## Header Inclusions

|  |
| --- |
| #include<stdio.h>  #include<stdlib.h>  #include<GL/glut.h>  #include<math.h>  #include<string.h>  #include<stdbool.h>  #include<time.h>  #include<Windows.h>  #include<mmsystem.h> |

* **Standard Libraries**:
  + <stdio.h>: Provides standard input/output functions.
  + <stdlib.h>: Offers memory allocation, control, conversion, and other utilities.
  + <math.h>: Used for mathematical operations, potentially for player movement or graphical adjustments.
  + <string.h>: Provides string manipulation functions.
  + <stdbool.h>: Allows for boolean data type usage, enhancing readability and functionality for conditions.
* **OpenGL/GLUT Libraries**:
  + <GL/glut.h>: This library manages window creation, rendering, and interaction via OpenGL, essential for game rendering.
* **Windows-specific Libraries**:
  + <Windows.h>: Allows for system-specific functions, such as control over timing.
  + <mmsystem.h>: Used for multimedia functions, like playing sounds.

## Global Constants and Variables

### Constants

|  |
| --- |
| #define WIDTH 620  #define HEIGHT 620  #define CELL\_SIZE 20  #define TIMER\_INTERVAL 1000 // Timer interval in milliseconds  #define TIME\_LIMIT 60 |

* **Maze and Screen Dimensions**:
  + WIDTH and HEIGHT set the maze's total dimensions in pixels, while CELL\_SIZE defines each cell's width and height.
* **Timer Parameters**:
  + TIMER\_INTERVAL sets the countdown speed to 1 second per decrement.
  + TIME\_LIMIT represents the countdown duration (60 seconds) in which the player must complete the maze.

### Global Variables

|  |
| --- |
| int maze[HEIGHT / CELL\_SIZE][WIDTH / CELL\_SIZE];  int visited[HEIGHT / CELL\_SIZE][WIDTH / CELL\_SIZE];  int path[HEIGHT / CELL\_SIZE][WIDTH / CELL\_SIZE];  int ratRow, ratCol;  int entranceRow, entranceCol;  int exitRow, exitCol;  int gameState = 0; // 0: Start menu, 1: In-game, 2: Win, 3: Lose, 4: Show path  int timer; |

* **Maze Structure and Navigation**:
  + maze[][]: Stores the maze structure as a 2D matrix where each cell is either open or blocked.
  + visited[][]: Tracks visited cells during maze generation and player movement.
  + path[][]: Contains the precomputed solution path from start to end.
* **Game Elements**:
  + ratRow and ratCol: Track the rat's current position in the maze.
  + entranceRow and entranceCol: Store the coordinates for the maze's starting point.
  + exitRow and exitCol: Set the destination (end) of the maze.
* **Game State and Timer**:
  + gameState: Manages the current game phase. Possible values include:
    - 0: Start menu
    - 1: In-game
    - 2: Win state
    - 3: Lose state
    - 4: Show solution path
  + timer: Counts down from TIME\_LIMIT (60 seconds) to zero.

## OpenGL and Game Initialization

### init Function

|  |
| --- |
| void init()  {  glClearColor(0.5, 0.5, 0.5, 0.0); // Grey background  glMatrixMode(GL\_PROJECTION);  glLoadIdentity();  gluOrtho2D(0.0, WIDTH, HEIGHT, 0.0);  glMatrixMode(GL\_MODELVIEW);  glLoadIdentity();  srand(time(NULL));  } |

* **Purpose**: Initializes the OpenGL environment and sets up the graphical window for rendering, as well as the random seed for maze generation.
* **Background Color**:
  + glClearColor(0.5, 0.5, 0.5, 0.0): Sets a grey background color for the game window using RGBA values.
* **Projection Settings**:
  + glMatrixMode(GL\_PROJECTION): Switches to projection matrix mode, which defines the camera projection in OpenGL.
  + glLoadIdentity(): Resets the projection matrix to the default identity matrix.
  + gluOrtho2D(0.0, WIDTH, HEIGHT, 0.0): Establishes a 2D orthographic projection for rendering, where (0,0) is the top-left corner, and the WIDTH and HEIGHT constants define the window size.
* **Model View Matrix**:
  + glMatrixMode(GL\_MODELVIEW): Switches back to the model view matrix mode to manage object transformations.
  + glLoadIdentity(): Resets this matrix to ensure no transformations are applied initially.
* **Random Seed**:
  + srand(time(NULL)): Seeds the random number generator using the current time, ensuring that the maze generated each time is unique.

## Maze Generation Function

### generateMaze Function

|  |
| --- |
| void generateMaze(int row, int col)  {  visited[row][col] = 1;  // Directions: right, down, left, up  int dir[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};  int permutation[4] = {0, 1, 2, 3};  // Randomizing directions  for (int i = 0; i < 4; i++)  {  int j = rand() % 4;  int temp = permutation[i];  permutation[i] = permutation[j];  permutation[j] = temp;  }  for (int i = 0; i < 4; i++)  {  int r = row + dir[permutation[i]][0] \* 2;  int c = col + dir[permutation[i]][1] \* 2;  // Cutting through the maze  if (r >= 0 && r < HEIGHT / CELL\_SIZE && c >= 0 && c < WIDTH / CELL\_SIZE && !visited[r][c])  {  maze[row + dir[permutation[i]][0]][col + dir[permutation[i]][1]] = 1;  maze[row + dir[permutation[i]][0] \* 2][col + dir[permutation[i]][1] \* 2] = 1;  generateMaze(r, c);  }  }  } |

* **Purpose**: This function uses Depth-First Search (DFS) to generate a random maze by carving paths through a grid, ensuring a "perfect maze" with a unique solution.
* **Parameters**:
  + row, col: Current cell coordinates from which the function recursively carves a path.
* **Visited Tracking**:
  + visited[row][col] = 1: Marks the current cell as visited to prevent revisiting.
* **Direction Array**:
  + int dir[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}}: Defines movement in four cardinal directions: right, down, left, and up.
* **Random Direction Permutation**:
  + The permutation array holds indices for each direction. The function shuffles this array to introduce randomness in the path.
* **Path Creation**:
  + For each direction in the permutation array:
    - The function calculates a new cell position (r, c) two steps away to ensure maze pathways are separated by walls.
    - If the new cell is within bounds and not visited, the function "cuts through" walls by setting adjacent cells to 1 to open up the path.
    - The function then recursively calls generateMaze(r, c) to continue carving the maze.

This function recursively builds the maze by dividing it into passages and walls, ultimately creating a single solution path from start to finish.

## BFS Queue

### Cell Struct

|  |
| --- |
| typedef struct {  int row;  int col;  } Cell; |

* **Purpose**: The Cell structure stores the coordinates of a cell in the maze. This struct makes it easy to handle and pass coordinates through the BFS queue, as each Cell instance holds the row and col of a specific cell.

### BFS Queue

The queue is used in the BFS algorithm for exploring the maze. Each enqueued cell represents a point to explore in the search for the path from the entrance to the exit.

|  |
| --- |
| Cell queue[HEIGHT \* WIDTH];  int front = -1, rear = -1; |

* **Queue Variables**:
  + queue: A 1D array of Cell structs, with a maximum size equal to the number of cells in the maze.
  + front and rear: Indices for managing the front and rear of the queue.

## Queue Functions

### enqueue

|  |
| --- |
| void enqueue(Cell cell) {  if (rear == HEIGHT \* WIDTH - 1) {  printf("Queue is full\n");  return;  }  if (front == -1) {  front = 0;  }  rear++;  queue[rear] = cell;  } |

* **Purpose**: Adds a new cell to the rear of the queue, enabling BFS to explore it.
* **Logic**:
  + If the queue is full, a message is printed, and the function exits.
  + If the queue is initially empty (front == -1), the front is set to 0 to initialize it.
  + The rear index is incremented, and the cell is added to the queue.

### isEmpty

|  |
| --- |
| bool isEmpty() {  return front == -1 || front > rear;  } |

* **Purpose**: Checks if the queue is empty.
* **Logic**:
  + Returns true if front is -1 or if front exceeds rear (indicating all cells have been dequeued).

### Dequeue

|  |
| --- |
| Cell dequeue() {  if (isEmpty()) {  printf("Queue is empty\n");  exit(1);  }  Cell cell = queue[front];  front++;  return cell;  } |

* **Purpose**: Removes and returns the cell at the front of the queue.
* **Logic**:
  + If the queue is empty, an error message is printed, and the program exits.
  + Otherwise, the cell at front is returned, and front is incremented.

### isValid

|  |
| --- |
| bool isValid(int row, int col, int maze[][WIDTH / CELL\_SIZE], int visited[][WIDTH / CELL\_SIZE]) {  return (row >= 0 && row < HEIGHT / CELL\_SIZE && col >= 0 && col < WIDTH / CELL\_SIZE && maze[row][col] == 1 && !visited[row][col]);  } |

* **Purpose**: Checks if a cell is valid for exploration.
* **Parameters**:
  + row, col: Coordinates of the cell to check.
  + maze[][]: The maze structure, where 1 indicates a passable cell.
  + visited[][]: The visited status of cells.
* **Logic**:
  + Returns true if the cell is within bounds, passable, and has not been visited.

## Pathfinding Function: findPath

### findPath

|  |
| --- |
| void findPath(int maze[][WIDTH / CELL\_SIZE], int entranceRow, int entranceCol, int exitRow, int exitCol, int path[][WIDTH / CELL\_SIZE]) {  memset(visited, 0, sizeof(visited)); // Reset visited matrix  memset(path, 0, sizeof(path)); // Initialize path matrix  Cell entrance = {entranceRow, entranceCol};  enqueue(entrance);  visited[entranceRow][entranceCol] = 1;  while (!isEmpty()) {  Cell current = dequeue();  int row = current.row;  int col = current.col;  // Check if reached exit  if (row == exitRow && col == exitCol) {  // Mark the path  while (row != entranceRow || col != entranceCol) {  path[row][col] = 1;  Cell parent = queue[row \* WIDTH + col];  row = parent.row;  col = parent.col;  }  path[entranceRow][entranceCol] = 1;  break;  }  // Check neighbors  int rowOffsets[] = {-1, 0, 1, 0};  int colOffsets[] = {0, 1, 0, -1};  for (int i = 0; i < 4; ++i) {  int nextRow = row + rowOffsets[i];  int nextCol = col + colOffsets[i];  if (isValid(nextRow, nextCol, maze, visited)) {  Cell nextCell = {nextRow, nextCol};  enqueue(nextCell);  visited[nextRow][nextCol] = 1;  // Mark the parent cell for reconstructing the path later  queue[nextRow \* WIDTH + nextCol] = current;  }  }  }  } |

* **Purpose**: Uses BFS to find the shortest path from the entrance to the exit in the maze and stores this path.
* **Parameters**:
  + maze[][]: 2D array representing the maze structure.
  + entranceRow, entranceCol: Coordinates of the maze's entrance.
  + exitRow, exitCol: Coordinates of the maze's exit.
  + path[][]: 2D array to store the solution path.
* **Logic**:
  + Initializes visited and path matrices.
  + Adds the entrance cell to the queue and marks it as visited.
  + While the queue is not empty:
    - Dequeues a cell (current) and checks if it matches the exit coordinates.
    - If so, traces back to the entrance using parent pointers in queue[], marking each cell in the path matrix.
  + Explores neighboring cells in four directions (up, right, down, left).
    - If a neighbor is valid, it is enqueued, marked as visited, and the parent cell is stored for path reconstruction.

The BFS algorithm efficiently finds the shortest path in the maze, which is later used to guide the player if needed or to display the solution.

## Rendering Text and Maze Elements

### drawHelveticaString

|  |
| --- |
| void drawHelveticaString(const char\* str)  {  while (\*str)  {  glutBitmapCharacter(GLUT\_BITMAP\_HELVETICA\_18, \*str);  str++;  }  } |

* **Purpose**: Renders a string of text in the GLUT\_BITMAP\_HELVETICA\_18 font.
* **Parameters**:
  + str: Pointer to the string to be rendered.
* **Logic**:
  + Iterates through each character in the string and displays it on the screen using glutBitmapCharacter.

### drawRomanString

|  |
| --- |
| void drawRomanString(const char\* str)  {  while (\*str)  {  glutBitmapCharacter(GLUT\_BITMAP\_TIMES\_ROMAN\_24, \*str);  str++;  }  } |

* **Purpose**: Similar to drawHelveticaString, but uses the GLUT\_BITMAP\_TIMES\_ROMAN\_24 font for a different text style.
* **Parameters**:
  + str: Pointer to the string to be rendered.

Both drawHelveticaString and drawRomanString are utility functions for text rendering, providing flexibility in font choice based on the game’s needs.

## Drawing the Player (Rat) and Key Maze Elements

### drawRat

|  |
| --- |
| void drawRat()  {  glColor3f(0.33, 0.0, 0.0); // Brown color for rat  glBegin(GL\_TRIANGLE\_FAN);  glVertex2f(ratCol \* CELL\_SIZE + CELL\_SIZE / 2, ratRow \* CELL\_SIZE + CELL\_SIZE / 2); // Center of the circle  float radius = CELL\_SIZE / 2;  int numTriangles = 20; // Number of triangles to approximate the circle  for (int i = 0; i <= numTriangles; ++i)  {  float angle = 2.0 \* M\_PI \* i / numTriangles;  float x = ratCol \* CELL\_SIZE + CELL\_SIZE / 2 + radius \* cos(angle);  float y = ratRow \* CELL\_SIZE + CELL\_SIZE / 2 + radius \* sin(angle);  glVertex2f(x, y);  }  glEnd();  } |

* **Purpose**: Renders the player-controlled character, represented as a brown circle in the maze.
* **Logic**:
  + Sets the drawing color to brown with glColor3f(0.33, 0.0, 0.0).
  + Begins a GL\_TRIANGLE\_FAN to draw a filled circle, centered at the player's position (ratRow, ratCol).
  + A for loop iterates through angles to create points along the circle’s circumference, which forms the rat’s circular shape when connected.

### drawEntranceAndExit

|  |
| --- |
| void drawEntranceAndExit()  {  glColor3f(1.0, 1.0, 0.0); // Yellow font  // Draw start and finish  glRasterPos2i(0, 15);  drawHelveticaString("START");  glRasterPos2i(WIDTH - 40, HEIGHT - 5);  drawHelveticaString("END");  // Entrance (green)  glColor3f(0.0, 1, 0.0); // Green color for entrance  glBegin(GL\_QUADS);  glVertex2i(entranceCol \* CELL\_SIZE, entranceRow \* CELL\_SIZE);  glVertex2i(entranceCol \* CELL\_SIZE + CELL\_SIZE, entranceRow \* CELL\_SIZE);  glVertex2i(entranceCol \* CELL\_SIZE + CELL\_SIZE, entranceRow \* CELL\_SIZE + CELL\_SIZE);  glVertex2i(entranceCol \* CELL\_SIZE, entranceRow \* CELL\_SIZE + CELL\_SIZE);  glEnd();  // Exit (red)  glColor3f(1, 0.0, 0.0); // Red color for exit  glBegin(GL\_QUADS);  glVertex2i(exitCol \* CELL\_SIZE, exitRow \* CELL\_SIZE);  glVertex2i(exitCol \* CELL\_SIZE + CELL\_SIZE, exitRow \* CELL\_SIZE);  glVertex2i(exitCol \* CELL\_SIZE + CELL\_SIZE, exitRow \* CELL\_SIZE + CELL\_SIZE);  glVertex2i(exitCol \* CELL\_SIZE, exitRow \* CELL\_SIZE + CELL\_SIZE);  glEnd();  } |

* **Purpose**: Renders the entrance and exit cells of the maze, as well as labels ("START" and "END") to indicate each.
* **Logic**:
  + Sets the color to yellow for text, places raster positions near the top-left and bottom-right of the maze, and renders "START" and "END" using drawHelveticaString.
  + Renders the entrance with a green color by drawing a filled square (GL\_QUADS) at the entrance coordinates (entranceRow, entranceCol).
  + Renders the exit with a red color in a similar manner, placing a filled square at (exitRow, exitCol).

## Game Interface Elements

### drawStartMenu

|  |
| --- |
| void drawStartMenu()  {  glColor3f(0.0, 0.0, 0.0); // Black font  // Draw menu options  glRasterPos2i(30, 235);  drawRomanString("Start Menu");  glRasterPos2i(30, 240);  drawRomanString("...................");  glRasterPos2i(30, 260);  drawHelveticaString("Press F1 to Start New Game");  glRasterPos2i(30, 280);  drawHelveticaString("Press F2 to Quit");  glRasterPos2i(30, 315);  drawRomanString("How to Play");  glRasterPos2i(30, 320);  drawRomanString("....................");  glRasterPos2i(30, 340);  drawHelveticaString("Use arrow keys to move and escape the maze before time runs out.");  glRasterPos2i(30, 380);  drawHelveticaString("You are given 60 seconds to escape the maze.");  glRasterPos2i(30, 400);  drawHelveticaString("The countdown starts the moment you start a new game.");  glRasterPos2i(30, 440);  drawHelveticaString("The maze does not change when you start a new game.");  glRasterPos2i(30, 460);  drawHelveticaString("If you lose, feel free to return to Start Menu and try again.");  glRasterPos2i(30, 500);  drawHelveticaString("Revealing the path will end the game.");  glRasterPos2i(30, 540);  drawHelveticaString("To try a new maze, quit the program and restart it.");  } |

* **Purpose**: Displays the main menu screen, providing game instructions and options for the player.
* **Logic**:
  + Sets text color to black using glColor3f(0.0, 0.0, 0.0).
  + Each menu line uses glRasterPos2i to position the text on the screen, followed by drawHelveticaString or drawRomanString to render the text.
  + Text includes options to start a new game (F1), quit (F2), game instructions, and tips for gameplay.

The menu serves as an initial interface, allowing the player to learn the game mechanics before beginning.

### drawTimer

|  |
| --- |
| void drawTimer()  {  glColor3f(1.0, 0.0, 1.0); // Pink font  glRasterPos2i(WIDTH - 100, 15);  char timeStr[15];  snprintf(timeStr, sizeof(timeStr), "Time: %d s", timer);  drawHelveticaString(timeStr);  } |

* **Purpose**: Renders the countdown timer on the game screen, showing the remaining time for the player to complete the maze.
* **Logic**:
  + Sets text color to pink (glColor3f(1.0, 0.0, 1.0)) for visibility.
  + Places the timer text at the top-right of the screen using glRasterPos2i.
  + Formats the remaining time into a string ("Time: %d s") using snprintf, then displays it with drawHelveticaString.

The countdown timer serves as a constant reminder of the game's time constraint, adding urgency to the player’s objective.

## Maze Rendering Functions

### drawMaze

|  |
| --- |
| void drawMaze()  {  glColor3f(0.0, 0.0, 0.0); // Black walls  // Draw maze walls  for (int i = 0; i < HEIGHT / CELL\_SIZE; i++)  {  for (int j = 0; j < WIDTH / CELL\_SIZE; j++)  {  if (maze[i][j] == 0)  {  glBegin(GL\_QUADS);  glVertex2i(j \* CELL\_SIZE, i \* CELL\_SIZE);  glVertex2i(j \* CELL\_SIZE + CELL\_SIZE, i \* CELL\_SIZE);  glVertex2i(j \* CELL\_SIZE + CELL\_SIZE, i \* CELL\_SIZE + CELL\_SIZE);  glVertex2i(j \* CELL\_SIZE, i \* CELL\_SIZE + CELL\_SIZE);  glEnd();  }  }  }  drawEntranceAndExit();  drawRat();  drawTimer();  glColor3f(0.0, 1.0, 0.0);  glRasterPos2i(WIDTH / 2 - 100, HEIGHT - 5);  drawHelveticaString("Press F1 to Reveal Path");  } |

* **Purpose**: Renders the maze with walls, the rat (player's character), the entrance and exit, and a countdown timer. It also provides an option to reveal the maze solution.
* **Logic**:
  + Sets the wall color to black (glColor3f(0.0, 0.0, 0.0)) and draws each cell marked as a wall (maze[i][j] == 0) as a black square.
  + Calls auxiliary functions to draw the entrance/exit (drawEntranceAndExit), the rat (drawRat), and the timer (drawTimer).
  + Displays a prompt at the top-center, advising the player to press F1 to reveal the solution path.

This function is called during active gameplay, providing a clear view of the maze and game elements.

### drawPath

|  |
| --- |
| void drawPath()  {  glColor3f(0.0, 0.0, 0.0); // Black walls  // Draw maze walls  for (int i = 0; i < HEIGHT / CELL\_SIZE; i++)  {  for (int j = 0; j < WIDTH / CELL\_SIZE; j++)  {  if (maze[i][j] == 0)  {  glBegin(GL\_QUADS);  glVertex2i(j \* CELL\_SIZE, i \* CELL\_SIZE);  glVertex2i(j \* CELL\_SIZE + CELL\_SIZE, i \* CELL\_SIZE);  glVertex2i(j \* CELL\_SIZE + CELL\_SIZE, i \* CELL\_SIZE + CELL\_SIZE);  glVertex2i(j \* CELL\_SIZE, i \* CELL\_SIZE + CELL\_SIZE);  glEnd();  }  }  }  drawEntranceAndExit();  glColor3f(0.0, 1.0, 0.0); // Green path  // Draw solution path  for (int i = 0; i < HEIGHT / CELL\_SIZE; i++)  {  for (int j = 0; j < WIDTH / CELL\_SIZE; j++)  {  if (path[i][j] == 1)  {  glBegin(GL\_QUADS);  glVertex2i(j \* CELL\_SIZE, i \* CELL\_SIZE);  glVertex2i(j \* CELL\_SIZE + CELL\_SIZE, i \* CELL\_SIZE);  glVertex2i(j \* CELL\_SIZE + CELL\_SIZE, i \* CELL\_SIZE + CELL\_SIZE);  glVertex2i(j \* CELL\_SIZE, i \* CELL\_SIZE + CELL\_SIZE);  glEnd();  }  }  }  // Message to return to start menu  glColor3f(0.0, 1.0, 0.0);  glRasterPos2i(WIDTH / 2 - 100, HEIGHT - 5);  drawHelveticaString("Press F1 to return to Start Menu");  } |

* **Purpose**: Displays the maze along with the solution path, highlighting the correct route from the start to the end point.
* **Logic**:
  + Like drawMaze, it renders the maze walls, entrance, and exit.
  + Additionally, any cell in the path matrix marked as part of the solution (path[i][j] == 1) is drawn in green.
  + Displays a prompt to return to the start menu by pressing F1.

This function is called when the player opts to reveal the solution, effectively ending the game and allowing the player to analyze the solution path.

## Game End Screens

### winnerScreen

|  |
| --- |
| void winnerScreen()  {  // Game over message (player wins)  glColor3f(0.0, 1.0, 0.0); // Green font  glRasterPos2i(WIDTH / 2 - 40, HEIGHT / 2);  drawRomanString("YOU WIN!");  // Return to start menu message  glColor3f(0.0, 0.0, 0.0);  glRasterPos2i(WIDTH / 2 - 100, HEIGHT / 2 + 20);  drawHelveticaString("Press F1 to return to Start Menu");  } |

* **Purpose**: Displays a congratulatory "YOU WIN!" message when the player successfully navigates the maze before the timer expires.
* **Logic**:
  + The function sets the text color to green (glColor3f(0.0, 1.0, 0.0)) to highlight the winning message.
  + Displays the message "YOU WIN!" in the center of the screen using drawRomanString.
  + Displays an additional prompt in black, advising the player to press F1 to return to the start menu.

This function creates a positive end screen, allowing the player to easily start a new game if desired.

### loserScreen

|  |
| --- |
| void loserScreen()  {  // Game over message (player loses)  glColor3f(1.0, 0.0, 0.0); // Red font  glRasterPos2i(WIDTH / 2 - 40, HEIGHT / 2);  drawRomanString("YOU LOSE!");  // Return to start menu message  glColor3f(0.0, 0.0, 0.0);  glRasterPos2i(WIDTH / 2 - 100, HEIGHT / 2 + 20);  drawHelveticaString("Press F1 to return to Start Menu");  } |

* **Purpose**: Displays a "YOU LOSE!" message when the player fails to reach the end of the maze within the time limit.
* **Logic**:
  + The text color is set to red (glColor3f(1.0, 0.0, 0.0)) to emphasize the loss.
  + The "YOU LOSE!" message is centered on the screen, followed by a black prompt suggesting pressing F1 to return to the start menu.

This function clearly informs the player of the game's end and provides an option to try again.

## Main Game Loop and User Input Handling

### display

|  |
| --- |
| void display()  {  glClear(GL\_COLOR\_BUFFER\_BIT);  if (gameState == 0)  {  drawStartMenu();  }  else if (gameState == 1)  {  drawMaze();  }  else if (gameState == 2)  {  winnerScreen();  }  else if (gameState == 3)  {  loserScreen();  }  else if (gameState == 4)  {  drawPath();  }  glutSwapBuffers();  } |

* **Purpose**: This function manages the main rendering logic based on the current game state, determining which screen or view to display.
* **Logic**:
  + Clears the current buffer with glClear(GL\_COLOR\_BUFFER\_BIT).
  + Checks gameState to determine which screen to render:
    - 0 - Start menu
    - 1 - Maze (gameplay)
    - 2 - Win screen
    - 3 - Lose screen
    - 4 - Revealed path screen
  + Swaps the display buffer with glutSwapBuffers() to update the view smoothly.

This function centralizes display management, making it easy to expand or modify state-based rendering.

### handleStartMenuKeys

|  |
| --- |
| void handleStartMenuKeys(int key)  {  if (key == GLUT\_KEY\_F1) // Start new game  {  gameState = 1;  ratRow = entranceRow; // Set rat's position to entrance  ratCol = entranceCol;  timer = TIME\_LIMIT; // Reset timer  glutPostRedisplay();  }  else if (key == GLUT\_KEY\_F2) // Quit  {  exit(0);  }  } |

* **Purpose**: Manages keyboard input for navigating the start menu and starting or quitting the game.
* **Logic**:
  + If F1 is pressed, it initializes a new game by setting gameState to 1, positioning the "rat" at the maze entrance, and resetting the timer.
  + If F2 is pressed, the program exits.

This function effectively starts the game or quits based on the player's input.

### handleInGameStateKeys

|  |
| --- |
| void handleInGameStateKeys(int key)  {  switch (key)  {  case GLUT\_KEY\_UP:  if (ratRow > 0 && maze[ratRow - 1][ratCol] == 1)  ratRow--;  break;  case GLUT\_KEY\_DOWN:  if (ratRow < HEIGHT / CELL\_SIZE - 1 && maze[ratRow + 1][ratCol] == 1)  ratRow++;  break;  case GLUT\_KEY\_LEFT:  if (ratCol > 0 && maze[ratRow][ratCol - 1] == 1)  ratCol--;  break;  case GLUT\_KEY\_RIGHT:  if (ratCol < WIDTH / CELL\_SIZE - 1 && maze[ratRow][ratCol + 1] == 1)  ratCol++;  break;  case GLUT\_KEY\_F1:  gameState = 4;  break;  }  // Check if the rat reached the exit  if (ratRow == exitRow && ratCol == exitCol)  {  gameState = 2; // Player wins  }  glutPostRedisplay();  } |

* **Purpose**: Handles player movement and in-game actions using the arrow keys and function keys.
* **Logic**:
  + Arrow keys control movement within the maze by adjusting ratRow and ratCol if movement is valid.
  + Pressing F1 switches gameState to 4 to reveal the maze solution.
  + If the player reaches the exit coordinates (exitRow and exitCol), gameState is set to 2 to indicate a win.
  + glutPostRedisplay() updates the display after each movement or action.

This function enables player control within the maze, checks for the win condition, and allows the player to reveal the solution.

## Key Handling and Timer Management Functions

### handleEndGameStateKeys

|  |
| --- |
| void handleEndGameStateKeys(int key) {  if (key == GLUT\_KEY\_F1) { // Return to start menu  gameState = 0;  glutPostRedisplay();  }  } |

* **Purpose**: Manages keyboard input for the end game screens (win, lose, or path reveal), allowing the player to return to the start menu.
* **Logic**:
  + If F1 is pressed, the game returns to the start menu by setting gameState to 0.
  + Calls glutPostRedisplay() to refresh the display.

This function provides an option to reset the game back to the start menu after reaching the end of a game state.

### specialKeys

|  |
| --- |
| void specialKeys(int key, int x, int y)  {  if (gameState == 0)  {  handleStartMenuKeys(key);  }  else if (gameState == 1)  {  handleInGameStateKeys(key);  }  else if (gameState == 2 || gameState == 3 || gameState == 4)  {  handleEndGameStateKeys(key);  }  } |

* **Purpose**: Acts as a central handler for special key inputs, routing them to specific functions based on the current gameState.
* **Logic**:
  + Routes gameState-dependent inputs:
    - **gameState == 0**: Calls handleStartMenuKeys() for the start menu.
    - **gameState == 1**: Calls handleInGameStateKeys() during gameplay.
    - **gameState == 2, 3, or 4**: Calls handleEndGameStateKeys() to handle inputs on the win, lose, or path reveal screens.

This function provides a structured way to manage key input by routing it to appropriate functions based on the active game screen.

### updateTimer

|  |
| --- |
| void updateTimer(int value)  {  if (gameState == 1)  {  timer--;  if (timer <= 0) {  gameState = 3; // Player loses  }  }  glutPostRedisplay();  glutTimerFunc(TIMER\_INTERVAL, updateTimer, 0);  } |

* **Purpose**: Manages the game countdown timer, updating it periodically and checking if the player has run out of time.
* **Logic**:
  + Decrements timer if gameState is 1 (game in progress).
  + If timer reaches 0, sets gameState to 3, indicating a loss.
  + Calls glutPostRedisplay() to refresh the screen.
  + Sets a recursive timer using glutTimerFunc(TIMER\_INTERVAL, updateTimer, 0) to trigger updateTimer() again after TIMER\_INTERVAL milliseconds.

This function maintains the countdown timer, ensuring that the game updates every second and transitions to the lose screen if time runs out.

## Main Program and Initialization

### main

|  |
| --- |
| int main(int argc, char \*\*argv)  {  glutInit(&argc, argv);  glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB);  glutInitWindowSize(WIDTH, HEIGHT);  glutCreateWindow("RatA");  init();  for (int i = 0; i < HEIGHT / CELL\_SIZE; i++)  {  for (int j = 0; j < WIDTH / CELL\_SIZE; j++)  {  // Initialize maze and visited arrays with 0 (walls)  maze[i][j] = 0;  visited[i][j] = 0;  }  }  generateMaze(1, 1);  // Print maze to terminal for reference  for (int i = 0; i < HEIGHT / CELL\_SIZE; i++)  {  for (int j = 0; j < WIDTH / CELL\_SIZE; j++)  {  printf("%d ", maze[i][j]);  }  printf("\n");  }  printf("\n");  // Set entrance and exit positions  entranceRow = 1;  entranceCol = 1;  exitRow = HEIGHT / CELL\_SIZE - 2;  exitCol = WIDTH / CELL\_SIZE - 1;  // Open entrance and exit in the maze  maze[entranceRow][entranceCol] = 1;  maze[exitRow][exitCol] = 1;  // Find the solution path from entrance to exit  findPath(maze, entranceRow, entranceCol, exitRow, exitCol, path);  // Save maze layout to a text file  FILE \*file = fopen("maze.txt", "a");  if (file == NULL) {  printf("Error opening file.\n");  exit(1);  }  for (int i = 0; i < HEIGHT / CELL\_SIZE; i++) {  for (int j = 0; j < WIDTH / CELL\_SIZE; j++) {  fprintf(file, "%d ", maze[i][j]);  }  fprintf(file, "\n");  }  fprintf(file, "\n");  fclose(file);  // Initialize rat's position at the entrance  ratRow = entranceRow;  ratCol = entranceCol;  // Start timer  glutTimerFunc(TIMER\_INTERVAL, updateTimer, 0);  // Set up display and key input handling  glutDisplayFunc(display);  PlaySound(TEXT("candyland.wav"), NULL, SND\_ASYNC | SND\_FILENAME | SND\_LOOP);  glutSpecialFunc(specialKeys);  // Enter the main event loop  glutMainLoop();  return 0;  } |

**Explanation**

1. **GLUT Initialization**:
   * Initializes the GLUT library and sets up the display mode with double buffering and RGB color.
   * Creates the main game window and initializes the graphics.
2. **Maze and visited Array Initialization**:
   * Initializes maze and visited matrices, setting all cells to 0 (representing walls).
3. **Maze Generation**:
   * Calls generateMaze(1, 1) to create a maze with a randomized structure.
4. **Print Maze to Terminal**:
   * Outputs the maze configuration in the terminal, aiding in debugging and visualization.
5. **Entrance and Exit Setup**:
   * Sets the entrance and exit points, ensuring both are open in the maze matrix.
6. **Pathfinding**:
   * Uses findPath() to generate the solution path from the entrance to the exit, storing it in the path matrix.
7. **Save Maze to Text File**:
   * Opens (or creates) maze.txt and appends the current maze configuration for future reference.
8. **Game Elements Initialization**:
   * Sets the rat’s starting position at the entrance.
   * Initializes and starts the game countdown timer using glutTimerFunc().
9. **Display and Key Bindings**:
   * Sets up the main display function display() to handle rendering.
   * Plays a background sound in loop using PlaySound().
   * Registers specialKeys() to handle keyboard input.
10. **Enter Main Loop**:
    * Starts the GLUT main event loop with glutMainLoop(), making the game interactive.