# Project MazeMind: Comprehensive Technical Strategy and Architectural Blueprint for 5x5 Grid Optimization

## 1. Executive Summary and Strategic Alignment

The "MazeMind" challenge presents a seemingly deceptive computational problem: the implementation of a $5 \times 5$ grid maze solver. While the search space of twenty-five cells appears trivial to the human eye, the project serves as a sophisticated microcosm for broader computer science challenges, including graph theory, state management, algorithmic efficiency, and user experience design. For a five-member hackathon team, the primary constraint is not computational resources—modern processing power can traverse a twenty-five-node graph in microseconds—but rather development velocity, architectural coherence, and the "polish" that distinguishes a functional prototype from a winning product.

This report provides an exhaustive, expert-level analysis of the optimal technical path for the MazeMind challenge. It rigorously evaluates the trade-offs between manual inputs and AI-driven pathfinding, specifically contrasting Breadth-First Search (BFS), A* (A-Star), and Depth-First Search (DFS) within the specific constraints of small-grid topology. Furthermore, it details a React-based implementation strategy utilizing Vite, Framer Motion, and Tailwind CSS, supported by a sophisticated prompt engineering framework designed to leverage AI code editors like Cursor and GitHub Copilot to their maximum potential.

The analysis indicates that for a $5 \times 5$ unweighted grid, Breadth-First Search (BFS) is the superior algorithmic choice due to its mathematical guarantee of finding the shortest path without the computational overhead of heuristic calculation required by A*. However, the success of the project in a hackathon setting hinges on the "Game Feel" or "Juice"—interactive feedback loops, particle effects, and fluid animations—which necessitates a component-based architecture (DOM elements) rather than a rasterized one (Canvas).

## 2. Hackathon Team Topology and Operational

# Dynamics

The structural organization of the team is the single most significant predictor of hackathon success. The literature on software engineering in time-constrained environments suggests that role clarity minimizes merge conflicts, reduces cognitive overhead, and streamlines the integration pipeline.[1] For MazeMind, the team must function as a specialized micro-service architecture where human resources are allocated to distinct but interoperable domains.

## 2.1 The MazeMind Squad Configuration

The recommended topology distributes the five members into specialized roles that cover the full stack of application development, from theoretical algorithm design to pixel-perfect rendering and deployment.

| Role | Primary Responsibility | Technical Domain | Critical Deliverables |
|---|---|---|---|
| **1. The Architect (Backend/Algo)** | Maze Generation & Solving Logic | Python/TypeScript, Graph Theory | generateMaze(), solveBFS(), validatePath(), Unit Tests |
| **2. The Visualizer (Frontend Lead)** | Grid Rendering & State Management | React, Vite, CSS Grid | Grid Component, Path Animation Loop, Responsive Layout |
| **3. The UX/Juice Designer** | Interactions & Game Feel | Framer Motion, Particles.js, Tailwind | Micro-interactions, Screen Shake, Theme Toggling, Sound FX |
| **4. The AI/Prompt Engineer** | Acceleration & Boilerplate | Cursor, Copilot, LLMs | PRD.md, Context Rules, Complex Hook Generation, Refactoring |

| | | | |
|---|---|---|---|
| **5. The Product Integrator** | PM, QA, & Pitch Narrative | Markdown, Git Management | README.md, Deployment Pipeline, Edge Case Testing, Pitch Deck |

## 2.2 Deep Dive: Role-Specific Operational Workflows

### 2.2.1 The Architect (Algorithm Specialist)

This role is the "Domain Expert".[1] In the context of MazeMind, they must abstract the grid into a mathematical structure. They are responsible for implementing the "Brain" of the application. They should likely work in a separate algorithms/ directory or even a standalone repository initially to ensure logic purity. Their primary focus is correctness: ensuring that the maze generation creates a "perfect" maze (no loops, fully connected) and that the solver handles edge cases (e.g., start node surrounded by walls) gracefully.[3]

### 2.2.2 The Visualizer (Frontend Lead)

The Visualizer must mediate between the abstract logic provided by the Architect and the user interface. This role involves complex state management. They must decide how to represent a 2D array in React state without causing unnecessary re-renders. They act as the bridge between the logic and the "Juice," ensuring that the animations trigger at the correct moments during the algorithmic execution.[5]

### 2.2.3 The UX/Juice Designer

"Juice" refers to the non-functional feedback that makes a game feel satisfying—screen shake, particle effects, and elastic animations.[7] While the Visualizer builds the grid, the UX Designer builds the *experience* of the grid. They focus on the tactile response of the manual

mode (e.g., keypress latency) and the visual spectacle of the AI mode. They are responsible for the aesthetic coherence of the application.[1]

### 2.2.4 The AI/Prompt Engineer

This is a meta-role unique to modern development. This member does not just write code; they generate code. Using tools like Cursor and Copilot, they produce the boilerplate, unit tests, and documentation that would otherwise consume valuable time. They serve as a force multiplier, unblocking other members by generating complex React hooks or CSS configurations on demand.[9]

### 2.2.5 The Product Integrator

Often the "Get Stuck In" role [1], the Integrator ensures the disparate parts come together. They manage the GitHub repository, handle merge conflicts, and deploy the application (e.g., to Vercel or Netlify). Crucially, they craft the narrative for the final demo, ensuring the judges understand *why* the team chose BFS over A* or how the maze generation algorithm works.[11]

# 3. Theoretical Foundations of Grid Topology

To implement a "probable solution," one must first understand the mathematical properties of the problem space. A $5 \times 5$ grid is a specific type of graph with distinct properties that influence algorithmic choice.

## 3.1 The Grid as a Graph

The maze can be modeled as a Graph $G = (V, E)$.

- **Vertices ($V$):** The set of all cells in the grid. For a $5 \times 5$ grid, $|V| = 25$.
- **Edges ($E$):** The connections between adjacent cells. In a grid, each vertex $(r, c)$ has

a maximum degree of 4 (connecting to Up, Down, Left, Right).
- **Weighted vs. Unweighted:** In the standard MazeMind challenge, the cost to move from one cell to another is uniform (usually 1). Therefore, $G$ is an **unweighted graph**.[12]

### 3.2 Complexity Analysis in Small Search Spaces

Standard algorithmic complexity analysis (Big O notation) predicts performance as $N \to \infty$. However, for $N=25$, constant factors and implementation overhead dominate.

- **BFS Complexity:** $O(V + E)$. For $V=25$ and $E \approx 40$, the operations count is negligible.
- *A Complexity:** $O(E)$, but with a higher constant factor due to heuristic calculation (Manhattan distance) and priority queue maintenance.[13]

**Insight:** On a $5 \times 5$ grid, the difference in execution time between BFS, DFS, and A* is measured in microseconds. Therefore, the decision criteria shift from *raw performance* to *implementation simplicity* and *visualization clarity*. BFS provides a predictable "ripple" expansion pattern that is easier to visualize and explain than the directed tunneling of A*.[14]

## 4. Algorithmic Paradigms: Solving the Maze

The core query requests a comparison between Manual solution and AI Pathfinding (specifically BFS). To provide the "most probable" solution, we must evaluate the contenders.

### 4.1 Breadth-First Search (BFS)

BFS traverses the graph layer by layer. It starts at the root (Start Node) and explores all neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

- **Mechanism:** Uses a **Queue** (First-In-First-Out) data structure.
- **Optimality:** BFS is **mathematically guaranteed** to find the shortest path in an unweighted graph.[12]
- **Visualization:** Expands in a diamond shape (Manhattan geometry) or circle (Euclidean geometry). This visualizes the concept of "search radius" effectively.

- **Verdict: Optimal** for the MazeMind AI Solver. It is robust, simple to implement, and its optimality guarantee is critical for a "solver" tool.

## 4.2 A* (A-Star) Search

A* is a best-first search algorithm that uses a heuristic function $f(n) = g(n) + h(n)$ to prioritize node expansion.

- **$g(n)$:** The cost from the start node to node $n$.
- **$h(n)$:** The estimated cost from node $n$ to the goal.
- **Mechanism:** Uses a **Priority Queue**.
- **Suitability:** A* shines in large, open maps where exploring every node (like BFS) is too expensive. In a constrained $5 \times 5$ maze with many walls, the heuristic provides less advantage because the "direct" path is often blocked, forcing A* to behave similarly to BFS.[16]
- **Verdict: Overkill**. While impressive, the added complexity of a Priority Queue and heuristic function adds development risk with no perceptible performance gain for the user.

## 4.3 Depth-First Search (DFS)

DFS explores as far as possible along each branch before backtracking.

- **Mechanism:** Uses a **Stack** (Last-In-First-Out) or Recursion.
- **Optimality: Not Guaranteed**. DFS simply finds *a* path, not the shortest path. It might wind through the entire grid before finding a target that was adjacent to the start.[18]
- **Verdict: Unsuitable for Solving**, but **Essential for Generation** (see Section 5).

## 4.4 Dijkstra's Algorithm

Dijkstra's algorithm is essentially BFS with a Priority Queue, designed for weighted graphs.

- **Suitability:** Since the MazeMind grid is unweighted, Dijkstra behaves identically to BFS but with slightly higher overhead for the Priority Queue operations.[6]
- **Verdict:** Redundant. Use BFS instead.

### 4.5 Comparative Matrix: Solver Selection

| Algorithm | Guarantees Shortest Path? | Data Structure | Complexity Class | 5x5 Suitability | Visual Appeal |
|---|---|---|---|---|---|
| **BFS** | **Yes** | Queue | $O(V+E)$ | **High** | High (Ripple) |
| **A*** | **Yes** | Priority Queue | $O(E)$ | Medium | High (Direct) |
| **DFS** | No | Stack | $O(V+E)$ | Low | Low (Erratic) |
| **Dijkstra** | **Yes** | Priority Queue | $O(V+E \log V)$ | Medium | High (Ripple) |

**Conclusion for "Most Probable Solution":** The team should implement **BFS** for the AI solving mode. It balances theoretical correctness with ease of implementation and visualization.

# 5. Procedural Generation: The Art of the Maze

To challenge the "Manual" user, the maze must be generated algorithmically. Randomly placing walls often results in unsolvable grids or trivial paths. We require a "Perfect Maze" algorithm—one that creates a spanning tree with no loops and exactly one path between any two points.[3]

### 5.1 Recursive Backtracking (The "River" Generator)

This is the most common algorithm for maze generation and relies on DFS.

- **Process:**
  1. Start at a random cell.
  2. Mark as visited.
  3. Randomly select an unvisited neighbor.
  4. Remove the wall between the current cell and the neighbor.
  5. Recursively repeat for the neighbor.
  6. If no unvisited neighbors exist, backtrack.[22]
- **Characteristics:** Produces long, winding corridors ("rivers") with few dead ends. This makes the maze fun to navigate manually, as the player can get "flow".[21]
- **Visual Potential:** The generation process itself can be visualized, showing the algorithm "carving" the maze out of a block of walls.[23]

### 5.2 Randomized Prim's Algorithm

- **Process:** Builds the maze by adding walls to a list and selecting them randomly to carve passages.
- **Characteristics:** Produces mazes with many short dead ends and a "noisy" texture. These are often harder to solve but less aesthetically pleasing than Backtracking mazes.[3]

### 5.3 Wilson’s Algorithm

- **Process:** Uses Loop-Erased Random Walks.
- **Characteristics:** Generates a Uniform Spanning Tree (unbiased sample).
- **Drawback:** Can be slow to converge, though irrelevant for $5 \times 5$. Implementation complexity is significantly higher.[3]

**Recommendation:** Implement **Recursive Backtracking**. It is simpler to code (using the stack) and produces mazes that are visually coherent and enjoyable for the "Manual" user mode.[25]

## 6. Frontend Architecture and Rendering Engines

The choice of rendering technology defines the application's performance profile and, more importantly, the ease with which "Juice" (animations and interactions) can be added.

## 6.1 Rendering Core: HTML Divs vs. HTML5 Canvas

The research highlights a critical trade-off between the DOM (Document Object Model) and Canvas (Raster Graphics).[27]

### 6.1.1 HTML5 Canvas

- **Pros:** High performance for massive numbers of objects (e.g., $100 \times 100$ grids). Single DOM element reduces reflow/repaint overhead.
- **Cons:** "Black box" rendering. The browser doesn't know what's inside the canvas. Interaction (clicking a cell) requires manual hit detection math (calculating $x,y$ coordinates relative to the canvas). Accessibility is poor. Styling requires JavaScript commands (ctx.fillStyle = 'red') rather than CSS.[27]

### 6.1.2 HTML Divs (CSS Grid)

- **Pros:** Each cell is a DOM element ( ). Use standard CSS for styling (hover states, transitions). Interactions use native event listeners (onClick, onMouseEnter). Accessibility is built-in (screen readers can traverse the grid).
- **Cons:** Performance degrades with thousands of elements due to DOM weight.
- **Analysis for 5x5:** A $5 \times 5$ grid creates only 25 DOM elements. Even a $50 \times 50$ grid (2500 elements) is handled easily by modern React (Virtual DOM). The performance penalty of Divs is non-existent at this scale.

**Decision:** Use **HTML Divs** with **CSS Grid** layout. This allows the team to use **Tailwind CSS** for rapid styling and **Framer Motion** for complex layout animations, which are difficult to replicate in Canvas.[30]

## 6.2 Tech Stack Selection

- **Framework: React** (via **Vite**). React's component-based architecture maps perfectly to the Grid/Cell structure. Vite provides a fast development loop (HMR) essential for hackathons.[5]
- **Language: TypeScript**. Provides type safety for the algorithmic logic (interface Node { row: number; col: number;... }), preventing common runtime errors.[33]
- **Styling: Tailwind CSS**. Allows for atomic styling and rapid prototyping. "Utility-first" prevents context switching between CSS files and JSX.[33]
- **Animation: Framer Motion**. The industry standard for React animations. Essential for the "layout animations" when the grid changes or walls appear.[34]

### 6.3 State Management Architecture

The application state needs to track:

1. **Grid Data:** 2D array of Node objects.
2. **Algorithm State:** Is it running? Is it finished?
3. **User Mode:** Manual vs. AI.
4. **Theme/Juice:** Particle settings, colors.

**Pattern:** Use React **Context API** or a lightweight store like **Zustand**. For a hackathon, passing props down 2-3 levels is acceptable, but a useGrid hook encapsulating the logic is cleaner.[36]

Visualizing the Algorithm (The "Loop"):
Crucially, the algorithm execution (instant) must be decoupled from the visualization (delayed).

- **Incorrect:** Using setInterval to advance the algorithm one step at a time. This mixes logic with UI and can be buggy.
- **Correct:** Run the algorithm to completion in memory. It returns a list of *Visited Nodes* in order. Then, use a useEffect to iterate through this list, applying a CSS class to each node with a progressive delay (e.g., $i \times 20ms$).[37]

## 7. The Physics of "Game Feel" (Juice)

In a hackathon, functionality is the baseline; "Juice" is the differentiator. Juice refers to the

feedback loops that make the interface feel alive—screen shake, particles, and sound.[7]

## 7.1 UX Micro-Interactions

### 7.1.1 Screen Shake

When the user (Manual Mode) tries to move into a wall, the grid should reject the action physically.

- **Implementation:** Apply a CSS transform that translates the grid on the X-axis rapidly: 0px $\to$ -5px $\to$ 5px $\to$ 0px.
- **Library:** react-spring or Framer Motion variants can handle this with a "spring" physics configuration for a natural decay.[40]

### 7.1.2 Particle Effects

Upon solving the maze (either AI or Manual), the user should be rewarded.

- **Technique:** Use react-confetti for a full-screen celebration or tsparticles for localized effects (e.g., sparks flying from the finish node).
- **Integration:** Trigger the particle component conditionally when status === 'won'.[42]

### 7.1.3 Path Animation

The shortest path shouldn't just appear; it should *flow*.

- **Technique:** Use CSS transition-delay based on the index of the node in the path, or Framer Motion's staggerChildren property. This creates a snake-like visual of the yellow path tracing itself back to the start.[44]

#### 7.1.4 Sound Design (Optional but Recommended)

- **Feedback:** A subtle "click" sound on valid moves, a "thud" on invalid moves (walls), and a "chime" on victory.
- **Tool:** use-sound hook for React. This adds a sensory dimension often missed in web apps.[45]

# 8. Implementation Strategy: Manual vs. AI Modes

The prototype requires two distinct modes of interaction that share the same underlying data structure.

## 8.1 Manual Mode: Keyboard Navigation

This mode turns the visualizer into a game.

- **Input Handling:** Listen for keydown events.
- **Constraint:** The standard useEffect listener can lead to memory leaks if not cleaned up.
- **Solution:** Create a custom useKeyPress hook that binds listeners to window and filters for Arrow keys (Up, Down, Left, Right).
- **Logic:**
  TypeScript
  ```
  const handleKeyDown = (key) => {
   const nextPos = getNextPosition(currentPos, key);
   if (isValid(nextPos) &&!isWall(nextPos)) {
     movePlayer(nextPos);
   } else {
     triggerShake(); // The Juice
   }
  };
  ```

- **Accessibility:** Ensure the grid container has tabIndex={0} so it can receive focus without hijacking the entire window.[46]

### 8.2 AI Mode: The Solver Simulation

This mode demonstrates the power of the algorithm.

- **Input:** A "Visualize" button.
- **Logic:**
  1. Reset the visualization state (clear colors).
  2. Run BFS(grid, start, end).
  3. Receive visitedNodes array and shortestPath array.
  4. Trigger the animation loop for visitedNodes (Blue).
  5. On completion of the blue animation, trigger the animation loop for shortestPath (Yellow).
- **Comparison:** The report recommends implementing **only BFS** for the solution to keep the UI clean, but perhaps including **DFS** to *show* why it is inferior (demonstrating the wandering path vs. the direct BFS path).[14]

## 9. AI-Accelerated Development & Prompt Engineering

The "AI/Prompt Engineer" role is critical for delivering this polish within the timeframe. This section provides the "Perfect Prompts" and keywords requested.

### 9.1 Keyword Strategy for AI Editors

When interacting with Cursor (Claude 3.5) or Copilot, use high-precision technical vocabulary to force the model into "Expert Mode":

- **Algorithms:** "Recursive Backtracking," "Breadth-First Search (BFS)," "Adjacency Matrix," "Queue Data Structure," "Path Reconstruction."
- **React:** "Custom Hook," "Memoization (useMemo/useCallback)," "Render Props," "Virtual DOM," "Race Conditions."
- **Styling/Juice:** "Framer Motion Layout Animations," "Spring Physics," "Tailwind Grid," "Shadcn UI," "Z-Index Stacking."

## 9.2 The Prompt Library

### Prompt Set A: Project Initialization (Architect/AI Engineer)

**Target:** Cursor Composer / Copilot Workspace

> Context: You are a Senior React Architect specializing in algorithm visualizers.
> Task: Scaffold a Vite + React + TypeScript project for "MazeMind".
> Requirements:
> 1. **File Structure:** Follow a feature-based architecture: src/features/Grid, src/features/Controls, src/algorithms.
> 2. **Styling:** Configure Tailwind CSS.
> 3. **Component:** Create a Grid component that renders a dynamic $5 \times 5$ grid of Node components.
> 4. **State:** Implement a useGrid custom hook to manage the 2D array state. Each cell must track: row, col, isWall, isStart, isEnd, isVisited, isPath.
> 5. Rendering: Use divs with CSS Grid, NOT Canvas.
>
> Output: Provide the file structure tree and the code for useGrid.ts and Grid.tsx.

### Prompt Set B: Algorithmic Logic (Backend/Algo)

**Target:** Cursor Chat (referencing @Grid.tsx)

> Context: We have a grid state managed by useGrid. @Grid.tsx
> Task: Implement the Breadth-First Search (BFS) algorithm in a pure TypeScript file.
> Constraints:
> 1. **Input:** A grid (2D array of nodes), startNode, finishNode.
> 2. **Output:** An object { visitedNodesInOrder: Node, nodesInShortestPathOrder: Node }.
> 3. **Logic:** Use a Queue. Explore neighbors in Up, Down, Left, Right order. Handle edge cases where the target is unreachable.
> 4. **Pathfinding:** Implement a helper to backtrack from the finish node to the start node using a previousNode property to reconstruct the shortest path.
> 5. **Performance:** Ensure the algorithm terminates immediately when the finish

node is reached.

**Prompt Set C: The "Juice" & Animation (UX Designer)**

**Target:** Cursor Chat (referencing @Node.tsx)

Context: We are using Framer Motion for animations. @Node.tsx
Task: Add layout animations to the Node component.
Requirements:

1. **Wall Creation:** When a node becomes a wall, it should scale up from 0 to 1 with a "spring" bounce effect.
2. **Visited Animation:** When a node is visited by the algorithm, it should pulse blue and change shape from a circle to a square.
3. Shortest Path: The shortest path nodes should animate in sequence (staggered) turning yellow.
   Output: The updated Node.tsx component code with motion.div integration.

**Prompt Set D: Manual Navigation Hook (Frontend/UX)**

**Target:** Copilot Chat

Task: Create a useMazeNavigation custom hook.
Logic:

1. Attach a keydown listener to the window.
2. Listen for Arrow Keys.
3. Validate the next move against the grid boundaries (0-4) and isWall status.
4. If the move is valid, update the playerNode state.
5. If the move is INVALID (hitting a wall), return a triggerShake boolean to act as a signal for screen shake.
6. Debounce the input by 50ms to prevent erratic movement.

# 10. Testing, Deployment, and Future Scalability

### 10.1 Testing Strategy

In a hackathon, formal TDD (Test Driven Development) is often too slow. However, "Strategic Testing" is vital.

- **Unit Tests:** The Architect should write Vitest tests for the solveBFS function to ensure it actually finds the shortest path on known grid configurations.[48]
- **Visual Regression:** The Integrator should manually verify that the "Reset" button actually clears all state (including animations) to prevent "ghost" paths on subsequent runs.

### 10.2 Deployment

The Integrator handles the pipeline.

- **Platform:** Vercel or Netlify (seamless integration with Vite/GitHub).
- **Configuration:** Ensure dist/ is the build directory.
- **CI/CD:** Set up a GitHub Action to run the linter on every push to main to prevent broken demos.[33]

### 10.3 Future Scalability: Beyond 5x5

While the prompt specifies $5 \times 5$, the architecture (BFS + React Grid) scales linearly.

- **Performance:** The current Div-based approach works well up to $\approx 50 \times 50$. Beyond that, the team would need to refactor the Grid component to use a **Virtual List** (react-window) or switch to **Canvas**.
- **Algorithms:** The modular algorithms/ folder allows easy addition of Dijkstra, A*, or Bidirectional Swarm algorithms without changing the UI code.

## 11. Conclusion

The "MazeMind" challenge, while constrained in dimensions, offers a boundless canvas for technical expression. By choosing **BFS** for its algorithmic purity, **Recursive Backtracking** for its generative aesthetics, and **React + Framer Motion** for its rendering capabilities, the team positions itself to deliver a prototype that is not only "probable" but optimal.

The distinguishing factor will be the execution of the **"Juice"**—the screen shakes, particles, and fluid animations that transform a simple graph traversal into an engaging interactive experience. By leveraging the defined roles and the **AI Prompt Library**, the team can compress days of development into hours, ensuring a high-fidelity submission that stands out for its polish and architectural soundness. The 5x5 grid is the stage; the code is the performance.

### Works cited


1. 5 Roles Every Hackathon Team Needs - EQ, accessed November 26, 2025, https://entrepreneurquarterly.com/5-roles-every-hackathon-team-needs/
2. The Role of Hackathons in Advancing Web Development Skills - HackathonParty, accessed November 26, 2025, https://www.hackathonparty.com/blog/the-role-of-hackathons-in-advancing-web-development-skills
3. Maze generation algorithm - Wikipedia, accessed November 26, 2025, https://en.wikipedia.org/wiki/Maze_generation_algorithm
4. Maze Algorithm That generates the most difficult mazes? - Stack Overflow, accessed November 26, 2025, https://stackoverflow.com/questions/14692818/maze-algorithm-that-generates-the-most-difficult-mazes
5. How to Create a Path-Finding Algorithm Visualizer with React - freeCodeCamp, accessed November 26, 2025, https://www.freecodecamp.org/news/path-finding-algorithm-visualizer-tutorial/
6. Path finding visualizer using React — from creating to building and deploying | by Prudhvi Gnv | Medium, accessed November 26, 2025, https://medium.com/@prudhvi.gnv/path-finding-visualizer-using-react-from-creating-to-building-and-deploying-bd1e2bc64696
7. GAME FEEL FOR ACTION GAMES - COOL UNITY TUTORIAL - YouTube, accessed November 26, 2025, https://www.youtube.com/watch?v=UsGuN69g2NI
8. Juice It Good: Adding Camera Shake To Your Game | by Antonio Delgado | Medium, accessed November 26, 2025, https://gt3000.medium.com/juice-it-adding-camera-shake-to-your-game-e63e1a16f0a6
9. Cursor AI: A Guide With 10 Practical Examples - DataCamp, accessed November 26, 2025, https://www.datacamp.com/tutorial/cursor-ai-code-editor
10. What I learned using CursorAI every day as an Engineer | Codeaholicguy, accessed November 26, 2025, https://codeaholicguy.com/2025/04/12/what-i-learned-using-cursorai-every-day-as-an-engineer/

11. The complete guide to organizing a successful hackathon - HackerEarth, accessed November 26, 2025, https://www.hackerearth.com/community-hackathons/resources/e-books/guide-to-organize-hackathon/
12. Finding shortest path through a maze - DEV Community, accessed November 26, 2025, https://dev.to/prathiksha_dcsbs_9a50f2d/finding-shortest-path-through-a-maze-mg7
13. AI: Fastest algorithm to find if path exists? - Stack Overflow, accessed November 26, 2025, https://stackoverflow.com/questions/15508370/ai-fastest-algorithm-to-find-if-path-exists
14. BFS Graph Visualization | Tom Sawyer Software, accessed November 26, 2025, https://blog.tomsawyer.com/bfs-graph-visualization
15. Returning the shortest path using breadth first search - DEV Community, accessed November 26, 2025, https://dev.to/a_b_102931/returning-the-shortest-path-using-breadth-first-search-3b52
16. A* Pathfinding Visualization Tutorial - Python A* Path Finding Tutorial - YouTube, accessed November 26, 2025, https://www.youtube.com/watch?v=JtiK0DOeI4A
17. 505. The Maze II - In-Depth Explanation, accessed November 26, 2025, https://algo.monster/liteproblems/505
18. Shortest path: DFS, BFS or both? - Stack Overflow, accessed November 26, 2025, https://stackoverflow.com/questions/14784753/shortest-path-dfs-bfs-or-both
19. Deep path traversal algorithms - Memgraph, accessed November 26, 2025, https://memgraph.com/docs/advanced-algorithms/deep-path-traversal
20. pathfinding-visualizer · GitHub Topics, accessed November 26, 2025, https://github.com/topics/pathfinding-visualizer
21. Building a Maze Generation Framework - Dimi Chakarov, accessed November 26, 2025, https://dchakarov.com/blog/maze-algorithms/
22. Buckblog: Maze Generation: Recursive Backtracking - Jamis Buck, accessed November 26, 2025, https://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtracking
23. Maze Generation — Recursive Backtracking | by Aryan Abed-Esfahani | Medium, accessed November 26, 2025, https://aryanab.medium.com/maze-generation-recursive-backtracking-5981bc5cc766
24. JavaScript Maze Generation (Depth First Search) Tutorial - YouTube, accessed November 26, 2025, https://www.youtube.com/watch?v=nHjqkLV_Tp0
25. Maze Generation - Backtracking Algorithm - GitHub Pages, accessed November 26, 2025, https://professor-l.github.io/mazes/
26. Tutorial - Random Maze Generation Algorithm in Javascript - dstromberg, accessed November 26, 2025, https://dstromberg.com/2013/07/tutorial-random-maze-generation-algorithm-in-

javascript/
27. HTML5 Canvas vs. SVG vs. div - javascript - Stack Overflow, accessed November 26, 2025, https://stackoverflow.com/questions/5882716/html5-canvas-vs-svg-vs-div
28. Efficiency of and s - Stack Overflow, accessed November 26, 2025, https://stackoverflow.com/questions/8009128/efficiency-of-canvas-and-divs
29. Improving HTML5 Canvas performance | Articles - web.dev, accessed November 26, 2025, https://web.dev/articles/canvas-performance
30. Div vs Canvas opinions? : r/gamedev - Reddit, accessed November 26, 2025, https://www.reddit.com/r/gamedev/comments/reclr2/div_vs_canvas_opinions/
31. Game of Life: Canvas, HTML table or SVG for making the board/grid?, accessed November 26, 2025, https://forum.freecodecamp.org/t/game-of-life-canvas-html-table-or-svg-for-making-the-board-grid/67371
32. Creating a Good Folder Structure For Your Vite App - ThatSoftwareDude.com, accessed November 26, 2025, https://www.thatsoftwaredude.com/content/14110/creating-a-good-folder-structure-for-your-vite-app
33. Was any of you actually able to build a more complex app with Composer/Tab without much programming knowledge? : r/cursor - Reddit, accessed November 26, 2025, https://www.reddit.com/r/cursor/comments/1gjzibg/was_any_of_you_actually_able_to_build_a_more/
34. Layout Animation — React FLIP & Shared Element - Motion, accessed November 26, 2025, https://motion.dev/docs/react-layout-animations
35. Everything about Framer Motion layout animations - The Blog of Maxime Heckel, accessed November 26, 2025, https://blog.maximeheckel.com/posts/framer-motion-layout-animations/
36. Prompt Coding with Cursor | AI-powered programming for existing codebases, accessed November 26, 2025, https://daveinside.com/blog/prompt-coding-with-cursor/
37. Pathfinding Visualizer Tutorial (software engineering project) - YouTube, accessed November 26, 2025, https://www.youtube.com/watch?v=msttfIHHkak
38. How to create a delay in React.js - Stack Overflow, accessed November 26, 2025, https://stackoverflow.com/questions/70547116/how-to-create-a-delay-in-react-js
39. Using React to visualize algorithms - what am I doing wrong? - Stack Overflow, accessed November 26, 2025, https://stackoverflow.com/questions/57999359/using-react-to-visualize-algorithms-what-am-i-doing-wrong
40. You can instantly add a lot of satisfaction to your game with JUICE! Screenshake, particles & VFX, SFX, haptics, etc. All thrown into to my 2.5D Portal game. What do you think? : r/Unity3D - Reddit, accessed November 26, 2025, https://www.reddit.com/r/Unity3D/comments/raxjux/you_can_instantly_add_a_lot_

of_satisfaction_to/
41. Juice it or loose it - How to make a game feel better by only altering its surface. : r/gamedev, accessed November 26, 2025, https://www.reddit.com/r/gamedev/comments/u75rh/juice_it_or_loose_it_how_to_make_a_game_feel/
42. Particles - shadcn.io, accessed November 26, 2025, https://www.shadcn.io/components/interactive/particles
43. Particles - React Bits, accessed November 26, 2025, https://reactbits.dev/backgrounds/particles
44. Create Shortest Path Visualizer in React using BFS algorithm - YouTube, accessed November 26, 2025, https://www.youtube.com/watch?v=D5H2HZV7j1Y
45. Good examples of "game juice"/ game feel? : r/gamedesign - Reddit, accessed November 26, 2025, https://www.reddit.com/r/gamedesign/comments/198fctp/good_examples_of_game_juice_game_feel/
46. React Grid: Keyboard Interaction - AG Grid, accessed November 26, 2025, https://www.ag-grid.com/react-data-grid/keyboard-navigation/
47. How to make arrow key navigation on responsive card grid like google drive in React?, accessed November 26, 2025, https://stackoverflow.com/questions/69250435/how-to-make-arrow-key-navigation-on-responsive-card-grid-like-google-drive-in-re
48. Prompt Engineering with GitHub Copilot - YouTube, accessed November 26, 2025, https://www.youtube.com/watch?v=yduxrQkqlxg