional wireframe edge overlay. The start and goal cells use distinct materials with different colours to match the legend.

Zombies and Mario are small stylized 3D characters built from simple geometries (boxes, spheres, cylinders). Boosters are glowing spheres with attached point lights to stand out.

4.3 Logical Grid and Level Generation

Internally, the game uses a 10×10 array `gameGrid`. On starting or restarting, the logic:

1. Clears any previous 3D objects from `gridGroup` and resets all state variables.
2. Fills the grid with `EMPTY`.
3. Marks the start and goal cells.
4. Randomly adds bombs with probability `BOMB_DENSITY`, avoiding the start and goal.
5. Places zombies at random empty cells that are sufficiently far from the start.
6. Places boosters similarly at random empty cells.

The procedure ensures that bombs, zombies, and boosters do not overlap and that the start and goal are never blocked at initialization.

4.4 Movement Animation

To keep movement visually appealing, every move is animated with a short tween:

- The model's position interpolates from source to target over a fixed duration using `requestAnimationFrame`.
- The agent is rotated to face its direction of travel.
- For Mario, a small vertical sinusoid is added to the animation to create a "jumping" effect during motion.

This is achieved in a function `animateMove(model, fromPos, toPos, duration)`, which returns a Promise so that the turn logic can await the movement.

5 AI and Search Techniques

5.1 Neighbourhood and Valid Moves

The helper function `getNeighbors(x, y, allowDiagonals)` returns all adjacent cells (4 or 8 neighbours). A move $(x, y) \to (x', y')$ is valid for Mario if:

- (x', y') is inside the grid bounds, and
- the target cell is not a bomb.

Zombies use a similar notion of neighbours but also treat other zombies as temporary obstacles during their planning to reduce stacking.

5.2 A* Search for Pathfinding

Both zombies and some internal evaluation for Mario rely on A* search as a low-level path planner. Given a start position and a goal, A* maintains an open set and a closed set, computing the cost:

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

where $g(n)$ is the cost so far and $h(n)$ is a heuristic estimate to the goal.

In this project:

- Moving horizontally or vertically has cost 1.
- Moving diagonally has cost approximately 1.414 (approx. $\sqrt{2}$).
- The heuristic h is the Manhattan distance:

$$h(n) = |x_n - x_{\text{goal}}| + |y_n - y_{\text{goal}}|.$$

Because diagonal movement is allowed while using Manhattan distance, the heuristic can overestimate the true path cost in some cases; hence, the specific implementation is not strictly admissible and cannot guarantee an absolutely optimal path in the presence of diagonals. However, in practice it still produces natural-looking paths and remains efficient.

Zombies invoke A* with a grid where bombs, boosters, and other zombies are treated as blocked. Mario's heuristic evaluation also uses A* to approximate distance to the goal under certain constraints.

5.3 Minimax for Mario's "Optimal AI"

When the user holds the spacebar, Mario switches from random movement to an adversarial decision-making mode using Minimax search with depth limitation.

5.3.1 State and Turn Model

A simulated state includes:

- Mario's position,
- positions of all zombies,
- the grid layout (bombs, boosters),
- whose turn it is (maximizing vs. minimizing).

The Minimax recursion alternates between:

- **Maximizing player**: Mario. Tries to select moves that maximize the evaluation score.
- **Minimizing player**: zombies. A combined move is simulated by moving all zombies one step, roughly in Mario's direction using A*.

The search is bounded by a fixed depth `AI_SEARCH_DEPTH` (e.g. 8 plies), and uses α–β pruning to cut off branches that are provably worse than already explored alternatives.

5.3.2 Heuristic Evaluation Function

If a terminal situation is detected during search:

- If Mario reaches the goal, a very large positive value (e.g. $+10^6$) is returned.
- If a zombie occupies Mario's position, a large negative value (e.g. -10^6) is returned.

In non-terminal states, the heuristic combines three priorities:

1. **Distance to goal**: reward states that are closer (via an A* path) to the goal.
2. **Distance from zombies**: reward safety by increasing score when Mario is far from the nearest zombie.
3. **Distance to boosters**: modestly reward being closer to the nearest booster.

Conceptually, a score of the form

$$\text{score} = w_1 \cdot (100 - d_{\text{goal}}) + w_2 \cdot d_{\text{zombie}} + w_3 \cdot (C - d_{\text{booster}})$$

is computed, where $w_1 \gg w_2 \gg w_3$ are weight constants, and C is a cap for the booster distance contribution. Thus:

- Reaching the goal is always top priority.
- Staying away from zombies is the second priority.
- Heading towards boosters is encouraged, but only after goal and safety.

5.3.3 Move Selection

At the root of the search tree, all valid moves for Mario are enumerated. A short-circuit is used: if any move immediately reaches the goal, it is chosen without further search. Otherwise, each move is evaluated with Minimax, and the move with the best heuristic value is selected.

If no valid moves exist (e.g. Mario is trapped by bombs), a large negative score is returned, representing an effectively losing state.

6 Qualitative Behaviour and Observations

In experiments (by playing multiple rounds at different difficulty levels), we observed:

- On **easy** difficulty, Mario's optimal AI can usually reach the goal reliably, collecting boosters opportunistically and rarely being cornered by the single zombie.
- On **medium** difficulty, the game becomes more interesting: two zombies can coordinate to restrict Mario. The Minimax search adapts by favouring paths that keep escape routes open.
- On **hard** difficulty, three zombies substantially increase the challenge. Mario must carefully balance progress towards the goal with staying out of zombie "funnels", making the heuristic tuning more critical.
- Visually, zombies appear to "hunt" Mario as they continuously recompute A* paths. Their shared speed reduction (skipping some turns) keeps the game playable and gives Mario time to exploit boosters and safe corridors.

The behaviour matches expectations about heuristic search and adversarial planning: good evaluation functions produce smart-looking behaviour even with limited search depth.

7 Conclusion

The *Mario Trap Avoidance* project successfully integrates:

- a 3D interactive visualization using `three.js`,
- a logical grid-world model,
- pathfinding (A* search),
- and adversarial decision-making (Minimax with heuristics),

into a single educational tool.

It demonstrates how an AI agent can:

- navigate around static traps,
- survive and even exploit interactions with moving enemies,
- and make trade-offs between multiple objectives (reaching the goal, staying safe, collecting resources).

The game-like presentation helps students see abstract AI concepts in action, making it easier to reason about search, heuristics, and turn-based decision processes.

Overall, the project offers a solid foundation for experimenting with and teaching AI concepts in a visually engaging context.