Computer Graphics

(UCS505)

Project on

<u>3D Maze</u>

Submitted By

Rajveer Singh 102203195

Japneet Singh 102203205

3C15

B.E. Third Year – COE

Submitted To:

Dr. Archana Kumari

Computer Science and Engineering Department

Thapar Institute of Engineering and Technology

TABLE OF CONTENTS

INTRODUCTION

Mazes have fascinated humans for centuries, serving as both entertainment and tools for developing spatial reasoning skills. In the digital age, 3D Maze games combine this timeless appeal with modern computer graphics to create immersive problem-solving experiences.

Our project implements a fully interactive 3D maze environment using OpenGL and C++. The application generates unique mazes algorithmically, ensuring no two playthroughs are exactly alike. Players navigate these labyrinths from a first-person perspective, with realistic movement controls and collision detection preventing passage through walls.

Key technical achievements include:

- Procedural maze generation using randomized algorithms
- First-person camera implementation with smooth movement
- Dynamic lighting that enhances depth perception
- Real-time collision detection system
- Interactive minimap for navigation assistance

Built using C++ with GLUT for window management, this application serves as both an engaging game and a practical demonstration of 3D graphics programming fundamentals. The clean, modular code structure makes it an excellent learning resource for those interested in game development or computer graphics.

Through this implementation, we showcase how relatively simple graphical techniques can combine to create an engaging and replayable 3D experience. The project could be extended with additional features like texture mapping, enemy AI, or multiplayer functionality for enhanced gameplay possibilities.

COMPUTER GRAPHICS CONCEPTS USED

1. 3D Modeling and Object Representation

- The entire maze structure—including walls, floors, and boundaries—is created using basic 3D primitives like cubes and quads. Each primitive is formed by specifying vertices in 3D space and connecting them into faces (usually triangles or quadrilaterals).

- These shapes are manually defined or generated using loops to build the grid-like pattern of the maze, where each block is a visible element in the virtual world.

- The player's position and path are determined relative to these shapes, and their placement defines the navigable and blocked regions in the maze.

2. Transformations (Translation, Rotation, Scaling)

- Translation is used to place walls, floor tiles, and other objects at their correct positions within the grid. This allows repeating shapes (like cubes) to be placed across the scene dynamically.

- Rotation is used to change the camera angle or object orientation—for example, if the maze supports turning corners or rotating objects.

- Scaling may be used to adjust the size of certain objects (like making walls taller or scaling floor tiles to fit a specific grid size). These transformations work together to build a dynamic and interactive 3D environment

3. Camera and Viewing (gluLookAt, Perspective Projection)

- The camera simulates the player’s eyes in a 3D environment, allowing navigation through the maze. The function gluLookAt() is typically used to define the camera’s position, the point it looks at, and the up direction, thus creating a realistic viewing direction.

- Perspective projection is used instead of orthographic projection to provide depth to the scene. It mimics how the human eye perceives objects: those farther away appear smaller, enhancing realism.

- As the player moves, the camera dynamically updates based on their coordinates and viewing direction, providing a smooth first-person or third-person experience.

4. User Interaction and Collision Detection

- Keyboard inputs are used to allow the player to move forward, backward, and turn left or right within the maze. These inputs are mapped to transformation functions that update the player’s position and direction.

- Collision detection is implemented to prevent the player from walking through walls or moving outside the maze boundaries. This involves checking the player's current or intended position against the maze’s structure to determine if movement is allowed.

- These mechanics ensure fair gameplay and help simulate realistic physics-like constraints, where solid objects cannot be passed through.

- Together, input handling and collision logic allow for smooth navigation, accurate control, and engaging interaction between the player and the 3D environment.

USER DEFINED FUNCTIONS

Geometric Functions:

1. Collision Detection (checkCollision())

- **Purpose:** Ensures the player cannot walk through walls.
- **Mechanism:**
    - **Grid Quantization:** Converts the player's continuous (floating-point) position into discrete grid coordinates.
    - **AABB Check:** Treats each wall as an axis-aligned bounding box (AABB) and checks if the player's quantized position overlaps with a wall cell (maze[gridX][gridZ] == 1).
- **Why It Matters:**
    - Provides realistic movement constraints.
    - Optimizes performance by avoiding complex geometric intersection tests.

---

2. Camera Control (First-Person View)

- **Purpose:** Simulates a natural first-person perspective.
- **Mechanism:**
    - **Translation:** Moves the camera to the player's position using glTranslatef(-x, -y, -z).
    - **Rotation:**
        - **Yaw (Horizontal):** Rotates the view around the Y-axis (playerAngle) using glRotatef().
        - **Pitch (Vertical):** Tilts the view up/down (playerLookUpDown), though limited in this implementation.
- **Why It Matters:**
    - Mimics real-world head movement.
    - Uses Euler angles for intuitive control.

---

3. Maze Rendering (drawMaze())

- **Purpose:** Visualizes walls, floors, and ceilings.
- **Mechanism:**
    - **Walls:** Rendered as unit cubes (glutSolidCube(1.0)) at grid positions where maze[i][j] == 1.
    - **Floors/Ceilings:** Thin planes (scaled cubes) at y=0 (floor) and y=1 (ceiling).
- **Why It Matters:**
    - Demonstrates polygonal modeling with primitive shapes.

    - Uses hierarchical transformations (glPushMatrix/glPopMatrix) for efficient rendering.

---

4. Minimap (drawMinimap())

- **Purpose:** Provides a navigational overview.
- **Mechanism:**
    - **Perspective Projection:** Switches to 2D mode (glOrtho) to draw the maze as a top-down grid.
    - **Coordinate Mapping:** Scales maze coordinates to minimap pixels.
    - Player Icon: Renders the player as a point with a direction vector (based on playerAngle).
- **Why It Matters:**
    - Combines 2D/3D rendering techniques.
    - Aids spatial awareness without breaking immersion.

---

5. Lighting (init())

- **Purpose:** Enhances depth perception and realism.
- **Mechanism:**
    - **Light Source:** A directional light (GL_LIGHT0) positioned overhead.
    - **Material Properties:** Surfaces reflect light based on glColor3f() values.
- **Why It Matters:**
    - Uses Phong lighting (ambient + diffuse) for basic shading.
    - Avoids complex shaders by leveraging OpenGL's fixed-function pipeline.

---

6. Procedural Maze Generation (generateMaze())

- **Purpose:** Creates randomized mazes algorithmically.
- **Mechanism:**
    - **Randomized DFS:**
        1. Starts at (1, 1).
        2. Recursively carves paths by removing walls between cells.
        3. Backtracks when no unvisited neighbors remain.
    - **3D Conversion**: Extrudes the 2D grid into 3D walls of uniform height.
- **Why It Matters:**
    - Applies graph theory (DFS) for maze topology.
    - Balances randomness and solvability.

---

7. View Frustum Culling (Implicit)

- **Purpose:** Optimizes rendering by skipping off-screen objects.
- **Mechanism:**
  - Only renders walls near the player's grid cell.
- **Why It Matters:**
  - Reduces unnecessary draw calls.
  - Implicitly uses the camera's view frustum.

---

8. Player Movement Vectors

- **Purpose:** Translates keyboard input into smooth motion**.**
- **Mechanism:**
  - **Forward/Backward:** Movement along the camera's look-at vector (sin/cos(playerAngle)).
  - **Strafe:** Not implemented but would use perpendicular vectors.
- **Why It Matters:**
  - Demonstrates vector math in game mechanics.

---

9. Start/End Markers

- **Purpose:** Visual cues for maze entry and exit.
- **Mechanism:**
  - Renders colored quads at fixed positions ((1.5, 0.1, 1.5) and (MAZE_SIZE-0.5, 0.1, MAZE_SIZE-0.5)).
- **Why It Matters:**
  - Uses world-space coordinates for consistent placement.

---

10. UI Text (drawText(), drawTimer())

- **Purpose:** Displays game information (time, instructions).
- **Mechanism:**
  - Switches to 2D orthographic mode.
  - Renders bitmap fonts (glutBitmapCharacter).
- **Why It Matters:**
  - Combines 2D text with a 3D scene.

---

11. Boundary Checks

- **Purpose:** Prevents player from leaving the maze.
- **Mechanism:**
  - Clamps player coordinates to [1, MAZE_SIZE].
- **Why It Matters:**
  - Simple AABB constraint for gameplay boundaries.

---

12. Lighting Falloff (Pseudo-Fog)

- **Purpose:** Simulates depth-based fading.
- **Mechanism:**
  - Dark background + limited light range creates a "fog-like" effect.
- **Why It Matters:**
  - Avoids explicit fog calculations while enhancing depth.

---

13. Maze Grid Topology

- **Purpose:** Underlies procedural generation.
- **Mechanism:**
  - 2D array (maze[i][j]) defines walkable (0) vs. solid (1) cells.
- **Why It Matters:**
  - Bridges discrete math (grids) and 3D rendering.

---

14. Player Hitbox

- **Purpose:** Simplifies collision checks.
- **Mechanism:**
  - Treats player as a single point or small cylinder.
- **Why It Matters:**
  - Avoids complex mesh collisions.

SOURCE CODE

3d_maze.cpp

```
#include <GLUT/glut.h>
#include <iostream>
#include <OpenGL/gl.h>
#include <OpenGL/glu.h>
#include <vector>
#include <random>
#include <string>
#include <cmath>
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif
#include <ctime>

// Window dimensions
const int WINDOW_WIDTH = 800;
const int WINDOW_HEIGHT = 600;

// Global variables
int MAZE_SIZE;
int k; // Difficulty level (1=easy, 2=medium, 3=hard)

// Function to get user input for difficulty
void getDifficultyInput() {
    std::cout << "Select difficulty level:" << std::endl;
    std::cout << "1. Easy (10x10 maze)" << std::endl;
    std::cout << "2. Medium (15x15 maze)" << std::endl;
    std::cout << "3. Hard (20x20 maze)" << std::endl;
    std::cout << "Enter your choice (1-3): ";

    while (true) {
        std::cin >> k;
        if (k >= 1 && k <= 3) {
            break;
        }
        std::cout << "Invalid input. Please enter 1, 2, or 3: ";
    }

    // Set maze size based on difficulty
    if (k == 1) {
        MAZE_SIZE = 10;
    } else if (k == 2) {
        MAZE_SIZE = 15;
    } else {
        MAZE_SIZE = 20;
    }
}

std::vector<std::vector<int>> maze;

// Player settings
float playerX = 1.5f;
float playerY = 0.5f;
float playerZ = 1.5f;
float playerAngle = 0.0f;
```

```
float playerLookUpDown = 0.0f;

// Movement settings
float cameraSpeed = 0.07f;
float rotationSpeed = 3.0f;

// Game settings
time_t gameStartTime;
bool gameFinished = false;
bool showMinimap = true;
bool showCongratsMessage = false;
int finalTime = 0;

// Key states - Updated to use special keys for arrows
bool keyStates[256] = { false };
bool specialKeyStates[256] = { false }; // For special keys like arrow keys

// Function prototypes
void init();
void display();
void reshape(int width, int height);
void keyboard(unsigned char key, int x, int y);
void keyboardUp(unsigned char key, int x, int y);
void specialKeyboard(int key, int x, int y); // Added for arrow keys
void specialKeyboardUp(int key, int x, int y); // Added for arrow keys
void timer(int value);
void generateMaze();
void drawMaze();
void drawMinimap();
void drawTimer();
void drawText(float x, float y, const char* text);
void drawInstructions();
void drawCongratsMessage();
bool checkCollision(float x, float y, float z);
void processMovement();

int main(int argc, char** argv)
{
    getDifficultyInput();
    // Initialize GLUT
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(WINDOW_WIDTH, WINDOW_HEIGHT);
    glutCreateWindow("3D Maze - GLUT (Arrow Keys Only)");

    // Register callbacks
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutKeyboardUpFunc(keyboardUp);
    glutSpecialFunc(specialKeyboard); // Added for arrow keys
    glutSpecialUpFunc(specialKeyboardUp); // Added for arrow keys
    glutTimerFunc(16, timer, 0); // ~60 FPS

    // Initialize OpenGL
    init();

    // Generate maze
    generateMaze();

    // Start game timer
    gameStartTime = time(NULL);
```

```
    // Enter main loop
    glutMainLoop();

    return 0;
}

void init()
{
    // Set background color
    glClearColor(0.1f, 0.1f, 0.1f, 1.0f);

    // Enable depth testing
    glEnable(GL_DEPTH_TEST);

    // Enable lighting
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    // Set up light
    GLfloat lightPosition[] = { 0.0f, 10.0f, 0.0f, 1.0f };
    GLfloat lightAmbient[] = { 0.3f, 0.3f, 0.3f, 1.0f };
    GLfloat lightDiffuse[] = { 0.7f, 0.7f, 0.7f, 1.0f };
    glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
    glLightfv(GL_LIGHT0, GL_AMBIENT, lightAmbient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, lightDiffuse);

    // Enable color material
    glEnable(GL_COLOR_MATERIAL);
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

    // Show cursor
    glutSetCursor(GLUT_CURSOR_LEFT_ARROW);
}

void display()
{
    // Clear the color and depth buffers
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Set up the camera
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // Apply camera rotation
    glRotatef(playerLookUpDown, 1.0f, 0.0f, 0.0f);
    glRotatef(playerAngle, 0.0f, 1.0f, 0.0f);

    // Apply camera translation
    glTranslatef(-playerX, -playerY, -playerZ);

    // Draw the maze
    drawMaze();

    // Draw HUD elements
    if (showMinimap) {
        drawMinimap();
    }

    drawTimer();
    drawInstructions();

    // Display congratulations message if finished
    if (showCongratsMessage) {
```

```
        drawCongratsMessage();
    }

    // Swap buffers
    glutSwapBuffers();
}

void reshape(int width, int height)
{
    // Set viewport
    glViewport(0, 0, width, height);

    // Set perspective projection
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0f, (float)width / (float)height, 0.1f, 100.0f);

    // Reset modelview matrix
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void keyboard(unsigned char key, int x, int y)
{
    keyStates[key] = true;

    // Exit on escape
    if (key == 27) {
        exit(0);
    }

    // Toggle minimap with 'm'
    if (key == 'm' || key == 'M') {
        showMinimap = !showMinimap;
    }

    // Dismiss congratulations message with space
    if (key == ' ' && showCongratsMessage) {
        showCongratsMessage = false;
    }
}

void keyboardUp(unsigned char key, int x, int y)
{
    keyStates[key] = false;
}

// Added for arrow keys
void specialKeyboard(int key, int x, int y)
{
    specialKeyStates[key] = true;
}

// Added for arrow keys
void specialKeyboardUp(int key, int x, int y)
{
    specialKeyStates[key] = false;
}

void timer(int value)
{
    // Process player movement if game not finished
    if (!showCongratsMessage) {
```

```
        processMovement();
    }

    // Check for game completion
    if (!gameFinished && playerX > MAZE_SIZE - 1.5f && playerZ > MAZE_SIZE - 1.5f) {
        gameFinished = true;
        showCongratsMessage = true;
        finalTime = time(NULL) - gameStartTime;
        std::cout << "Maze completed! Time: " << finalTime << " seconds" << std::endl;
    }

    // Redraw the scene
    glutPostRedisplay();

    // Register the next timer callback
    glutTimerFunc(16, timer, 0);
}

void processMovement()
{
    // Calculate movement vectors based on player orientation
    float dx = sin(playerAngle * M_PI / 180.0f);
    float dz = -cos(playerAngle * M_PI / 180.0f);

    float newX = playerX;
    float newZ = playerZ;

    // Process movement keys (Arrow keys instead of WASD)
    if (specialKeyStates[GLUT_KEY_UP]) {
        newX += dx * cameraSpeed;
        newZ += dz * cameraSpeed;
    }
    if (specialKeyStates[GLUT_KEY_DOWN]) {
        newX -= dx * cameraSpeed;
        newZ -= dz * cameraSpeed;
    }

    // Left and Right arrow keys for rotation
    if (specialKeyStates[GLUT_KEY_LEFT]) {
        playerAngle -= rotationSpeed;
        if (playerAngle < 0.0f) playerAngle += 360.0f;
    }
    if (specialKeyStates[GLUT_KEY_RIGHT]) {
        playerAngle += rotationSpeed;
        if (playerAngle >= 360.0f) playerAngle -= 360.0f;
    }

    // Check for collisions before updating position
    if (!checkCollision(newX, playerY, newZ)) {
        playerX = newX;
        playerZ = newZ;
    }
}

void generateMaze()
{
    // Initialize maze with all walls
    maze.resize(MAZE_SIZE + 2, std::vector<int>(MAZE_SIZE + 2, 1));

    // Create a path through the maze using randomized DFS with increased complexity
    std::vector<std::pair<int, int>> stack;
    std::random_device rd;
    std::mt19937 rng(rd());
```

```
// Start at (1,1) (accounting for boundary walls)
maze[1][1] = 0;
stack.push_back({ 1, 1 });

// Define direction vectors
const int dx[] = { 0, 1, 0, -1 };
const int dy[] = { -1, 0, 1, 0 };

while (!stack.empty()) {
    int x = stack.back().first;
    int y = stack.back().second;

    // Find unvisited neighbors
    std::vector<int> neighbors;
    for (int i = 0; i < 4; i++) {
        int nx = x + 2 * dx[i];
        int ny = y + 2 * dy[i];

        if (nx >= 1 && nx <= MAZE_SIZE && ny >= 1 && ny <= MAZE_SIZE && maze[nx][ny] == 1) {
            neighbors.push_back(i);
        }
    }

    // If no unvisited neighbors, backtrack
    if (neighbors.empty()) {
        stack.pop_back();
    }
    else {
        // Choose a random neighbor
        std::shuffle(neighbors.begin(), neighbors.end(), rng);
        int dir = neighbors[0];

        // Carve a path
        int nx = x + 2 * dx[dir];
        int ny = y + 2 * dy[dir];
        maze[x + dx[dir]][y + dy[dir]] = 0; // Remove wall between cells
        maze[nx][ny] = 0; // Mark new cell as visited

        stack.push_back({ nx, ny });
    }
}

// Set entrance and exit
maze[1][1] = 0;
maze[MAZE_SIZE][MAZE_SIZE] = 0;

// Add some additional random openings to increase complexity (30% chance of removing a wall)
for (int i = 2; i < MAZE_SIZE; i++) {
    for (int j = 2; j < MAZE_SIZE; j++) {
        if (maze[i][j] == 1 && (rng() % 100) < 30) {
            // Check if removing this wall would create a valid path
            int pathCount = 0;
            for (int k = 0; k < 4; k++) {
                int nx = i + dx[k];
                int ny = j + dy[k];
                if (nx >= 1 && nx <= MAZE_SIZE && ny >= 1 && ny <= MAZE_SIZE && maze[nx][ny] == 0) {
                    pathCount++;
                }
            }

            // Only remove if it connects at least 2 existing paths
            if (pathCount >= 2) {
```

```
                maze[i][j] = 0;
            }
        }
    }
  }
}

void drawMaze()
{
    // Draw each cell in the maze
    for (int i = 0; i < MAZE_SIZE + 2; i++) {
        for (int j = 0; j < MAZE_SIZE + 2; j++) {
            if (maze[i][j] == 1) { // Wall
                // All walls are the same color now (including boundary walls)
                glColor3f(0.5f, 0.5f, 0.7f); // Blue-ish for all walls

                // Draw wall cube
                glPushMatrix();
                glTranslatef(i, 0.5f, j);
                glutSolidCube(1.0f);
                glPopMatrix();
            }
            else { // Floor and ceiling
                // Draw floor
                glColor3f(0.3f, 0.3f, 0.3f);
                glPushMatrix();
                glTranslatef(i, 0.0f, j);
                glScalef(1.0f, 0.01f, 1.0f);
                glutSolidCube(1.0f);
                glPopMatrix();

                // Draw ceiling
                glColor3f(0.2f, 0.2f, 0.2f);
                glPushMatrix();
                glTranslatef(i, 1.0f, j);
                glScalef(1.0f, 0.01f, 1.0f);
                glutSolidCube(1.0f);
                glPopMatrix();
            }
        }
    }

    // Draw special markers for start and goal
    // Start marker
    glColor3f(0.0f, 1.0f, 0.0f);
    glPushMatrix();
    glTranslatef(1.5f, 0.1f, 1.5f);
    glScalef(0.3f, 0.1f, 0.3f);
    glutSolidCube(1.0f);
    glPopMatrix();

    // Goal marker
    glColor3f(1.0f, 0.0f, 0.0f);
    glPushMatrix();
    glTranslatef(MAZE_SIZE - 0.5f, 0.1f, MAZE_SIZE - 0.5f);
    glScalef(0.3f, 0.1f, 0.3f);
    glutSolidCube(1.0f);
    glPopMatrix();
}

void drawMinimap()
{
    // Disable lighting and depth testing for 2D elements
```

```
glDisable(GL_LIGHTING);
glDisable(GL_DEPTH_TEST);

// Save current matrices
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
gluOrtho2D(0, WINDOW_WIDTH, 0, WINDOW_HEIGHT);

glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();

// Calculate minimap position and size - IMPROVED MINIMAP
int minimapSize = 200; // Bigger minimap
int minimapX = WINDOW_WIDTH - minimapSize - 10;
int minimapY = 10;

// Calculate cell size (smaller for larger maze)
float cellSizeFloat = (float)minimapSize / (MAZE_SIZE + 2);

// Draw minimap background with border
// Background
glColor4f(0.0f, 0.0f, 0.0f, 0.8f); // More opaque background
glBegin(GL_QUADS);
glVertex2f(minimapX - 2, minimapY - 2);
glVertex2f(minimapX + minimapSize + 2, minimapY - 2);
glVertex2f(minimapX + minimapSize + 2, minimapY + minimapSize + 2);
glVertex2f(minimapX - 2, minimapY + minimapSize + 2);
glEnd();

// Border
glColor3f(1.0f, 1.0f, 1.0f);
glBegin(GL_LINE_LOOP);
glVertex2f(minimapX - 2, minimapY - 2);
glVertex2f(minimapX + minimapSize + 2, minimapY - 2);
glVertex2f(minimapX + minimapSize + 2, minimapY + minimapSize + 2);
glVertex2f(minimapX - 2, minimapY + minimapSize + 2);
glEnd();

// Draw maze cells
for (int i = 0; i < MAZE_SIZE + 2; i++) {
    for (int j = 0; j < MAZE_SIZE + 2; j++) {
        float x = minimapX + i * cellSizeFloat;
        float y = minimapY + j * cellSizeFloat;

        if (maze[i][j] == 1) {
            // All walls are the same color
            glColor3f(0.6f, 0.6f, 0.8f); // Consistent with 3D view
        }
        else {
            // Draw path
            if ((i == 1 && j == 1) || (i == MAZE_SIZE && j == MAZE_SIZE)) {
                if (i == 1 && j == 1) {
                    glColor3f(0.0f, 0.8f, 0.0f); // Start (green)
                }
                else {
                    glColor3f(0.8f, 0.0f, 0.0f); // End (red)
                }
            }
            else {
                glColor3f(0.2f, 0.2f, 0.2f); // Regular path (dark gray)
            }
```

```
        }

        glBegin(GL_QUADS);
        glVertex2f(x, y);
        glVertex2f(x + cellSizeFloat, y);
        glVertex2f(x + cellSizeFloat, y + cellSizeFloat);
        glVertex2f(x, y + cellSizeFloat);
        glEnd();
    }
}

// Draw player position with better visibility
// First draw a black outline
glColor3f(0.0f, 0.0f, 0.0f);
glPointSize(8.0f);
glBegin(GL_POINTS);
glVertex2f(minimapX + playerX * cellSizeFloat, minimapY + playerZ * cellSizeFloat);
glEnd();

// Then draw the yellow player marker
glColor3f(1.0f, 1.0f, 0.0f); // Yellow for player
glPointSize(6.0f);
glBegin(GL_POINTS);
glVertex2f(minimapX + playerX * cellSizeFloat, minimapY + playerZ * cellSizeFloat);
glEnd();

// Draw player direction with better visibility
float dx = sin(playerAngle * M_PI / 180.0f) * 8.0f; // Longer direction indicator
float dz = -cos(playerAngle * M_PI / 180.0f) * 8.0f;

// Black outline for direction line
glColor3f(0.0f, 0.0f, 0.0f);
glLineWidth(3.0f);
glBegin(GL_LINES);
glVertex2f(minimapX + playerX * cellSizeFloat, minimapY + playerZ * cellSizeFloat);
glVertex2f(minimapX + playerX * cellSizeFloat + dx, minimapY + playerZ * cellSizeFloat + dz);
glEnd();

// Yellow direction line
glColor3f(1.0f, 1.0f, 0.0f);
glLineWidth(1.5f);
glBegin(GL_LINES);
glVertex2f(minimapX + playerX * cellSizeFloat, minimapY + playerZ * cellSizeFloat);
glVertex2f(minimapX + playerX * cellSizeFloat + dx, minimapY + playerZ * cellSizeFloat + dz);
glEnd();

// Draw minimap title
glColor3f(1.0f, 1.0f, 1.0f);
glRasterPos2f(minimapX, minimapY + minimapSize + 15);
const char* minimapTitle = "MAZE MAP";
for (int i = 0; minimapTitle[i] != '\0'; i++) {
    glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, minimapTitle[i]);
}

// Restore matrices and states
glMatrixMode(GL_PROJECTION);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
glPopMatrix();

// Re-enable lighting and depth testing
glEnable(GL_DEPTH_TEST);
glEnable(GL_LIGHTING);
```

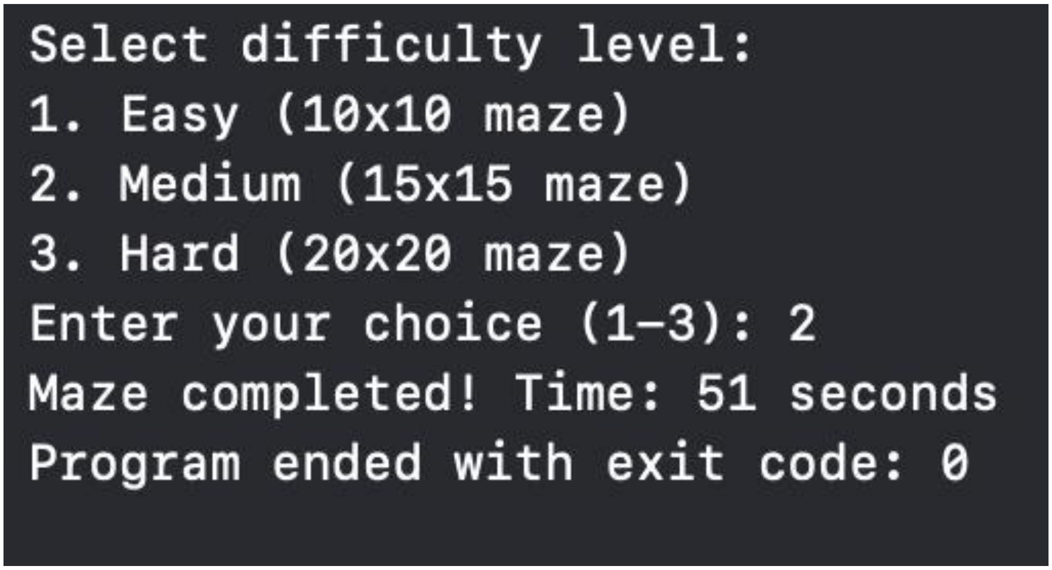
```
}

void drawTimer()
{
    // Calculate elapsed time
    int elapsedTime = gameFinished ? finalTime : (int)(time(NULL) - gameStartTime);

    // Convert to minutes:seconds format
    int minutes = elapsedTime / 60;
    int seconds = elapsedTime % 60;
    char timeString[32];
    snprintf(timeString, sizeof(timeString), "Time: %02d:%02d", minutes, seconds);

    // Save current matrices
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();

    // Set up 2D orthographic projection for UI elements
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, WINDOW_WIDTH, 0, WINDOW_HEIGHT, -1, 1);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // Disable lighting for 2D elements
    glDisable(GL_LIGHTING);
    glDisable(GL_DEPTH_TEST);

    // Draw timer text
    glColor3f(1.0f, 1.0f, 1.0f);
    glRasterPos2f(10, WINDOW_HEIGHT - 20);
    for (int i = 0; timeString[i] != '\0'; i++) {
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, timeString[i]);
    }

    // Re-enable lighting and depth testing
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_LIGHTING);

    // Restore matrices
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();
}

void drawInstructions()
{
    // Save current matrices
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();

    // Set up 2D orthographic projection for UI elements
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, WINDOW_WIDTH, 0, WINDOW_HEIGHT, -1, 1);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
```

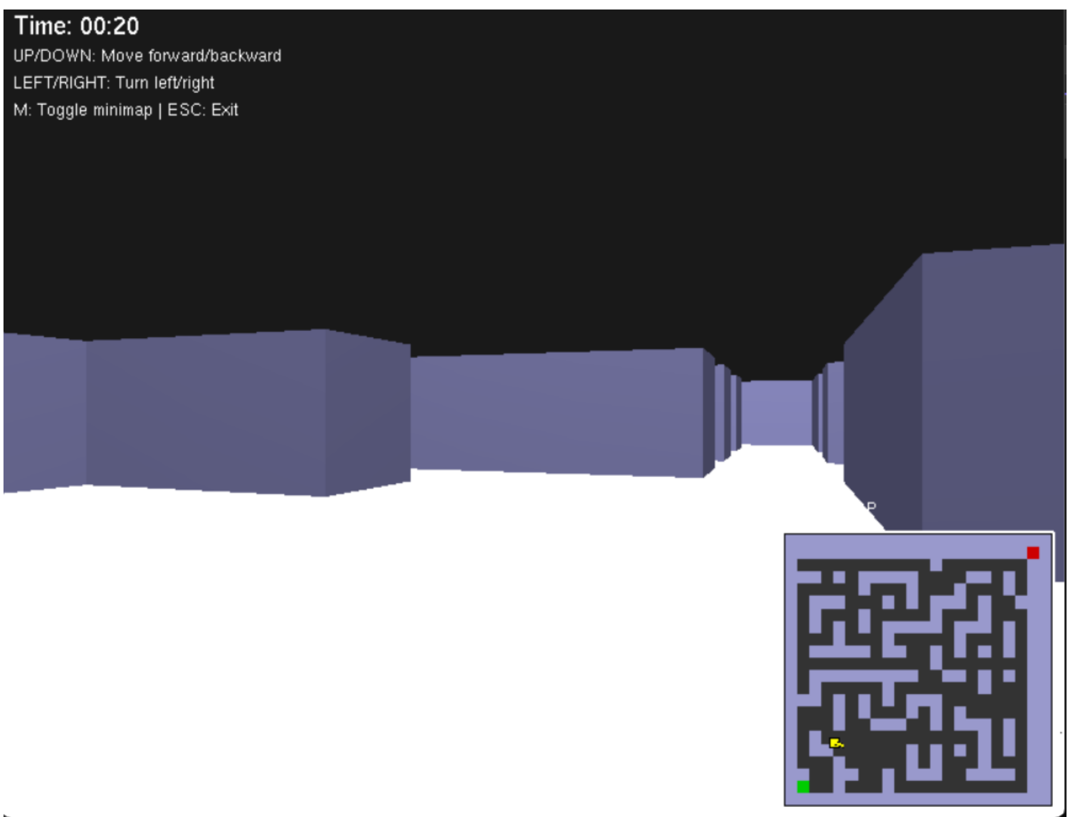
```
    // Disable lighting for 2D elements
    glDisable(GL_LIGHTING);
    glDisable(GL_DEPTH_TEST);

    // Draw instruction text
    glColor3f(1.0f, 1.0f, 1.0f);

    // Line 1 - Changed from W/S to UP/DOWN
    glRasterPos2f(10, WINDOW_HEIGHT - 40);
    const char* line1 = "UP/DOWN: Move forward/backward";
    for (int i = 0; line1[i] != '\0'; i++) {
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, line1[i]);
    }

    // Line 2 - Changed from A/D to LEFT/RIGHT
    glRasterPos2f(10, WINDOW_HEIGHT - 60);
    const char* line2 = "LEFT/RIGHT: Turn left/right";
    for (int i = 0; line2[i] != '\0'; i++) {
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, line2[i]);
    }

    // Line 3
    glRasterPos2f(10, WINDOW_HEIGHT - 80);
    const char* line3 = "M: Toggle minimap | ESC: Exit";
    for (int i = 0; line3[i] != '\0'; i++) {
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, line3[i]);
    }

    // Re-enable lighting and depth testing
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_LIGHTING);

    // Restore matrices
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();
}

void drawCongratsMessage()
{
    // Save current matrices and attributes
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();

    // Set up orthographic projection for 2D overlay
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, WINDOW_WIDTH, 0, WINDOW_HEIGHT, -1, 1);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // Disable lighting and depth testing
    glDisable(GL_LIGHTING);
    glDisable(GL_DEPTH_TEST);

    // Draw semi-transparent background
    glColor4f(0.0f, 0.0f, 0.0f, 0.7f);
    glBegin(GL_QUADS);
    glVertex2f(0, 0);
    glVertex2f(WINDOW_WIDTH, 0);
```

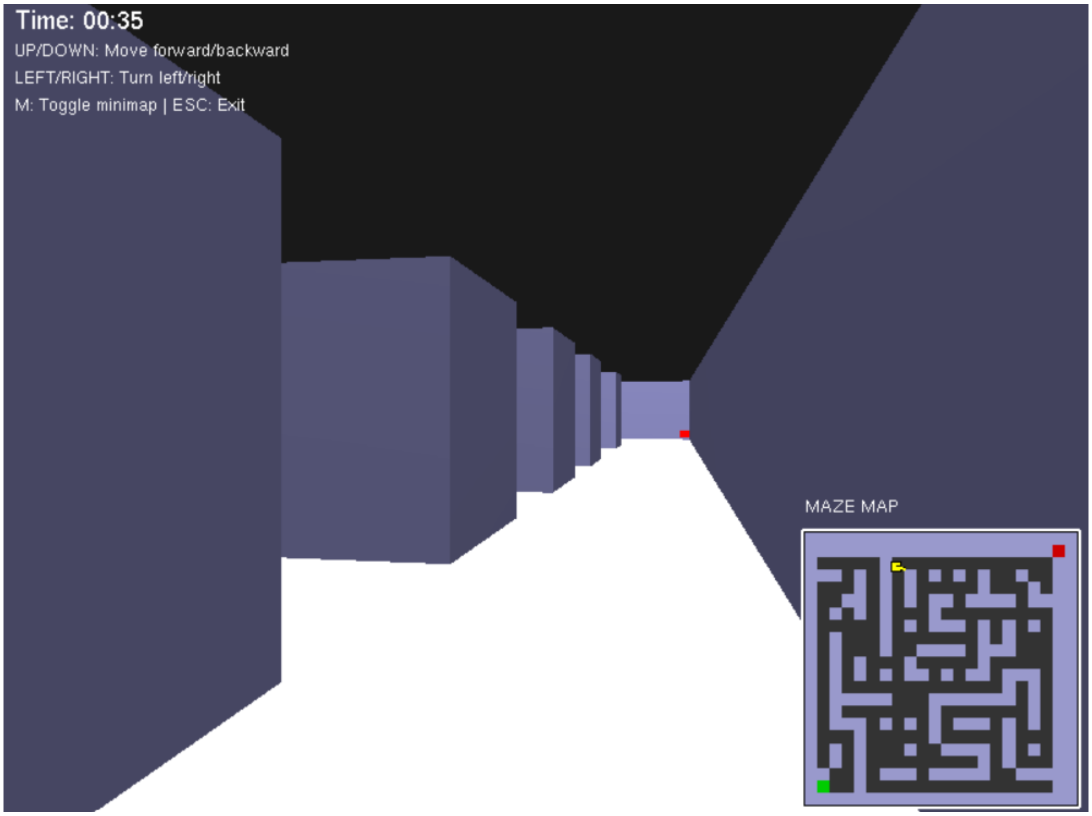
```
glVertex2f(WINDOW_WIDTH, WINDOW_HEIGHT);
glVertex2f(0, WINDOW_HEIGHT);
glEnd();

// Convert completion time to string
int minutes = finalTime / 60;
int seconds = finalTime % 60;
char timeString[100];
snprintf(timeString, sizeof(timeString), "Time: %02d:%02d", minutes, seconds);

// Draw congratulations text
glColor3f(1.0f, 1.0f, 0.0f); // Yellow

// Title
const char* congratsTitle = "CONGRATULATIONS!";
int titleWidth = 0;
for (int i = 0; congratsTitle[i] != '\0'; i++) {
   titleWidth += glutBitmapWidth(GLUT_BITMAP_TIMES_ROMAN_24, congratsTitle[i]);
}
glRasterPos2f((WINDOW_WIDTH - titleWidth) / 2, WINDOW_HEIGHT / 2 + 30);
for (int i = 0; congratsTitle[i] != '\0'; i++) {
   glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, congratsTitle[i]);
}

// Message
const char* congratsMessage = "You successfully completed the maze!";
int messageWidth = 0;
for (int i = 0; congratsMessage[i] != '\0'; i++) {
   messageWidth += glutBitmapWidth(GLUT_BITMAP_HELVETICA_18, congratsMessage[i]);
}
glRasterPos2f((WINDOW_WIDTH - messageWidth) / 2, WINDOW_HEIGHT / 2);
for (int i = 0; congratsMessage[i] != '\0'; i++) {
   glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, congratsMessage[i]);
}

// Time
int timeWidth = 0;
for (int i = 0; timeString[i] != '\0'; i++) {
   timeWidth += glutBitmapWidth(GLUT_BITMAP_HELVETICA_18, timeString[i]);
}
glRasterPos2f((WINDOW_WIDTH - timeWidth) / 2, WINDOW_HEIGHT / 2 - 30);
for (int i = 0; timeString[i] != '\0'; i++) {
   glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, timeString[i]);
}

// Continue message
const char* continueMessage = "Press SPACE to continue";
int continueWidth = 0;
for (int i = 0; continueMessage[i] != '\0'; i++) {
   continueWidth += glutBitmapWidth(GLUT_BITMAP_HELVETICA_12, continueMessage[i]);
}
glRasterPos2f((WINDOW_WIDTH - continueWidth) / 2, WINDOW_HEIGHT / 2 - 70);
for (int i = 0; continueMessage[i] != '\0'; i++) {
   glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, continueMessage[i]);
}

// Re-enable disabled states
glEnable(GL_DEPTH_TEST);
glEnable(GL_LIGHTING);

// Restore matrices
glMatrixMode(GL_PROJECTION);
glPopMatrix();
```

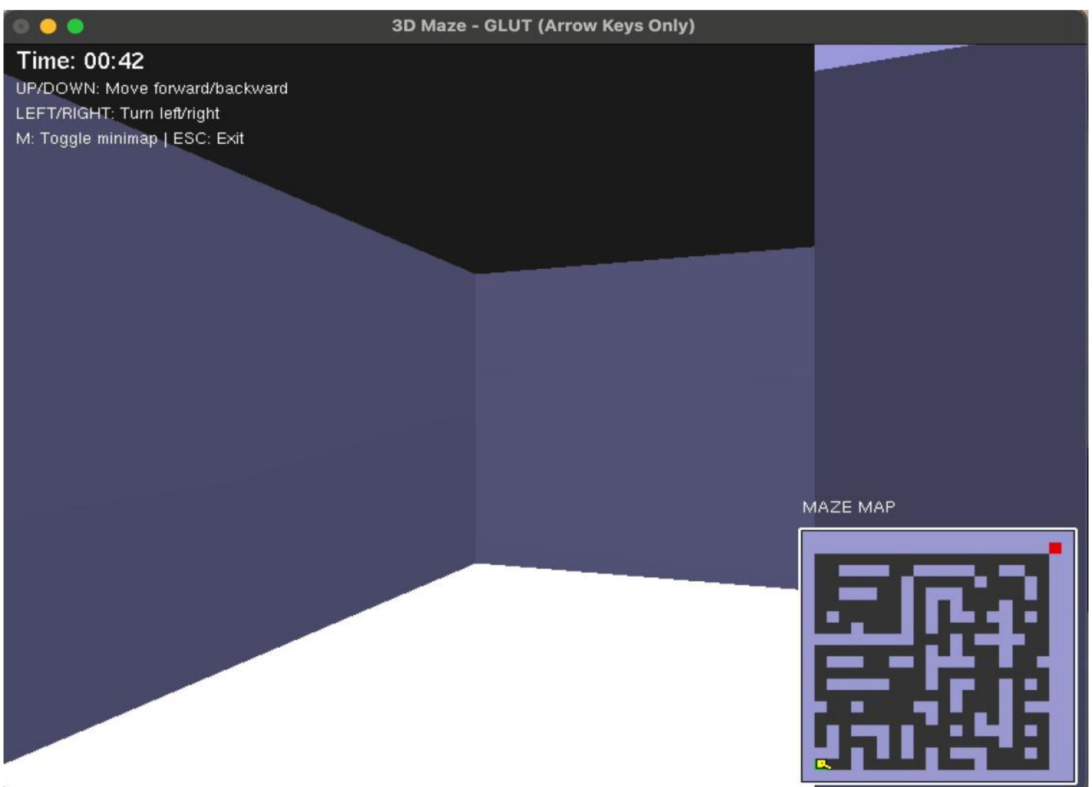
```
    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();
}

void drawText(float x, float y, const char* text)
{
    // Save current matrices and attributes
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();

    // Set up orthographic projection
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-1.0, 1.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // Disable lighting
    glDisable(GL_LIGHTING);

    // Draw text
    glColor3f(1.0f, 1.0f, 0.0f);
    glRasterPos2f(x, y);
    for (int i = 0; text[i] != '\0'; i++) {
        glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, text[i]);
    }

    // Re-enable lighting
    glEnable(GL_LIGHTING);

    // Restore matrices
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();
}

bool checkCollision(float x, float y, float z)
{
    // Calculate grid cell coordinates
    int gridX = (int)(x + 0.5f);
    int gridZ = (int)(z + 0.5f);

    // Check if out of bounds
    if (gridX < 0 || gridX >= MAZE_SIZE + 2 || gridZ < 0 || gridZ >= MAZE_SIZE + 2) {
        return true;
    }

    // Check collision with wall
    if (maze[gridX][gridZ] == 1) {
        return true;
    }

    // Allow movement in empty space
    return false;
}
```

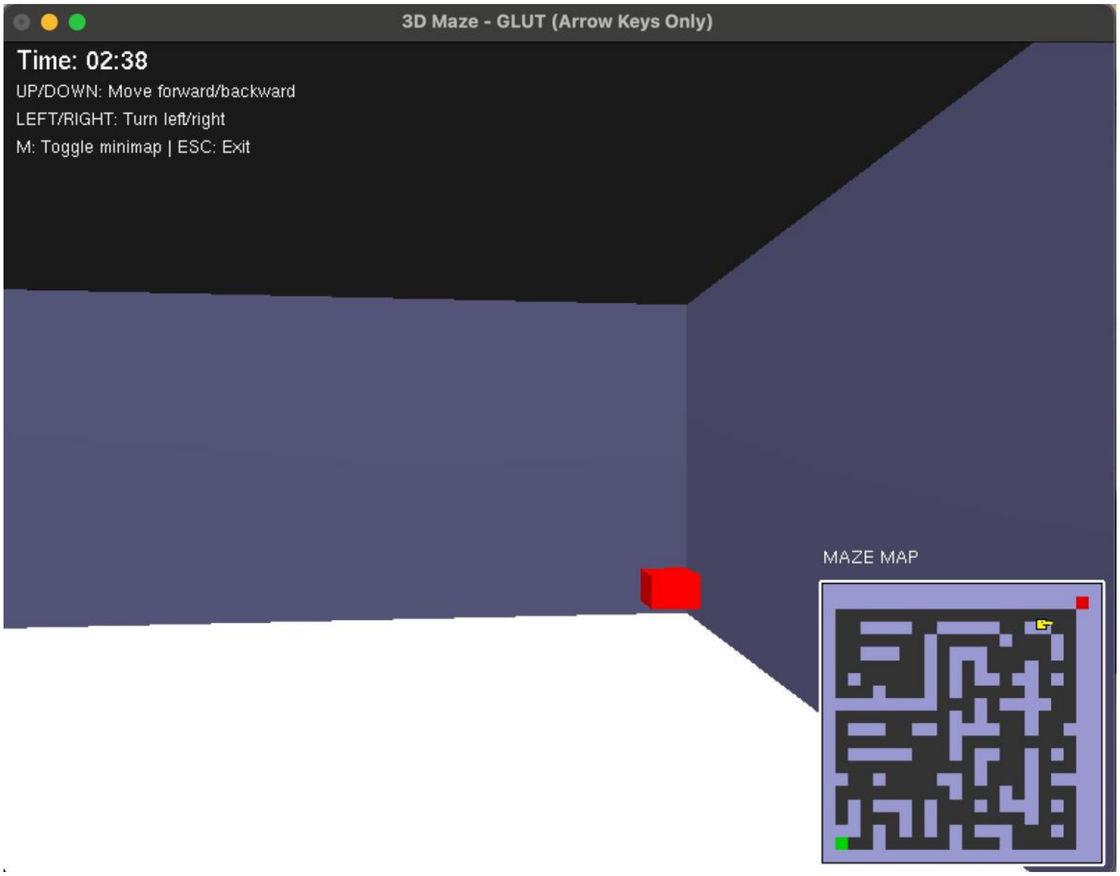
PROJECT SCREENSHOTS






Time: 00:35
UP/DOWN: Move forward/backward
LEFT/RIGHT: Turn left/right
M: Toggle minimap | ESC: Exit
MAZE MAP


3D Maze - GLUT (Arrow Keys Only)
Time: 00:42
UP/DOWN: Move forward/backward
LEFT/RIGHT: Turn left/right
M: Toggle minimap | ESC: Exit
MAZE MAP


3D Maze - GLUT (Arrow Keys Only)
Time: 02:38
UP/DOWN: Move forward/backward
LEFT/RIGHT: Turn left/right
M: Toggle minimap | ESC: Exit
MAZE MAP


CONGRATULATIONS!
You successfully completed the maze!
Time: 01:24
Press SPACE to continue