Computer Graphics (UCS505)
Project Report
on
Cosmic Light Weaver: Space Explorer Game

Submitted by:

Shivane Kapoor (102203191)
Kaustubh Singh (102203194)
Gurleen Kaur (102203197)

B.E. Third Year – 3C15
Submitted to:
Ms. Archana Kumari

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

Computer Science and Engineering Department

Thapar Institute of Engineering and

Technology Patiala – 147001

Table of Contents

Introduction to Project

Computer graphics is an exciting and rapidly evolving field that deals with the creation, manipulation, and display of visual content using computers. It involves the use of mathematical algorithms and computer programming to generate, animate, and render 2D and 3D images, videos, and animations. The application of computer graphics is vast and can be seen in various fields such as entertainment, education, engineering, medicine, and design. From creating visually stunning video games and movies to designing buildings and engineering products, computer graphics plays a vital role in enhancing the visual communication and problem-solving abilities of professionals across different domains.

Our Project: Cosmic Light Weaver:

Cosmic Light Weaver is an immersive top-down space navigation game built using C and OpenGL/GLUT. Players control a spacecraft trapped in a dangerous asteroid field, where they must strategically navigate a grid-based maze to collect all glowing energy cores before escaping through a randomly placed wormhole. The game features three key survival mechanics: a constantly depleting light energy system that drains with each movement, a strict time limit, and procedurally generated asteroid obstacles that vary in density based on the selected difficulty level (Easy, Medium, or Hard).

What makes the game particularly engaging is its intelligent level design - the game uses the A* pathfinding algorithm to guarantee that every randomly generated map has at least one viable solution path, with energy cores deliberately placed along this route to ensure fair gameplay. Visual elements like dynamic particle effects create a captivating atmosphere, with the player's ship leaving a fading energy trail, pulsating energy cores, and a beautifully rendered wormhole exit featuring an animated accretion disk. The ship's glow dynamically dims as energy depletes, providing clear visual feedback about remaining resources. With its perfect balance of strategic pathfinding, resource management tension, and retro-futuristic aesthetics, Cosmic Light Weaver offers a compelling challenge for players who enjoy thoughtful, time-pressured puzzle navigation games.

Computer Graphics Concepts Used

1. 2D Object Representation & Rendering

Grid-Based World

- The game world is a 2D grid (`spaceMap[GRID_HEIGHT][GRID_WIDTH]`), where each cell represents:
    - `0` – empty space
    - `1` – asteroid obstacle
- Objects (player, coins, exit) use **floating-point coordinates** within the grid for smooth transitions.

Primitive Shapes

- **Player Ship**: Drawn as a triangle (`GL_QUADS & GL_TRIANGLE`) with dynamic rotation based on movement.
- **Coins**: Rendered as **lightning bolts** using `GL_LINES` and `GL_TRIANGLE_STRIP`, with animated particle effects.
- **Exit Wormhole**: A circle (`GL_TRIANGLE_FAN`) with a **spiraling accretion disk**.
- **Asteroids**: Simple squares drawn using `GL_TRIANGLE_FAN(Body)and GL_LINE_LOOP` adds a cracked outline.

Coordinate System

- Uses OpenGL's **2D orthographic projection** (`glutOrtho2D()`).
- **Top-left origin**: (0, 0) at the top-left; (GRID_WIDTH×CELL_SIZE, GRID_HEIGHT×CELL_SIZE) at bottom-right.

---

2. 2D Transformations

Translation

- Moves objects to their grid positions.
  Example: `glTranslatef(player.x * CELL_SIZE, player.y * CELL_SIZE, 0)`

Rotation

- Applied only to the **player's ship** to match movement direction.
  - Calculated with: `atan2(dy, dx)`
  - Applied using: `glRotatef(angle_degrees, 0, 0, 1)`

Scaling

- Used for visual adjustments, such as:
  - UI elements (e.g., minimap icons)
  - Particle size scaling for depth illusion.

3. Camera & Viewport (2D Projection)

Static Camera

- A **fixed 2D viewport** using: `glOrtho(0, GRID_WIDTH*CELL_SIZE, GRID_HEIGHT*CELL_SIZE, 0, -1, 1)`

Viewport Management

- Handles resizing using `glutReshapeFunc(reshape)` to scale the grid proportionally without distortion.

---

4. User Interaction & Collision

Input Handling

- **WASD / Arrow Keys**: Move the player one cell at a time.

- **Theme Toggle**: Switches color palette instantly (`updateThemeColors()`).

- **Grid-Based Movement**:
  - Uses `isValidMove()` to verify if the destination cell is walkable (`spaceMap[gridY][gridX] == 0`)
- **Coin Collection**:
  - Based on distance: `sqrt(dx² + dy²) < 0.7f` (handled in `checkCoinCollision()`)
- **Boundary Checks**:
  - Prevents out-of-bounds movement: `player.x ∈ [0, GRID_WIDTH]`

---

5. Lighting & Visual Effects (2D Techniques)

Dynamic Lighting

- **Player Light**:
  - Exhaust glow brightness reflects `player.light` level.
  - Smoke trail particles fade as their intensity decays.

- **Coin Effects**:
  - Pulse using sine animation:
    `sin(glutGet(GLUT_ELAPSED_TIME))`

Particle Systems

- **Stars**:
  - Drawn as `GL_POINTS` with twinkling alpha values.
- **Nebulas**:
  - Use `GL_TRIANGLE_FAN` with **radial gradients** and **sinusoidal pulsation**.
- **Smoke Trail**:
  - `GL_QUADS` fading over time, spawned at player's previous positions.

Theme System

- Swaps between:
  - **darkTheme** (deep blues)
  - **lightTheme** (amber hues)
- Applies globally via `updateThemeColors()`.

---

6. Optimization & Performance

Minimap Rendering

- Uses a **separate `glOrtho()` projection** for UI overlay, reducing calculation overhead.

Efficient Pathfinding

- Implements **A*** algorithm in `pathfindAStar()`:
    - Uses Manhattan distance as the heuristic.

Particle Recycling

- Instead of deleting particles:
    - They are **respawned** when `age >= lifespan`.

---

7. Key Libraries & Functions

Libraries Used

OpenGL/GLUT Libraries

- <GL/glut.h> – Main OpenGL Utility Toolkit (GLUT) library for:
    - Creating windows
    - Handling input
    - Rendering graphics
- **OpenGL Functions** for 2D rendering:
    - glBegin, glEnd, glVertex2f, etc.
- **GLUT Functions** for window management:
    - glutInit, glutCreateWindow, glutDisplayFunc, etc.

Standard C Libraries

- <stdbool.h> – Boolean support (bool, true, false)
- <stdlib.h> – Memory allocation (malloc, free), random number generation (rand, srand), exit()
- <string.h> – String and memory operations (memset, memmove, memcpy)
- <stdio.h> – File I/O (fopen, fclose, fread, fwrite), formatted output (printf, snprintf)
- <time.h> – Time functions (time) for random seed initialization
- <math.h> – Math functions (sin, cos, sqrt, pow, abs, atan2, M_PI)

Key Built-in Functions Used

1. OpenGL/GLUT Functions

Window & Display Management

- glutInit() – Initializes GLUT
- glutInitDisplayMode() – Sets display mode (e.g., GLUT_DOUBLE)
- glutInitWindowSize() / glutInitWindowPosition() – Sets window size and position
- glutCreateWindow() – Creates application window
- glutDisplayFunc() – Registers display callback
- glutReshapeFunc() – Handles window resizing
- glutSwapBuffers() – Swaps front/back buffers for double buffering
- glutPostRedisplay() – Triggers screen redraw

Input Handling

- glutKeyboardFunc() – Keyboard input handling
- glutSpecialFunc() – Handles special keys (arrows, function keys)

Timers & Animation

- glutTimerFunc() – Sets up timed callback functions
- glutGet(GLUT_ELAPSED_TIME) – Gets time elapsed since program start

2D Rendering

- glClear() – Clears screen buffer
- glColor3f() / glColor4f() – Sets RGB/RGBA colors
- glBegin() / glEnd() – Begin and end drawing primitives (GL_QUADS, GL_TRIANGLES, etc.)
- glVertex2f() – Specifies 2D vertex
- glPointSize() / glLineWidth() – Sets size of points and lines

- glEnable(GL_BLEND) – Enables alpha blending (transparency)
- glBlendFunc() – Defines how colors blend
- glMatrixMode() / glLoadIdentity() / glOrtho2D() – Sets 2D orthographic projection
- glPushMatrix() / glPopMatrix() – Saves/restores transformation state
- glScalef() / glTranslatef() / glRotatef() – Applies transformations

Text Rendering

- glutBitmapCharacter() – Renders characters with bitmap fonts
- glRasterPos2f() – Sets position for text rendering

---

2. Standard C Library Functions

Memory & String Operations

- memset() – Initializes memory
- memmove() / memcpy() – Copies memory blocks
- strlen() – Returns string length
- snprintf() – Safe string formatting

File I/O

- fopen() / fclose() – Opens and closes files
- fread() / fwrite() – Reads and writes binary data

Math & Random Numbers

- rand() / srand() – Pseudo-random number generation
- sin(), cos() – Trigonometric functions
- atan2() – Angle calculation using arc tangent

Time Functions

- time() – Returns current time, used for seeding randomness

User Defined Functions

1. Core Game Logic

1.1. `init()`

Purpose: Initializes all game systems for startup.
Key Actions:

- **Sets up OpenGL (blending, clear color)**
- **Generates first map (`generateEnvironment(false)`)**
- **Places player at start position (1.5, 1.5)**
- **Loads best scores from file (`saveLoadBestScore(false)`)**
- **Initializes stars, nebulas, and particles (`initGameObjects()`)**

1.2. `startNewGame()`

Purpose: Resets all game state for a fresh run.
Resets:

- **Player position and light energy**
- **Trail points**
- **Coin states**
- **Timer (`gameTime = 0`)**
- **Generates new map (random or guaranteed path)**

1.3. `update(int value)`

Purpose: Game loop handler (called every 100ms).
Key Updates:

- **Reduces `player.light` based on difficulty**

- **Ages and removes faded trail points**
- **Animates particles (movement/respawning)**
- **Updates nebula pulsation and star twinkling**
- **Checks lose conditions (light/time)**

1.4. `updateTimer(int value)`

Purpose: Increments `gameTime` every second.
Ties to: Time limit checks in `update()`.

1.5. `checkWinCondition()`

Purpose: Checks if player reached exit with all coins.
Mechanism:

- **Distance check to exit (< 0.7f)**
- **Verifies `player.coinsCollected == totalCoins`**
- **Triggers `GAME_WIN` state and saves best score if applicable**

1.6. `checkCoinCollision()`

Purpose: Handles coin collection logic.
Effects:

- **Deactivates collected coins**
- **Boosts `player.light` (scales with difficulty)**
- **Placeholder for audio cue**

1.7. `saveLoadBestScore(bool save)`

Purpose: Manages persistent high scores.

File Handling:

- **Binary file `cosmiclightweaver.dat`**
- **Header `"CLWSAV01"` for version control**
- **Stores best times per difficulty**

1.8. `updateDifficultySettings()`

Purpose: Adjusts game parameters by difficulty.
Configures:

- **`timeLimit` (Easy: 60s, Medium: 45s, Hard: 30s)**
- **`lightDecayRate` (faster depletion for higher difficulties)**

---

2. Map Generation & Pathfinding

2.1. `generateEnvironment(bool guaranteePath)`

Purpose: Creates playable asteroid fields.
Workflow:

- **Calls `generateRandomMap()`**
- **Places exit via `findValidExit()`**
- **Falls back to `createGuaranteedPath()` if needed**
- **Distributes coins using `placeCoins()`**

2.2. `generateRandomMap()`

Purpose: Fills `spaceMap` with asteroid obstacles.

Logic:

- **Difficulty-based density**
- **Preserves clear start area (0,0)-(3,3)**
- **Uses weighted randomness for natural clustering**

2.3. `createGuaranteedPath()`

Purpose: Ensures solvable levels.
Method:

- **Carves a winding path from start to exit**
- **Adds surrounding obstacles**
- **Places coins along the path**

2.4. `findValidExit()`

Purpose: Positions exit far from start.
Constraints:

- **Minimum distance per difficulty (Easy: `GRID_WIDTH/3`)**
- **Clears 3x3 area around exit**

2.5. `placeCoins()`

Purpose: Distributes energy cores.
Rules:

- **Avoids start/exit proximity**
- **Ensures each coin is reachable via `pathfindAStar()`**
- **Adjusts count by difficulty (Easy: 7, Hard: 10)**

2.6. `pathfindAStar()`

Purpose: A* pathfinding between grid points.
Features:

- **8-directional movement (diagonal cost = √2)**
- **Manhattan distance heuristic**
- **Optimized with open/closed sets**

2.7. `verifyAllPathsExist()`

Purpose: Validates level solvability.
Checks:

- **Start → All coins → Exit connectivity using** `pathfindAStar()`

2.8. `heuristic()`

Purpose: Manhattan distance formula.
Formula: `abs(x1 - x2) + abs(y1 - y2)`

---

3. Rendering System

3.1. Entities

3.1.1. `renderPlayer()`

- **Draws ship as oriented triangle**
- **Thruster glow tied to** `player.light`
- **Direction based on last movement vector**

3.1.2. `renderCoins()`

- Animated lightning bolt style
- Electric effects with pulsing brightness

3.1.3. `renderExit()`

- Wormhole visualization
- Black event horizon with spiraling particles

3.2. Environment

3.2.1. `renderSpace()`

- Draws asteroids based on `spaceMap`
- Performs frustum culling

3.2.2. `renderTrail()`

- Fading smoke trail
- Alpha fades via `trail[i].intensity`
- Size scales with age

3.2.3. `renderStarsAndNebulas()`

- Stars: Random twinkling points
- Nebulas: Pulsing clouds with `sin(time)`

3.2.4. `renderBackgroundEffects()`

- **Grid lines with parallax**
- **Distant galaxy effects**

3.3. UI & Menus

3.3.1. `renderHUD()`

- **Displays light meter, time, and coin count**
- **Light bar color-coded**
- **Time pulses red when critical**

3.3.2. `renderMenu()`

- **Difficulty selection**
- **Theme toggle (dark/light)**
- **Animated title**

3.3.3. `renderGameState()`

- **Win/lose screens**
- **Shows mission time**
- **Displays restart prompt**

---

4. Utilities & Helpers

4.1. `updateThemeColors()`

- Switches between dark/light palettes
- Updates all rendering colors

4.2. `isValidMove()`

- Checks grid collision
- Converts position to grid index
- Returns true if cell is walkable (`spaceMap == 0`)

4.3. `addTrailPoint()`

- Adds trail point to circular buffer
- Sets initial intensity to `5.0f`

4.4. `compareNodes()`

- A* sorting helper (compares `f = g + h`)

4.5. `initGameObjects()`

- Spawns:
    - Stars (random positions)
    - Nebulas (color variants)
    - Particles (initial spawn)

5. Input Handling (GLUT Callbacks)

5.1. `keyboard()`

- **Handles:**
    - **Movement (WASD)**
    - **Menu interaction (`Enter`)**
    - **Theme toggle (`T`)**

5.2. `specialKeys()`

- **Handles arrow keys:**
    - **Menu navigation**
    - **Alternative movement**

5.3. `reshape()`

- **Adjusts OpenGL viewport on window resize**

5.4. `display()`

- **Clears screen**
- **Renders all components**
- **Swaps buffers**

Project Source Code

```c
#include <GL/glut.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include <math.h>

// Constants
#define GRID_WIDTH 15
#define GRID_HEIGHT 15
#define CELL_SIZE 40.0f
#define MAX_LIGHT_DURATION 100
#define LIGHT_DECAY_RATE 0.5f
#define MAX_TRAIL_LENGTH 1000
#define M_PI 3.142
#define MAX_COINS 10
#define MAX_PATH_LENGTH 100
#define MAX_STARS 200
#define MAX_NEBULAS 8
#define MAX_PARTICLES 120

#ifndef GLUT_BITMAP_HELVETICA_10
#define GLUT_BITMAP_HELVETICA_10 (void*)4
#endif
#ifndef GLUT_BITMAP_HELVETICA_12
#define GLUT_BITMAP_HELVETICA_12 (void*)5
#endif
#ifndef GLUT_BITMAP_HELVETICA_14
#define GLUT_BITMAP_HELVETICA_14 (void*)6
#endif
#ifndef GLUT_BITMAP_HELVETICA_18
#define GLUT_BITMAP_HELVETICA_18 (void*)7
#endif

// Utility macro
#define min(a,b) ((a) < (b) ? (a) : (b))

// Direction vectors
const int dx[4] = { 0, 1, 0, -1 }, dy[4] = { -1, 0, 1, 0 };
const int dx_path[8] = { 0, 1, 0, -1, 1, 1, -1, -1 }, dy_path[8] = { -1, 0, 1, 0, -1, 1,
1, -1 };

// Enums
typedef enum { GAME_MENU, GAME_PLAYING, GAME_WIN, GAME_LOSE } GameState;
typedef enum { DIFFICULTY_EASY, DIFFICULTY_MEDIUM, DIFFICULTY_HARD } DifficultyLevel;
typedef enum { THEME_DARK, THEME_LIGHT } ThemeMode;
typedef enum { MENU_EASY, MENU_MEDIUM, MENU_HARD, MENU_THEME, MENU_START, MENU_EXIT,
MENU_COUNT } MenuOption;

// Structures
typedef struct { float x, y; float width, height; bool isOn; } ToggleSwitch;
typedef struct {
    float bgR, bgG, bgB; float gridR, gridG, gridB; float textR, textG, textB;
    float uiR, uiG, uiB; float accentR, accentG, accentB;
} ThemeColors;
typedef struct { int x, y; } Point;
typedef struct { Point pos; int parent; float g, h, f; } Node;
typedef struct { float x, y; float light; int coinsCollected; } Player;
typedef struct { float x, y; float intensity; } TrailPoint;
typedef struct { float x, y; bool active; } Coin;
typedef struct { float x, y; float brightness; float size; } Star;
typedef struct { float x, y; float radius; float r, g, b, a; float pulse_speed; }
Nebula;
typedef struct {
    float x, y; float vx, vy; float size; float alpha; float color[3];
    float lifespan; float age;
} Particle;

// Global variables
int windowWidth = GRID_WIDTH * CELL_SIZE, windowHeight = GRID_HEIGHT * CELL_SIZE,
spaceMap[GRID_HEIGHT][GRID_WIDTH];
int gameTime = 0, timeLimit = 180, bestScores[3] = { -1, -1, -1 }, totalCoins = 0,
trailLength = 0;
float exitX, exitY, lightDecayRate;
bool pathExists = false;

GameState currentState = GAME_MENU;
DifficultyLevel currentDifficulty = DIFFICULTY_MEDIUM;
ThemeMode currentTheme = THEME_DARK;
MenuOption selectedOption = MENU_START;

Player player = { 1.5f, 1.5f, MAX_LIGHT_DURATION, 0 };
ToggleSwitch themeSwitch = { 0, 0, 60.0f, 30.0f, false };
TrailPoint trail[MAX_TRAIL_LENGTH];
Coin coins[MAX_COINS];
Star stars[MAX_STARS];
Nebula nebulas[MAX_NEBULAS];
Particle particles[MAX_PARTICLES];

// Theme colors
ThemeColors darkTheme = {
    0.05f, 0.06f, 0.12f,     // Background
    0.3f, 0.3f, 0.4f,        // Grid
    0.8f, 0.8f, 1.0f,        // Text
    0.2f, 0.2f, 0.3f,        // UI
    0.3f, 0.5f, 0.9f         // Accent
}, lightTheme = {
    0.92f, 0.85f, 0.75f,     // Background: Soft amber-gold (sunrise sky)
    0.70f, 0.60f, 0.75f,     // Grid: Lavender-amber (morning light on nebulae)
    0.40f, 0.25f, 0.35f,     // Text: Deep plum (dawn shadow text)
    0.85f, 0.75f, 0.65f,     // UI: Warm tan (dawn clouds)
    1.0f, 0.70f, 0.40f       // Accent: Bright amber-gold (morning star)
}, currentColors;

// Function prototypes
void init(void); void display(void); void reshape(int w, int h);
void keyboard(unsigned char key, int x, int y); void specialKeys(int key, int x, int y);
void update(int value); void updateTimer(int value); void renderGame(void);
```

```c
void renderMenu(void); void addTrailPoint(float x, float y); bool isValidMove(float x,
float y);
void checkCoinCollision(void); void checkWinCondition(void);
void generateEnvironment(bool guaranteePath); bool verifyAllPathsExist(void);
void startNewGame(void); void updateDifficultySettings(void);
void saveLoadBestScore(bool save); float heuristic(int x1, int y1, int x2, int y2);
bool pathfindAStar(int startX, int startY, int goalX, int goalY);
void generateRandomMap(void); void findValidExit(void); void placeCoins(void);
void createGuaranteedPath(void);

// Helper functions
float heuristic(int x1, int y1, int x2, int y2) {
    return (float)(abs(x1 - x2) + abs(y1 - y2));
}

int compareNodes(const void* a, const void* b) {
    Node* nodeA = (Node*)a, * nodeB = (Node*)b;
    return nodeA->f < nodeB->f ? -1 : (nodeA->f > nodeB->f ? 1 : 0);
}

void updateThemeColors(void) {
    currentColors = currentTheme == THEME_DARK ? darkTheme : lightTheme;
    glClearColor(currentColors.bgR, currentColors.bgG, currentColors.bgB, 1.0f);
}

// Initialization functions
void initGameObjects(void) {
    // Initialize stars
    for (int i = 0; i < MAX_STARS; i++) {
        stars[i].x = (float)(rand() % windowWidth);
        stars[i].y = (float)(rand() % windowHeight);
        stars[i].brightness = 0.3f + ((float)rand() / RAND_MAX) * 0.7f;
        stars[i].size = 1.0f + ((float)rand() / RAND_MAX) * 2.0f;
    }
    // Initialize nebulas
    for (int i = 0; i < MAX_NEBULAS; i++) {
        nebulas[i].x = (float)(rand() % windowWidth);
        nebulas[i].y = (float)(rand() % windowHeight);
        nebulas[i].radius = 100.0f + (rand() % 200);
        // Color palette
        int colorType = rand() % 4;
        switch (colorType) {
        case 0: // Purple
            nebulas[i].r = 0.3f + ((float)rand() / RAND_MAX) * 0.2f;
            nebulas[i].g = 0.1f + ((float)rand() / RAND_MAX) * 0.1f;
            nebulas[i].b = 0.4f + ((float)rand() / RAND_MAX) * 0.3f; break;
        case 1: // Blue
            nebulas[i].r = 0.1f + ((float)rand() / RAND_MAX) * 0.1f;
            nebulas[i].g = 0.2f + ((float)rand() / RAND_MAX) * 0.2f;
            nebulas[i].b = 0.5f + ((float)rand() / RAND_MAX) * 0.3f; break;
        case 2: // Teal
            nebulas[i].r = 0.1f + ((float)rand() / RAND_MAX) * 0.1f;
            nebulas[i].g = 0.3f + ((float)rand() / RAND_MAX) * 0.2f;
            nebulas[i].b = 0.4f + ((float)rand() / RAND_MAX) * 0.2f; break;
        case 3: // Pink
            nebulas[i].r = 0.4f + ((float)rand() / RAND_MAX) * 0.2f;
            nebulas[i].g = 0.1f + ((float)rand() / RAND_MAX) * 0.1f;
            nebulas[i].b = 0.3f + ((float)rand() / RAND_MAX) * 0.2f; break;
        }
        nebulas[i].a = 0.05f + ((float)rand() / RAND_MAX) * 0.05f;
        nebulas[i].pulse_speed = 0.5f + ((float)rand() / RAND_MAX) * 1.5f;
    }
    // Initialize particles
    for (int i = 0; i < MAX_PARTICLES; i++) {
        particles[i].x = rand() % windowWidth;
        particles[i].y = rand() % windowHeight;
        particles[i].vx = (float)(rand() % 100 - 50) / 200.0f;
        particles[i].vy = (float)(rand() % 100 - 50) / 200.0f;
        particles[i].size = 1.0f + (rand() % 30) / 10.0f;
        particles[i].color[0] = 0.1f + (rand() % 30) / 100.0f;
        particles[i].color[1] = 0.2f + (rand() % 40) / 100.0f;
        particles[i].color[2] = 0.5f + (rand() % 50) / 100.0f;
        particles[i].alpha = 0.1f + (rand() % 40) / 100.0f;
        particles[i].lifespan = 50.0f + rand() % 100;
        particles[i].age = rand() % (int)particles[i].lifespan;
    }
}

void init(void) {
    updateThemeColors();
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    generateEnvironment(false);
    initGameObjects();
    player.x = 1.5f;
    player.y = 1.5f;
    player.light = MAX_LIGHT_DURATION;
    player.coinsCollected = 0;
    addTrailPoint(player.x, player.y);
    updateDifficultySettings();
    saveLoadBestScore(false); // Load scores
}

// Path finding and map generation
bool pathfindAStar(int startX, int startY, int goalX, int goalY) {
    // Validate inputs
    if (startX < 0 || startX >= GRID_WIDTH || startY < 0 || startY >= GRID_HEIGHT ||
        goalX < 0 || goalX >= GRID_WIDTH || goalY < 0 || goalY >= GRID_HEIGHT ||
        spaceMap[startY][startX] == 1 || spaceMap[goalY][goalX] == 1)
        return false;

    // A* algorithm
    Node openSet[GRID_WIDTH * GRID_HEIGHT];
    bool closedSet[GRID_HEIGHT][GRID_WIDTH] = { {false} };
    int openSetSize = 0;

    // Add start node
      Node startNode = { {startX, startY}, -1, 0.0f, heuristic(startX, startY, goalX,
goalY) };
    startNode.f = startNode.g + startNode.h;
    openSet[openSetSize++] = startNode;
    while (openSetSize > 0) {
        // Find node with lowest f score
        int currentIndex = 0;
        for (int i = 1; i < openSetSize; i++) {
            if (openSet[i].f < openSet[currentIndex].f) {
                currentIndex = i;
            }
        }

        Node current = openSet[currentIndex];

        // Remove from open set
        openSet[currentIndex] = openSet[--openSetSize];

        // Check if reached goal
        if (current.pos.x == goalX && current.pos.y == goalY) return true;

        closedSet[current.pos.y][current.pos.x] = true;

        // Check neighbors - all 8 directions for more natural paths
        for (int i = 0; i < 8; i++) {
            int nx = current.pos.x + dx_path[i], ny = current.pos.y + dy_path[i];

            if (nx < 0 || nx >= GRID_WIDTH || ny < 0 || ny >= GRID_HEIGHT ||
                spaceMap[ny][nx] == 1 || closedSet[ny][nx]) continue;

            // For diagonals, check if both adjacent cells are not blocked
            if (i >= 4) { // Diagonal directions
                 int ax1 = current.pos.x + dx_path[i - 4]; // Adjacent cells in cardinal
directions
                int ay1 = current.pos.y;
                int ax2 = current.pos.x;
                int ay2 = current.pos.y + dy_path[i - 4];
                if (spaceMap[ay1][ax1] == 1 || spaceMap[ay2][ax2] == 1) continue;
            }

            float g = current.g + (i < 4 ? 1.0f : 1.414f); // Diagonal moves cost more

            // Check if neighbor is in open set
            bool inOpenSet = false;
            int openIndex = -1;
            for (int j = 0; j < openSetSize; j++) {
                if (openSet[j].pos.x == nx && openSet[j].pos.y == ny) {
                    inOpenSet = true; openIndex = j; break;
                }
            }

            if (inOpenSet) {
                // Update if better path
                if (g < openSet[openIndex].g) {
                    openSet[openIndex].g = g;
                    openSet[openIndex].f = g + openSet[openIndex].h;
                          openSet[openIndex].parent = current.pos.y * GRID_WIDTH +
current.pos.x;
                }
            }
            else if (openSetSize < GRID_WIDTH * GRID_HEIGHT) {
                // Add new node
                 Node neighbor = { {nx, ny}, current.pos.y * GRID_WIDTH + current.pos.x,
g, heuristic(nx, ny, goalX, goalY) };
                neighbor.f = neighbor.g + neighbor.h;
                openSet[openSetSize++] = neighbor;
            }
        }
    }
    return false;
}

void generateRandomMap(void) {
    // Initialize all cells as safe
    memset(spaceMap, 0, sizeof(spaceMap));

    // Create asteroid fields based on difficulty
    int numAsteroidFields;
    switch (currentDifficulty) {
    case DIFFICULTY_EASY: numAsteroidFields = GRID_WIDTH * GRID_HEIGHT / 8; break;
    case DIFFICULTY_MEDIUM: numAsteroidFields = GRID_WIDTH * GRID_HEIGHT / 6; break;
    case DIFFICULTY_HARD: numAsteroidFields = GRID_WIDTH * GRID_HEIGHT / 4; break;
    default: numAsteroidFields = GRID_WIDTH * GRID_HEIGHT / 6;
    }

    // Create large asteroid clusters
    for (int i = 0; i < numAsteroidFields / 4; i++) {
        int centerX = 3 + rand() % (GRID_WIDTH - 6);
        int centerY = 3 + rand() % (GRID_HEIGHT - 6);
        int radius = 1 + rand() % 2;

        for (int y = centerY - radius; y <= centerY + radius; y++) {
            for (int x = centerX - radius; x <= centerX + radius; x++) {
                if (x >= 0 && x < GRID_WIDTH && y >= 0 && y < GRID_HEIGHT) {
                    // Don't block starting area or exit area
                     if (!(x <= 3 && y <= 3) && !(x >= GRID_WIDTH - 4 && y >= GRID_HEIGHT
- 4)) {
                        if (rand() % 100 < 60) spaceMap[y][x] = 1;
                    }
                }
            }
        }
    }

    // Add some scattered smaller asteroids
    for (int i = 0; i < numAsteroidFields * 3 / 4; i++) {
        int x = rand() % GRID_WIDTH, y = rand() % GRID_HEIGHT;
        // Don't block starting area or exit area
        if (!(x <= 3 && y <= 3) && !(x >= GRID_WIDTH - 4 && y >= GRID_HEIGHT - 4)) {
            spaceMap[y][x] = 1;
        }
    }

    // Ensure starting area is safe
    for (int y = 0; y <= 3; y++) {
        for (int x = 0; x <= 3; x++) {
            spaceMap[y][x] = 0;
        }
    }
}

void placeCoins(void) {
    // Adjust coin count based on difficulty
    switch (currentDifficulty) {
    case DIFFICULTY_EASY: totalCoins = MAX_COINS - 3; break;
    case DIFFICULTY_MEDIUM: totalCoins = MAX_COINS - 1; break;
    case DIFFICULTY_HARD: totalCoins = MAX_COINS; break;
    }

    // Reset coins
    for (int i = 0; i < MAX_COINS; i++) coins[i].active = false;

    // Try random placement
    int coinsPlaced = 0, attempts = 0;
    const int maxAttempts = 200;

    while (coinsPlaced < totalCoins && attempts < maxAttempts) {
        int x = rand() % GRID_WIDTH, y = rand() % GRID_HEIGHT;

        if (spaceMap[y][x] == 0) {
            float distFromStart = sqrt(pow(x - 1, 2) + pow(y - 1, 2));
            float distFromExit = sqrt(pow(x - (int)exitX, 2) + pow(y - (int)exitY, 2));

            if (distFromStart > 2 && distFromExit > 2) {
                    if (pathfindAStar(1, 1, x, y) && pathfindAStar(x, y, (int)exitX,
(int)exitY)) {
                    // Check distance from other coins
                    bool tooClose = false;
                    for (int j = 0; j < coinsPlaced; j++) {
                        if (coins[j].active) {
                                float dist = sqrt(pow(x - (int)coins[j].x, 2) + pow(y -
(int)coins[j].y, 2));
                            if (dist < 3) { tooClose = true; break; }
                        }
                    }

                    if (!tooClose) {
                        coins[coinsPlaced].x = x + 0.5f;
                        coins[coinsPlaced].y = y + 0.5f;
                        coins[coinsPlaced].active = true;
                        coinsPlaced++;
                    }
                }
            }
        }
        attempts++;
    }

    // If not all coins placed, try along valid paths
    if (coinsPlaced < totalCoins) {
        bool visited[GRID_HEIGHT][GRID_WIDTH] = { {false} };
            int queue[GRID_WIDTH * GRID_HEIGHT][3], path[GRID_WIDTH * GRID_HEIGHT][2],
pathLength = 0, queueFront = 0, queueBack = 0;

        // BFS to find path
           queue[queueBack][0] = 1; queue[queueBack][1] = 1; queue[queueBack][2] = -1;
queueBack++;
        visited[1][1] = true;

        bool foundPath = false;
        while (queueFront < queueBack && !foundPath) {
                    int x = queue[queueFront][0], y = queue[queueFront][1], parent =
queue[queueFront][2];
            queueFront++;

            if (x == (int)exitX && y == (int)exitY) {
                // Reconstruct path
                int current = queueFront - 1;
                while (current != -1) {
                    path[pathLength][0] = queue[current][0];
                    path[pathLength][1] = queue[current][1];
                    pathLength++; current = queue[current][2];
                }
                foundPath = true; break;
            }

            // Try all directions
            for (int i = 0; i < 4; i++) {
                int nx = x + dx[i], ny = y + dy[i];
                if (nx >= 0 && nx < GRID_WIDTH && ny >= 0 && ny < GRID_HEIGHT &&
                    spaceMap[ny][nx] == 0 && !visited[ny][nx]) {
                    queue[queueBack][0] = nx; queue[queueBack][1] = ny;
                    queue[queueBack][2] = queueFront - 1; queueBack++; visited[ny][nx] =
true;
                }
            }
        }

        // Use path to place remaining coins
        if (foundPath && pathLength > 0) {
            // Reverse path (start to exit)
            for (int i = 0; i < pathLength / 2; i++) {
                int tempX = path[i][0], tempY = path[i][1];
                path[i][0] = path[pathLength - i - 1][0]; path[i][1] = path[pathLength -
i - 1][1];
                    path[pathLength - i - 1][0] = tempX; path[pathLength - i - 1][1] =
tempY;
            }

            // Place coins evenly
            int coinsLeft = totalCoins - coinsPlaced;
            if (coinsLeft > 0 && pathLength > 4) {
                int interval = pathLength / (coinsLeft + 1);
                if (interval < 1) interval = 1;

                for (int i = 1; i <= coinsLeft && coinsPlaced < totalCoins; i++) {
                    int pathIndex = i * interval;
                    if (pathIndex < pathLength) {
                        int x = path[pathIndex][0], y = path[pathIndex][1];

                        // Check if spot is available
                        bool isFree = true;
                        for (int j = 0; j < coinsPlaced; j++) {
                                        if  (coins[j].active  &&  (int)coins[j].x  ==  x  &&
(int)coins[j].y == y) {
                                isFree = false; break;
                            }
                        }

                        if (isFree) {
                            coins[coinsPlaced].x = x + 0.5f;
                            coins[coinsPlaced].y = y + 0.5f;
                            coins[coinsPlaced].active = true;
                            coinsPlaced++;
                        }
                    }
                }
            }
        }
    }
    // Update actual count
    totalCoins = coinsPlaced;
}

void findValidExit(void) {
    int attempts = 0;
    const int maxAttempts = 100;

    // Set minimum distance based on difficulty
    float minDistance;
    switch (currentDifficulty) {
    case DIFFICULTY_EASY: minDistance = GRID_WIDTH / 3; break;
    case DIFFICULTY_MEDIUM: minDistance = GRID_WIDTH / 2.5; break;
    case DIFFICULTY_HARD: minDistance = GRID_WIDTH / 2; break;
    default: minDistance = GRID_WIDTH / 2.5;
    }

    // Try to place exit
    while (attempts < maxAttempts) {
        int x = GRID_WIDTH / 2 + rand() % (GRID_WIDTH / 2 - 2);
        int y = GRID_HEIGHT / 2 + rand() % (GRID_HEIGHT / 2 - 2);
        float distFromStart = sqrt(pow(x - 1, 2) + pow(y - 1, 2));

        if (spaceMap[y][x] == 0 && distFromStart > minDistance) {
            exitX = x + 0.5f; exitY = y + 0.5f; return;
        }
        attempts++;
    }

    // Fallback - try on the far side from start
    exitX = GRID_WIDTH - 3 + 0.5f; exitY = GRID_HEIGHT - 3 + 0.5f;

    // Make sure exit area is safe
    int exitGridX = (int)exitX, exitGridY = (int)exitY;
    for (int y = exitGridY - 1; y <= exitGridY + 1; y++) {
        for (int x = exitGridX - 1; x <= exitGridX + 1; x++) {
            if (x >= 0 && x < GRID_WIDTH && y >= 0 && y < GRID_HEIGHT) {
                spaceMap[y][x] = 0;
            }
        }
    }
}

void createGuaranteedPath(void) {
    // Clear the map
    memset(spaceMap, 0, sizeof(spaceMap));

    // Set exit position
    exitX = GRID_WIDTH - 3 + 0.5f; exitY = GRID_HEIGHT - 3 + 0.5f;

    // Create a path from start to exit
    int currentX = 1, currentY = 1, exitGridX = (int)exitX, exitGridY = (int)exitY;
    int pathPoints[MAX_COINS][2], pathLength = 0;

    // Add start point
        pathPoints[pathLength][0]  =  currentX;  pathPoints[pathLength][1]  =  currentY;
pathLength++;

    // Create path segments
        while  ((currentX  <  exitGridX  ||  currentY  <  exitGridY)  &&  pathLength  <
MAX_PATH_LENGTH - 1) {
        bool moveHorizontalFirst = (rand() % 2 == 0);

        if (moveHorizontalFirst) {
            if (currentX < exitGridX) currentX += 1 + rand() % 2;
            if (currentY < exitGridY) currentY += 1 + rand() % 2;
        }
        else {
            if (currentY < exitGridY) currentY += 1 + rand() % 2;
            if (currentX < exitGridX) currentX += 1 + rand() % 2;
        }

        // Keep in bounds
        currentX = min(currentX, GRID_WIDTH - 2);
        currentY = min(currentY, GRID_HEIGHT - 2);

        // Add point to path
           pathPoints[pathLength][0] = currentX; pathPoints[pathLength][1] = currentY;
pathLength++;

        // Add random obstacles near the path
        if (rand() % 3 == 0) {
            for (int y = currentY - 3; y <= currentY + 3; y++) {
                for (int x = currentX - 3; x <= currentX + 3; x++) {
                    if (x >= 0 && x < GRID_WIDTH && y >= 0 && y < GRID_HEIGHT) {
                        if (abs(x - currentX) > 1 || abs(y - currentY) > 1) {
                            if (rand() % 100 < 30) spaceMap[y][x] = 1;
                        }
                    }
                }
            }
        }
    }

    // Add exit point
       pathPoints[pathLength][0]  =  exitGridX;  pathPoints[pathLength][1]  =  exitGridY;
pathLength++;

    // Clear the path for passability
    for (int i = 0; i < pathLength; i++) {
        int x = pathPoints[i][0], y = pathPoints[i][1];
        for (int ny = y - 1; ny <= y + 1; ny++) {
            for (int nx = x - 1; nx <= x + 1; nx++) {
                if (nx >= 0 && nx < GRID_WIDTH && ny >= 0 && ny < GRID_HEIGHT) {
                    spaceMap[ny][nx] = 0;
                }
            }
        }
    }

    // Place coins along the path
    totalCoins = min(pathLength - 2, MAX_COINS);

    for (int i = 0; i < MAX_COINS; i++) coins[i].active = false;

    for (int i = 0; i < totalCoins; i++) {
        int pathIndex = 1 + i * (pathLength - 2) / totalCoins;
        if (pathIndex >= pathLength - 1) pathIndex = pathLength - 2;

        coins[i].x = pathPoints[pathIndex][0] + 0.5f;
        coins[i].y = pathPoints[pathIndex][1] + 0.5f;
        coins[i].active = true;
    }
    pathExists = true;
}

bool verifyAllPathsExist(void) {
    // Check if exit is reachable from start
    if (!pathfindAStar(1, 1, (int)exitX, (int)exitY)) return false;

    // Check if all coins are reachable
    for (int i = 0; i < totalCoins; i++) {
        if (coins[i].active) {
            int coinX = (int)coins[i].x, coinY = (int)coins[i].y;
                if  (!pathfindAStar(1,  1,  coinX,  coinY)  ||  !pathfindAStar(coinX,  coinY,
(int)exitX, (int)exitY)) {
                return false;
            }
        }
    }
    return true;
}
void generateEnvironment(bool guaranteePath) {
    if (guaranteePath) {
        createGuaranteedPath();
        return;
    }

    int attempts = 0;
    const int maxAttempts = 5;

    while (attempts < maxAttempts) {
        generateRandomMap();
        findValidExit();
        placeCoins();
        if (verifyAllPathsExist()) {
            pathExists = true;
            return;
        }
        attempts++;
    }
    // If all attempts failed, create a guaranteed path
    createGuaranteedPath();
}

// Game state functions
void updateDifficultySettings(void) {
    switch (currentDifficulty) {
    case DIFFICULTY_EASY:
        timeLimit = 60;
        lightDecayRate = LIGHT_DECAY_RATE * 2.5f; // Much slower light depletion
        break;
    case DIFFICULTY_MEDIUM:
        timeLimit = 45;
        lightDecayRate = LIGHT_DECAY_RATE * 3.0f; // Medium light depletion
        break;
    case DIFFICULTY_HARD:
        timeLimit = 30;
        lightDecayRate = LIGHT_DECAY_RATE * 3.5f; // Faster light depletion
        break;
    }
}

void startNewGame(void) {
    gameTime = 0;
    trailLength = 0;
    generateEnvironment(false);
    player.x = 1.5f;
    player.y = 1.5f;
    player.light = MAX_LIGHT_DURATION;
    player.coinsCollected = 0;
    addTrailPoint(player.x, player.y);
    currentState = GAME_PLAYING;
}

void saveLoadBestScore(bool save) {
    if (save) {
                        if   (bestScores[currentDifficulty]   ==   -1   ||   gameTime   <
bestScores[currentDifficulty]) {
            bestScores[currentDifficulty] = gameTime;
            FILE* file = NULL;
            fopen_s(&file, "cosmiclightweaver.dat", "wb");
            if (file) {
                const char header[] = "CLWSAV01";
                fwrite(header, sizeof(char), 8, file);
                fwrite(bestScores, sizeof(int), 3, file);
                fclose(file);
            }
        }
    }
    else { // Load scores
        for (int i = 0; i < 3; i++) bestScores[i] = -1;
        FILE* file = NULL;
        fopen_s(&file, "cosmiclightweaver.dat", "rb");
        if (file) {
            char header[8];
             if (fread(header, sizeof(char), 8, file) == 8 && memcmp(header, "CLWSAV01",
8) == 0) {
                fread(bestScores, sizeof(int), 3, file);
            }
            fclose(file);
        }
    }
}

// Game mechanics
void addTrailPoint(float x, float y) {
    if (trailLength >= MAX_TRAIL_LENGTH) {
        memmove(&trail[0], &trail[1], (MAX_TRAIL_LENGTH - 1) * sizeof(TrailPoint));
        trailLength--;
    }
     trail[trailLength].x = x; trail[trailLength].y = y; trail[trailLength].intensity =
5.0f;
    trailLength++;
}

bool isValidMove(float x, float y) {
    if (x < 0 || x >= GRID_WIDTH || y < 0 || y >= GRID_HEIGHT) return false;
    int gridX = (int)x, gridY = (int)y;
    return spaceMap[gridY][gridX] != 1;
}

void checkCoinCollision(void) {
    for (int i = 0; i < totalCoins; i++) {
        if (coins[i].active) {
            float dx = player.x - coins[i].x, dy = player.y - coins[i].y;
            float distance = sqrt(dx * dx + dy * dy);

            if (distance < 0.7f) {
                coins[i].active = false;
                player.coinsCollected++;

                // Energy boost based on difficulty
                float energyBoost;
                switch (currentDifficulty) {
                case DIFFICULTY_EASY: energyBoost = MAX_LIGHT_DURATION * 0.25f; break;
                case DIFFICULTY_MEDIUM: energyBoost = MAX_LIGHT_DURATION * 0.2f; break;
                case DIFFICULTY_HARD: energyBoost = MAX_LIGHT_DURATION * 0.15f; break;
                default: energyBoost = MAX_LIGHT_DURATION * 0.2f;
                }

                player.light += energyBoost;
                               if  (player.light  >  MAX_LIGHT_DURATION)  player.light  =
MAX_LIGHT_DURATION;
            }
        }
    }
}

void checkWinCondition(void) {
    float dx = player.x - exitX, dy = player.y - exitY;
    float distance = sqrt(dx * dx + dy * dy);
    if (distance < 0.7f && player.coinsCollected == totalCoins) {
        currentState = GAME_WIN;
        saveLoadBestScore(true);
    }
}

// Rendering functions
void renderBackgroundEffects(void) {
    float time = glutGet(GLUT_ELAPSED_TIME) * 0.001f;
    float centerX = windowWidth * 0.5f, centerY = windowHeight * 0.5f;

    // Background vortex
    for (int arm = 0; arm < 3; arm++) {
        float armOffset = 2.0f * M_PI * arm / 3.0f;
        glBegin(GL_LINE_STRIP);
        for (float t = 0; t < 15.0f; t += 0.1f) {
            float radius = 10.0f + t * 30.0f;
            float angle = t * 1.5f + time * (1.0f - t / 15.0f) + armOffset;
            float x = centerX + radius * cos(angle), y = centerY + radius * sin(angle);
            float alpha = 0.5f * (1.0f - t / 15.0f);
            // Color transitions
               float r = 0.2f + 0.3f * sin(t + time), g = 0.3f + 0.3f * sin(t + time +
2.0f);
            float b = 0.6f + 0.3f * sin(t + time + 4.0f);
            glColor4f(r, g, b, alpha); glVertex2f(x, y);
        }
        glEnd();
    }

    // Energy grid
    float gridSpacing = 70.0f, lineAlpha = 0.1f + 0.05f * sin(time * 0.5f);
    // Horizontal lines
    for (float y = 0; y < windowHeight; y += gridSpacing) {
        glBegin(GL_LINE_STRIP);
        for (float x = 0; x < windowWidth; x += 5) {
            float wave = 5.0f * sin(x * 0.02f + time * 1.5f);
            float alpha = lineAlpha * (0.5f + 0.5f * sin(x * 0.01f + time));
            glColor4f(0.2f, 0.5f, 0.8f, alpha); glVertex2f(x, y + wave);
        }
        glEnd();
    }
    // Vertical lines
    for (float x = 0; x < windowWidth; x += gridSpacing) {
        glBegin(GL_LINE_STRIP);
        for (float y = 0; y < windowHeight; y += 5) {
            float wave = 5.0f * sin(y * 0.02f + time * 1.2f + M_PI / 2);
            float alpha = lineAlpha * (0.5f + 0.5f * sin(y * 0.01f + time));
            glColor4f(0.3f, 0.4f, 0.9f, alpha); glVertex2f(x + wave, y);
        }
        glEnd();
    }
}

void renderStarsAndNebulas(void) {
    float time = glutGet(GLUT_ELAPSED_TIME) * 0.001f;
    float starAlphaMultiplier = (currentTheme == THEME_DARK) ? 1.0f : 0.6f;
    float nebulaAlphaMultiplier = (currentTheme == THEME_DARK) ? 1.0f : 0.4f;

    // Stars
    for (int i = 0; i < MAX_STARS; i++) {
        float brightness = stars[i].brightness * starAlphaMultiplier;
        glColor4f(brightness, brightness, brightness * 1.2f, brightness);
        glPointSize(stars[i].size * (currentTheme == THEME_DARK ? 1.0f : 0.8f));
        glBegin(GL_POINTS); glVertex2f(stars[i].x, stars[i].y); glEnd();

        // Glow for bright stars
        if (stars[i].brightness > 0.8f) {
                  glColor4f(brightness  *  0.8f,  brightness  *  0.8f,  brightness,  0.3f  *
starAlphaMultiplier);
            glBegin(GL_TRIANGLE_FAN);
            glVertex2f(stars[i].x, stars[i].y);
            for (int j = 0; j <= 8; j++) {
                float angle = 2.0f * M_PI * j / 8;
                glVertex2f(stars[i].x + cos(angle) * stars[i].size * 2.0f,
                    stars[i].y + sin(angle) * stars[i].size * 2.0f);
            }
            glEnd();
        }
    }

    // Nebulas
    for (int i = 0; i < MAX_NEBULAS; i++) {
        float pulse = 1.0f + 0.1f * sin(time * nebulas[i].pulse_speed);
        float radius = nebulas[i].radius * pulse;
        float alpha = nebulas[i].a * nebulaAlphaMultiplier;

        // Draw layers with gradient
        for (int j = 0; j < 5; j++) {
            float layerAlpha = alpha * (1.0f - j * 0.2f);
            float size = radius * (1.0f - j * 0.15f);
            // Adjust colors for theme
            float r = nebulas[i].r, g = nebulas[i].g, b = nebulas[i].b;
            if (currentTheme == THEME_LIGHT) {
                r = 0.7f + (r * 0.3f); g = 0.7f + (g * 0.3f); b = 0.8f + (b * 0.2f);
            }
            glColor4f(r, g, b, layerAlpha);
            glBegin(GL_TRIANGLE_FAN);
            glVertex2f(nebulas[i].x, nebulas[i].y);
            for (int k = 0; k <= 20; k++) {
                float angle = 2.0f * M_PI * k / 20;
                float distortion = 1.0f + 0.2f * sin(angle * 5 + time);
                glVertex2f(nebulas[i].x + cos(angle) * size * distortion,
                    nebulas[i].y + sin(angle) * size * distortion);
            }
            glEnd();
        }
    }
}

void renderParticles(void) {
    float time = glutGet(GLUT_ELAPSED_TIME) * 0.001f;
    for (int i = 0; i < MAX_PARTICLES; i++) {
        // Calculate fade
        float fade = 1.0f;
        if (particles[i].age < 10) fade = particles[i].age / 10.0f;
                 else  if  (particles[i].age  >  particles[i].lifespan  -  10)  fade  =
(particles[i].lifespan - particles[i].age) / 10.0f;
        // Pulse size
        float sizeMultiplier = 0.8f + 0.2f * sin(time * 2.0f + i * 0.1f);
        // Draw glow
        glPointSize(particles[i].size * sizeMultiplier * 3.0f);
        glColor4f(particles[i].color[0], particles[i].color[1],
            particles[i].color[2], particles[i].alpha * 0.2f * fade);
        glBegin(GL_POINTS); glVertex2f(particles[i].x, particles[i].y); glEnd();
        // Draw core
        glPointSize(particles[i].size * sizeMultiplier);
        glColor4f(particles[i].color[0] + 0.2f, particles[i].color[1] + 0.2f,
            particles[i].color[2] + 0.2f, particles[i].alpha * fade);
        glBegin(GL_POINTS); glVertex2f(particles[i].x, particles[i].y); glEnd();
    }
    glPointSize(1.0f);
}

void renderSpace(void) {
    float time = glutGet(GLUT_ELAPSED_TIME) * 0.001f;
    for (int y = 0; y < GRID_HEIGHT; y++) {
        for (int x = 0; x < GRID_WIDTH; x++) {
            if (spaceMap[y][x] == 1) {  // Asteroid field
                float offsetX = x * CELL_SIZE + CELL_SIZE / 2;
                float offsetY = y * CELL_SIZE + CELL_SIZE / 2;
                // Draw multiple asteroids per cell
                for (int i = 0; i < 3; i++) {
                    float seedX = (float)(x * 10 + y * 7 + i * 3);
                    float seedY = (float)(y * 10 + x * 3 + i * 7);
                    float asteroidX = offsetX + sin(seedX) * CELL_SIZE * 0.3f;
                    float asteroidY = offsetY + cos(seedY) * CELL_SIZE * 0.3f;
                    float size = (0.2f + 0.1f * sin(time + seedX)) * CELL_SIZE;
                    // Theme-appropriate colors
                    float r, g, b;
                    if (currentTheme == THEME_DARK) {
                         r = 0.3f + 0.05f * sin(seedX); g = 0.25f + 0.05f * sin(seedY); b
= 0.35f;
                    }
                    else {
                        r = 0.5f + 0.05f * sin(seedX); g = 0.45f + 0.05f * sin(seedY); b
= 0.4f;
                    }
                    // Draw asteroid body
                    glColor3f(r, g, b);
                    glBegin(GL_TRIANGLE_FAN);
                    glVertex2f(asteroidX, asteroidY);
                    for (int j = 0; j <= 8; j++) {
                        float angle = 2.0f * M_PI * j / 8;
                        float irregularity = 0.7f + 0.3f * sin(angle * 3 + seedY);
                        glVertex2f(asteroidX + cos(angle) * size * irregularity,
                            asteroidY + sin(angle) * size * irregularity);
                    }
                    glEnd();
                    // Asteroid highlights
                     glColor3f((currentTheme == THEME_DARK) ? 0.4f + 0.1f * sin(seedX) :
0.6f + 0.1f * sin(seedX),
                        (currentTheme == THEME_DARK) ? 0.3f : 0.55f,
                        (currentTheme == THEME_DARK) ? 0.45f : 0.5f);
                    glBegin(GL_LINE_LOOP);
                    for (int j = 0; j <= 8; j++) {
                        float angle = 2.0f * M_PI * j / 8;
                        float irregularity = 0.7f + 0.3f * sin(angle * 3 + seedY);
                        glVertex2f(asteroidX + cos(angle) * size * irregularity,
                            asteroidY + sin(angle) * size * irregularity);
                    }
                    glEnd();
                    // Subtle glow
                    float glowAlpha = (currentTheme == THEME_DARK) ? 0.1f : 0.05f;
                    if (i == 0 && rand() % 4 == 0) {
                        glColor4f((currentTheme == THEME_DARK) ? 0.3f : 0.5f,
                            (currentTheme == THEME_DARK) ? 0.15f : 0.4f,
                            (currentTheme == THEME_DARK) ? 0.4f : 0.3f, glowAlpha);
                        glBegin(GL_TRIANGLE_FAN);
                        glVertex2f(asteroidX, asteroidY);
                        for (int j = 0; j <= 12; j++) {
                            float angle = 2.0f * M_PI * j / 12;
                            float irregularity = 0.9f + 0.1f * sin(angle * 2 + time);
                                     glVertex2f(asteroidX + cos(angle) * size * 1.8f *
irregularity,
                                asteroidY + sin(angle) * size * 1.8f * irregularity);
                        }
                        glEnd();
                    }
                }
            }
        }
    }
}

// Updated rocket to be 30% larger than the smaller version (but still smaller than
original)
void renderPlayer(void) {
    float time = glutGet(GLUT_ELAPSED_TIME) * 0.005f;
    float radius = CELL_SIZE * 0.273f; // Increased by 30% from 0.21f
    float pulse = 0.7f + 0.3f * sin(time);
    float lightRatio = player.light / MAX_LIGHT_DURATION;

    // Calculate rocket orientation based on movement
    float angle = 0;
    if (trailLength >= 2) {
        float dx = player.x - trail[trailLength - 2].x;
        float dy = player.y - trail[trailLength - 2].y;
        if (dx != 0 || dy != 0) angle = atan2(dy, dx);
    }

    // Draw rocket with rotation
    glPushMatrix();
    glTranslatef(player.x * CELL_SIZE, player.y * CELL_SIZE, 0);
    glRotatef(angle * 180 / M_PI, 0, 0, 1);

    // Rocket body - more detailed, with new size
    // Nose cone (red-orange with highlight)
    glBegin(GL_TRIANGLES);
    glColor4f(0.9f, 0.4f, 0.2f, 0.9f * lightRatio);
    glVertex2f(radius * 1.6f, 0);
    glVertex2f(radius * 0.6f, radius * 0.45f);
    glVertex2f(radius * 0.6f, -radius * 0.45f);
    glEnd();

    // Nose cone highlight
    glBegin(GL_TRIANGLES);
    glColor4f(1.0f, 0.7f, 0.5f, 0.9f * lightRatio);
    glVertex2f(radius * 1.6f, 0);
    glVertex2f(radius * 0.6f, radius * 0.15f);
    glVertex2f(radius * 0.6f, -radius * 0.15f);
    glEnd();

    // Main body (silver with shadow)
    glBegin(GL_QUADS);
    glColor4f(0.9f, 0.9f, 0.95f, 0.9f * lightRatio);
    glVertex2f(radius * 0.6f, radius * 0.45f);
    glVertex2f(radius * 0.6f, -radius * 0.45f);
    glVertex2f(-radius * 1.0f, -radius * 0.45f);
    glVertex2f(-radius * 1.0f, radius * 0.45f);
    glEnd();

    // Body shadow/detail
    glBegin(GL_QUADS);
    glColor4f(0.7f, 0.7f, 0.75f, 0.9f * lightRatio);
    glVertex2f(radius * 0.6f, -radius * 0.15f);
    glVertex2f(-radius * 1.0f, -radius * 0.15f);
    glVertex2f(-radius * 1.0f, -radius * 0.45f);
    glVertex2f(radius * 0.6f, -radius * 0.45f);
    glEnd();

    // Body stripes/details
    glBegin(GL_QUADS);
    glColor4f(0.3f, 0.6f, 0.8f, 0.9f * lightRatio);
    // Top stripe
    glVertex2f(radius * 0.4f, radius * 0.45f);
    glVertex2f(radius * 0.2f, radius * 0.45f);
```

```
glVertex2f(radius * 0.2f, -radius * 0.45f);
glVertex2f(radius * 0.4f, -radius * 0.45f);
// Middle stripe
glVertex2f(-radius * 0.2f, radius * 0.45f);
glVertex2f(-radius * 0.4f, radius * 0.45f);
glVertex2f(-radius * 0.4f, -radius * 0.45f);
glVertex2f(-radius * 0.2f, -radius * 0.45f);
glEnd();

// Fins (blue with highlights)
glBegin(GL_TRIANGLES);
// Top fin
glColor4f(0.2f, 0.4f, 0.9f, 0.9f * lightRatio);
glVertex2f(-radius * 0.7f, radius * 0.45f);
glVertex2f(-radius * 1.2f, radius * 0.9f);
glVertex2f(-radius * 1.0f, radius * 0.45f);

// Top fin highlight
glColor4f(0.4f, 0.6f, 1.0f, 0.9f * lightRatio);
glVertex2f(-radius * 0.75f, radius * 0.45f);
glVertex2f(-radius * 1.15f, radius * 0.8f);
glVertex2f(-radius * 0.95f, radius * 0.45f);

// Bottom fin
glColor4f(0.2f, 0.4f, 0.9f, 0.9f * lightRatio);
glVertex2f(-radius * 0.7f, -radius * 0.45f);
glVertex2f(-radius * 1.2f, -radius * 0.9f);
glVertex2f(-radius * 1.0f, -radius * 0.45f);

// Bottom fin highlight
glColor4f(0.4f, 0.6f, 1.0f, 0.9f * lightRatio);
glVertex2f(-radius * 0.75f, -radius * 0.45f);
glVertex2f(-radius * 1.15f, -radius * 0.8f);
glVertex2f(-radius * 0.95f, -radius * 0.45f);
glEnd();

// Windows/porthole (brighter blue)
glColor4f(0.4f, 0.8f, 1.0f, 0.9f * lightRatio);
glBegin(GL_TRIANGLE_FAN);
float windowX = radius * 0.2f;
float windowY = 0;
float windowSize = radius * 0.22f;
glVertex2f(windowX, windowY);
for (int i = 0; i <= 16; i++) {
    float a = 2.0f * M_PI * i / 16;
    glVertex2f(windowX + cos(a) * windowSize, windowY + sin(a) * windowSize);
}
glEnd();

// Window highlight/reflection
glColor4f(0.8f, 0.9f, 1.0f, 0.7f * lightRatio);
glBegin(GL_TRIANGLE_FAN);
glVertex2f(windowX - windowSize * 0.3f, windowY - windowSize * 0.3f);
for (int i = 0; i <= 8; i++) {
    float a = 2.0f * M_PI * i / 16;
    glVertex2f(windowX - windowSize * 0.3f + cos(a) * windowSize * 0.4f,
            windowY - windowSize * 0.3f + sin(a) * windowSize * 0.4f);
    }
    glEnd();

    // Engine exhaust/smoke - varies with energy level
    float exhaustScale = lightRatio * pulse;

    // Main engine fire (multiple layers for depth)
    for (int layer = 0; layer < 3; layer++) {
        float layerAlpha = (0.8f - layer * 0.2f) * exhaustScale;
        float layerLength = (1.8f - layer * 0.3f) * radius * exhaustScale;

        // Color gradient from white->yellow->orange->red based on energy
        float r = 1.0f;
        float g = 0.3f + lightRatio * 0.7f;
        float b = (lightRatio > 0.7f) ? 0.5f * lightRatio : 0.0f;

        glColor4f(r, g, b, layerAlpha);
        glBegin(GL_TRIANGLE_FAN);
        glVertex2f(-radius * 1.0f, 0);

        for (int i = 0; i <= 16; i++) {
            float a = M_PI * (i / 16.0f + 0.5f); // Half circle for exhaust
             float flicker = 1.0f + 0.4f * sin(time * 20.0f + i * 0.7f); // More dynamic
flame flicker
            float exhaustWidth = (0.4f - layer * 0.1f) * radius * flicker;
            glVertex2f(-radius * 1.0f - cos(a) * layerLength, sin(a) * exhaustWidth);
        }
        glEnd();
    }

    // Smoke particles from exhaust (only visible with enough energy)
    if (lightRatio > 0.2f) {
        glPointSize(3.5f); // Adjusted for increased rocket size
        glBegin(GL_POINTS);
        for (int i = 0; i < 8; i++) {
             float smokeX = -radius * (1.5f + (i * 0.5f)) + sin(time * 5.0f + i) * radius
* 0.1f;
            float smokeY = sin(time * 8.0f + i * 2.0f) * radius * 0.25f;
            float smokeAlpha = (0.7f - i * 0.09f) * lightRatio;

            // Smoke gradient from orange to gray
            float smokeVal = 0.3f + i * 0.08f;
            float r = smokeVal + (i < 2 ? 0.3f : 0.0f); // More orange near engine
            float g = smokeVal * 0.9f;
            float b = smokeVal * 0.8f;

            glColor4f(r, g, b, smokeAlpha);
            glVertex2f(smokeX, smokeY);
        }
        glEnd();
        glPointSize(1.0f);
    }

    glPopMatrix();
}
```

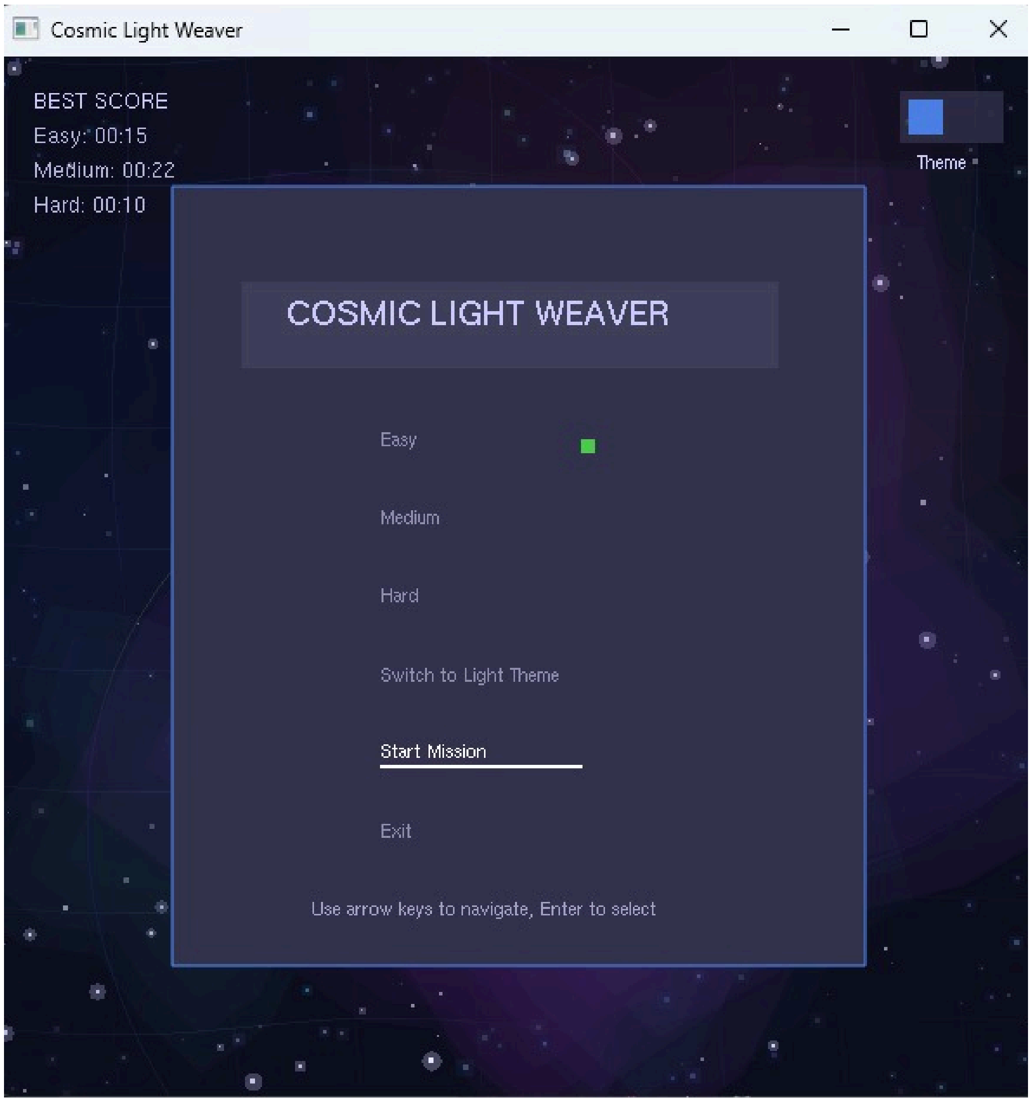
```
void renderTrail(void) {
    float time = glutGet(GLUT_ELAPSED_TIME) * 0.01f;
    for (int i = 0; i < trailLength; i++) {
        float alpha = trail[i].intensity / 5.0f;
        if (alpha <= 0) continue;

        // Smoke gets larger and more transparent the older it is
        float ageRatio = (float)i / trailLength;
        float size = CELL_SIZE * (0.15f + ageRatio * 0.2f);
        float trailX = trail[i].x * CELL_SIZE, trailY = trail[i].y * CELL_SIZE;

        // Smoke color changes from orange to gray as it ages
        float smoke = 0.35f + ageRatio * 0.45f;
        float r = smoke + (1.0f - ageRatio) * 0.3f; // More orange when fresh
        float g = smoke * 0.9f;
        float b = smoke * 0.7f;

        // Smoke puffs with slight pulse and more random shape
        float pulse = 0.8f + 0.2f * sin(time + i * 0.2f);

        glColor4f(r, g, b, alpha * (0.8f - ageRatio * 0.6f));
        glBegin(GL_TRIANGLE_FAN);
        glVertex2f(trailX, trailY);
        for (int j = 0; j <= 16; j++) {
            float angle = 2.0f * M_PI * j / 16;
            float wobble = 1.0f + 0.4f * sin(angle * 4 + time + i * 0.3f);
            glVertex2f(trailX + cos(angle) * size * pulse * wobble,
                trailY + sin(angle) * size * pulse * wobble);
        }
        glEnd();
    }
}

// Updated coins to look like lightning/electricity bolts ⚡
void renderCoins(void) {
    float time = glutGet(GLUT_ELAPSED_TIME) * 0.001f;
    for (int i = 0; i < totalCoins; i++) {
        if (!coins[i].active) continue;
        float x = coins[i].x * CELL_SIZE, y = coins[i].y * CELL_SIZE;
        float rotation = time * 1.5f + i * 0.5f;
        float pulse = 0.8f + 0.2f * sin(time * 3.0f + i);
        float size = CELL_SIZE * 0.35f * pulse;

        // Draw warning triangle background (for standard high voltage symbol)
        // (optional - uncomment if you want triangular background)
        /*
        // Triangle with rounded corners and glow
        glColor4f(0.9f, 0.8f, 0.0f, 0.2f + 0.1f * sin(time * 2.0f + i));
        glBegin(GL_TRIANGLE_FAN);
        glVertex2f(x, y);
        // Triangle with slight rotation
        for (int j = 0; j <= 3; j++) {
            float angle = 2.0f * M_PI * j / 3.0f + rotation * 0.1f;
            float dist = size * 2.0f;
            glVertex2f(x + cos(angle) * dist, y + sin(angle) * dist);
        }
        glEnd();
        */

        // Electric field glow (outer)
        glColor4f(0.3f, 0.6f, 1.0f, 0.2f + 0.1f * sin(time * 2.0f + i));
        glBegin(GL_TRIANGLE_FAN);
        glVertex2f(x, y);
        for (int j = 0; j <= 20; j++) {
            float angle = 2.0f * M_PI * j / 20 + rotation * 0.1f;
            float wobble = 1.0f + 0.2f * sin(angle * 4 + time * 3.0f);
            glVertex2f(x + cos(angle) * size * 2.0f * wobble,
                y + sin(angle) * size * 2.0f * wobble);
        }
        glEnd();

        // Draw the standard electricity/high voltage warning bolt
        // Two layers - outer glow and inner bright core
        for (int layer = 0; layer < 2; layer++) {
            // Outer glow is blue, inner core is bright white-blue
            if (layer == 0) {
                    glColor4f(0.4f, 0.6f, 1.0f, (0.8f + 0.2f * sin(time * 5.0f + i)) *
pulse);
            }
            else {
                    glColor4f(0.9f, 0.95f, 1.0f, (0.9f + 0.1f * sin(time * 8.0f + i)) *
pulse);
            }

            float boltSize = size * (layer == 0 ? 1.1f : 0.9f);

            // Draw the classic down-pointing lightning bolt
            glBegin(GL_TRIANGLE_STRIP);

            // Top of the bolt
            glVertex2f(x - boltSize * 0.2f, y - boltSize * 1.1f);
            glVertex2f(x + boltSize * 0.2f, y - boltSize * 1.1f);

            // First zag to right
            glVertex2f(x, y - boltSize * 0.5f);
            glVertex2f(x + boltSize * 0.4f, y - boltSize * 0.5f);

            // Second zag to left
            glVertex2f(x, y + boltSize * 0.1f);
            glVertex2f(x - boltSize * 0.4f, y + boltSize * 0.1f);

            // Third zag to bottom point
            glVertex2f(x - boltSize * 0.2f, y + boltSize * 1.1f);
            glVertex2f(x + boltSize * 0.2f, y + boltSize * 1.1f);
            glEnd();
        }

        // Add electric spark particles around the bolt
        glPointSize(3.0f);
        glBegin(GL_POINTS);
        for (int j = 0; j < 12; j++) {
            // Random but consistent spark positions
            float sparkAngle = j * M_PI / 6.0f + time * (1.0f + i * 0.1f);
            float sparkDist = size * (1.0f + 0.5f * sin(j * 0.5f + time * 3.0f));
            float sparkX = x + cos(sparkAngle) * sparkDist;
            float sparkY = y + sin(sparkAngle) * sparkDist;

            // Dynamic brightness
            float brightness = 0.7f + 0.3f * sin(time * 10.0f + j);

            // Color gradient from white to blue
            float blueRatio = 0.5f + 0.5f * sin(j * 0.7f + time * 2.0f);
            glColor4f(0.7f + 0.3f * (1.0f - blueRatio),
                0.8f + 0.2f * (1.0f - blueRatio),
                1.0f,
                brightness);
            glVertex2f(sparkX, sparkY);
        }
        glEnd();

        // Add electric arcs connecting to sparks
        glLineWidth(1.5f);
        glBegin(GL_LINES);
        for (int j = 0; j < 8; j++) {
            float arcAngle1 = j * M_PI / 4.0f + time * 2.0f;
            float arcAngle2 = j * M_PI / 4.0f + 0.2f + time * 2.0f;

            float arcX1 = x + cos(arcAngle1) * size * 0.7f;
            float arcY1 = y + sin(arcAngle1) * size * 0.7f;
            float arcX2 = x + cos(arcAngle2) * size * 1.6f;
            float arcY2 = y + sin(arcAngle2) * size * 1.6f;

            float alpha = 0.6f + 0.4f * sin(time * 8.0f + j);
            glColor4f(0.4f, 0.7f, 1.0f, alpha);
            glVertex2f(arcX1, arcY1);
            glVertex2f(arcX2, arcY2);
        }
        glEnd();
        glLineWidth(1.0f);

        // Occasional energy burst (only on some coins and at random intervals)
        if (i % 3 == 0 && (int)(time * 3.0f) % 2 == 0) {
            glColor4f(0.5f, 0.8f, 1.0f, 0.3f * pulse);
            glBegin(GL_TRIANGLE_FAN);
            glVertex2f(x, y);
            for (int j = 0; j <= 16; j++) {
                float burstAngle = 2.0f * M_PI * j / 16;
                 float burstDist = size * 2.5f * (1.0f + 0.3f * sin(burstAngle * 5 + time
* 7.0f));
                      glVertex2f(x + cos(burstAngle) * burstDist, y + sin(burstAngle) *
burstDist);
            }
            glEnd();
        }
    }
}
```

```
void renderExit(void) {
    float radius = CELL_SIZE * 0.6f;
    float time = glutGet(GLUT_ELAPSED_TIME) * 0.001f;
    float rotation = time * 2.0f;
    float pulse = 1.0f + 0.1f * sin(time * 3.0f);
    float exitPosX = exitX * CELL_SIZE, exitPosY = exitY * CELL_SIZE;

    // Outer event horizon layers
    for (int i = 0; i < 5; i++) {
        float alpha = 0.15f - i * 0.02f;
        float size = (1.2f + i * 0.4f) * pulse;
        float hue = i / 5.0f;
        float rotDir = (i % 2 == 0) ? 1 : -1;
         glColor4f(0.2f + 0.2f * sin(hue * M_PI + time), 0.0f + 0.2f * sin(hue * M_PI *
2),
            0.4f - 0.1f * hue, alpha);
        glBegin(GL_TRIANGLE_FAN);
        glVertex2f(exitPosX, exitPosY);
        for (int j = 0; j <= 30; j++) {
            float angle = 2.0f * M_PI * j / 30 + rotation * rotDir;
            float wobble = 1.0f + 0.2f * sin(angle * 6 + time * 4);
            glVertex2f(exitPosX + cos(angle) * radius * size * wobble,
                exitPosY + sin(angle) * radius * size * wobble);
        }
        glEnd();
    }

    // Inner event horizon
    glColor4f(0.4f, 0.0f, 0.6f, 0.5f);
    glBegin(GL_TRIANGLE_FAN);
    glVertex2f(exitPosX, exitPosY);
    for (int i = 0; i <= 20; i++) {
        float angle = 2.0f * M_PI * i / 20 - rotation;
        float wobble = 1.0f + 0.15f * sin(angle * 4 + time * 5);
        glVertex2f(exitPosX + cos(angle) * radius * 0.8f * pulse * wobble,
            exitPosY + sin(angle) * radius * 0.8f * pulse * wobble);
    }
    glEnd();
    // Central singularity
    glColor4f(0.0f, 0.0f, 0.0f, 0.95f);
    glBegin(GL_TRIANGLE_FAN);
    glVertex2f(exitPosX, exitPosY);
    for (int i = 0; i <= 20; i++) {
        float angle = 2.0f * M_PI * i / 20;
        glVertex2f(exitPosX + cos(angle) * radius * 0.5f * pulse,
            exitPosY + sin(angle) * radius * 0.5f * pulse);
    }
    glEnd();

    // Accretion disk
    for (int s = 0; s < 3; s++) {
        float spiralOffset = s * 2.0f * M_PI / 3.0f;
        float brightness = 0.7f + 0.3f * sin(time * 2.0f + s);

        glBegin(GL_LINE_STRIP);
        for (int i = 0; i <= 100; i++) {
            float t = i / 100.0f * 8.0f * M_PI;
            float r = 0.2f + 0.6f * t / (8.0f * M_PI);
            float colorPos = i / 100.0f;
            float alpha = brightness * (1.0f - colorPos * 0.7f);

            // Color based on spiral arm
            switch (s) {
                case 0: glColor4f(0.7f - 0.4f * colorPos, 0.1f + 0.3f * colorPos, 0.9f,
alpha); break;
                case 1: glColor4f(0.2f + 0.5f * colorPos, 0.0f + 0.3f * colorPos, 0.8f -
0.3f * colorPos, alpha); break;
                case 2: glColor4f(0.7f - 0.3f * colorPos, 0.2f * colorPos, 0.5f + 0.3f *
colorPos, alpha); break;
            }
            glVertex2f(exitPosX + cos(t + rotation + spiralOffset) * radius * r,
                exitPosY + sin(t + rotation + spiralOffset) * radius * r);
        }
        glEnd();
    }

    // Sparkles
    glPointSize(2.0f);
    glBegin(GL_POINTS);
    for (int i = 0; i < 30; i++) {
        float angle = (rand() % 628) / 100.0f;
        float dist = (0.9f + 0.6f * (rand() % 100) / 100.0f) * radius;
        float brightness = 0.5f + 0.5f * sin(time * 5.0f + i * 0.5f);
        switch (i % 3) {
        case 0: glColor4f(0.9f, 0.7f, 1.0f, brightness); break;
        case 1: glColor4f(0.7f, 0.9f, 1.0f, brightness); break;
        case 2: glColor4f(1.0f, 0.8f, 0.5f, brightness); break;
        }
        glVertex2f(exitPosX + cos(angle) * dist, exitPosY + sin(angle) * dist);
    }
    glEnd();
}

// Updated HUD to show "LIGHT" instead of "FUEL"
void renderHUD(void) {
    // Set up 2D overlay
    glMatrixMode(GL_PROJECTION); glPushMatrix(); glLoadIdentity();
    gluOrtho2D(0, windowWidth, windowHeight, 0);
    glMatrixMode(GL_MODELVIEW); glLoadIdentity();

    float time = glutGet(GLUT_ELAPSED_TIME) * 0.001f;
    float pulse = 0.8f + 0.2f * sin(time * 2.0f);

    // HUD panel background and border
    glColor4f(0.1f, 0.1f, 0.2f, 0.7f);
    glBegin(GL_QUADS);
    glVertex2f(10, 10); glVertex2f(windowWidth - 10, 10);
    glVertex2f(windowWidth - 10, 50); glVertex2f(10, 50);
    glEnd();

    glColor4f(0.3f, 0.5f, 0.8f, 0.5f * pulse);
    glBegin(GL_LINE_LOOP);
    glVertex2f(10, 10); glVertex2f(windowWidth - 10, 10);
    glVertex2f(windowWidth - 10, 50); glVertex2f(10, 50);
    glEnd();

    // Light meter (changed from Fuel meter)
    float lightBarWidth = windowWidth / 4.0f;
    float lightBarHeight = 15.0f;
    float lightBarX = 20.0f;
    float lightBarY = 25.0f;
    float lightPercentage = player.light / MAX_LIGHT_DURATION;

    // Light bar background
    glColor4f(0.15f, 0.15f, 0.25f, 0.8f);
    glBegin(GL_QUADS);
    glVertex2f(lightBarX, lightBarY);
    glVertex2f(lightBarX + lightBarWidth, lightBarY);
    glVertex2f(lightBarX + lightBarWidth, lightBarY + lightBarHeight);
    glVertex2f(lightBarX, lightBarY + lightBarHeight);
    glEnd();

    // Light indicator with color
    float r, g, b;
     if (lightPercentage > 0.6f) { r = 0.2f; g = 0.7f; b = 1.0f; } // Bright blue when
high
     else if (lightPercentage > 0.3f) { r = 0.4f; g = 0.6f; b = 0.9f; } // Medium blue
when medium
    else {
        r = 0.6f; g = 0.4f; b = 0.8f; // Purple-blue when low
        pulse = 0.7f + 0.3f * sin(time * 10.0f); // Pulsing effect when low
    }

    glColor4f(r, g, b, 0.8f * pulse);
    glBegin(GL_QUADS);
    glVertex2f(lightBarX, lightBarY);
    glVertex2f(lightBarX + lightBarWidth * lightPercentage, lightBarY);
    glVertex2f(lightBarX + lightBarWidth * lightPercentage, lightBarY + lightBarHeight);
    glVertex2f(lightBarX, lightBarY + lightBarHeight);
    glEnd();

    // Light meter label - LIGHT instead of FUEL
    glColor3f(0.8f, 0.8f, 1.0f);
    glRasterPos2f(lightBarX, lightBarY - 5);
    const char lightLabel[] = "LIGHT";
                for     (const     char*     c     =     lightLabel;     *c;     c++)
glutBitmapCharacter(GLUT_BITMAP_HELVETICA_10, *c);

    // Time remaining - no changes needed
    char timeStr[50];
    int timeRemaining = timeLimit - gameTime;
    if (timeRemaining < 0) timeRemaining = 0;
         snprintf(timeStr,  sizeof(timeStr),  "TIME:  %02d:%02d",  timeRemaining  /  60,
timeRemaining % 60);

    // Color based on remaining time
    if (timeRemaining > timeLimit / 2) glColor3f(0.7f, 1.0f, 0.7f); // Green
    else if (timeRemaining > timeLimit / 5) glColor3f(1.0f, 1.0f, 0.5f); // Yellow
    else { // Pulsing red
        float urgentPulse = 0.7f + 0.3f * sin(time * 8.0f);
        glColor3f(1.0f * urgentPulse, 0.3f * urgentPulse, 0.3f * urgentPulse);
    }

    glRasterPos2f(windowWidth - 100, 25);
    for (const char* c = timeStr; *c; c++) glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12,
*c);

    // Energy bolts collected
    char boltStr[50];
        snprintf(boltStr,  sizeof(boltStr),  "ENERGY:  %d/%d",  player.coinsCollected,
totalCoins);

    // Visual indication when all energy bolts collected
    if (player.coinsCollected == totalCoins) {
        // Electric blue pulsing effect
        float energyPulse = 0.5f + 0.5f * sin(time * 5.0f);
        glColor3f(0.3f + 0.4f * energyPulse,
            0.7f + 0.3f * energyPulse,
            1.0f);
    }
    else {
        glColor3f(0.6f, 0.8f, 1.0f);
    }

    glRasterPos2f(windowWidth / 2 - 40, 25);
    for (const char* c = boltStr; *c; c++) glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12,
*c);

    // Restore the projection matrix
    glMatrixMode(GL_PROJECTION); glPopMatrix(); glMatrixMode(GL_MODELVIEW);
}



void renderGameState(void) {
    // Set up 2D overlay for game states
    glMatrixMode(GL_PROJECTION); glPushMatrix(); glLoadIdentity();
    gluOrtho2D(0, windowWidth, windowHeight, 0);
    glMatrixMode(GL_MODELVIEW); glLoadIdentity();

    float time = glutGet(GLUT_ELAPSED_TIME) * 0.001f;

    // Render appropriate state screen
    if (currentState == GAME_WIN || currentState == GAME_LOSE) {
        // Background
        bool isWin = (currentState == GAME_WIN);
        glColor4f(isWin ? 0.0f : 0.2f, isWin ? 0.0f : 0.0f, isWin ? 0.2f : 0.0f, 0.7f);
        glBegin(GL_QUADS);
        glVertex2f(windowWidth / 2 - 250, windowHeight / 2 - 50);
        glVertex2f(windowWidth / 2 + 250, windowHeight / 2 - 50);
        glVertex2f(windowWidth / 2 + 250, windowHeight / 2 + 100);
        glVertex2f(windowWidth / 2 - 250, windowHeight / 2 + 100);
        glEnd();
        // Border
        float borderPulse = 0.7f + 0.3f * sin(time * 3.0f);
        glColor4f(isWin ? 0.3f : 0.8f * borderPulse, isWin ? 0.7f * borderPulse : 0.2f,
            isWin ? 0.3f * borderPulse : 0.2f, 0.8f);
        glLineWidth(2.0f);
        glBegin(GL_LINE_LOOP);
        glVertex2f(windowWidth / 2 - 250, windowHeight / 2 - 50);
        glVertex2f(windowWidth / 2 + 250, windowHeight / 2 - 50);
        glVertex2f(windowWidth / 2 + 250, windowHeight / 2 + 100);
        glVertex2f(windowWidth / 2 - 250, windowHeight / 2 + 100);
        glEnd();
        glLineWidth(1.0f);

        // Main message
        const char* mainMsg;
        if (isWin) {
            mainMsg = "WORMHOLE TRAVERSED SUCCESSFULLY!";
            float textPulse = 0.8f + 0.2f * sin(time * 2.0f);
            glColor3f(0.3f * textPulse, 1.0f * textPulse, 0.3f * textPulse);
        }
        else {
            // Changed to LIGHT instead of FUEL
               mainMsg = player.light <= 0 ? "LIGHT DEPLETED - MISSION FAILED!" : "TIME
EXPIRED - MISSION FAILED!";
            float textPulse = 0.8f + 0.2f * sin(time * 2.0f);
            glColor3f(1.0f * textPulse, 0.3f * textPulse, 0.3f * textPulse);
        }

        glRasterPos2f(windowWidth / 2 - 150, windowHeight / 2 - 20);
                               for    (const    char*    c    =    mainMsg;    *c;    c++)
glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, *c);

        // Stats
        if (isWin) {
            char timeMsg[100];
            snprintf(timeMsg, sizeof(timeMsg), "Mission Time: %02d:%02d", gameTime / 60,
gameTime % 60);
            glColor3f(0.7f, 0.9f, 1.0f);
            glRasterPos2f(windowWidth / 2 - 70, windowHeight / 2 + 20);
                                     for   (const   char*   c   =   timeMsg;   *c;   c++)
glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, *c);

            // Best time if applicable
            if (bestScores[currentDifficulty] != -1) {
                char bestMsg[100];
                snprintf(bestMsg, sizeof(bestMsg), "Best Time: %02d:%02d",
                     bestScores[currentDifficulty] / 60, bestScores[currentDifficulty] %
60);
                glColor3f(1.0f, 0.9f, 0.5f);
                glRasterPos2f(windowWidth / 2 - 60, windowHeight / 2 + 50);
                                        for   (const   char*   c   =   bestMsg;   *c;   c++)
glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, *c);
            }
        }
        else {
            char statsMsg[100];
```

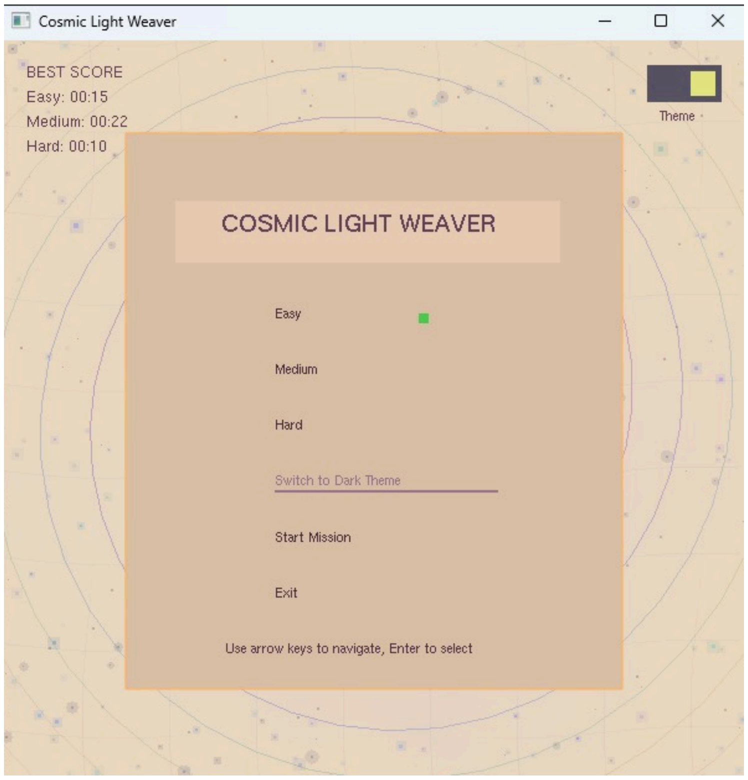
```
                      snprintf(statsMsg,  sizeof(statsMsg),  "Energy  Collected:  %d/%d",
player.coinsCollected, totalCoins);
            glColor3f(0.7f, 0.8f, 1.0f); // More blue tint for energy
            glRasterPos2f(windowWidth / 2 - 70, windowHeight / 2 + 20);
                                    for   (const   char*   c   =   statsMsg;   *c;   c++)
glutBitmapCharacter(GLUT_BITMAP_HELVETICA_14, *c);
        }

        // Instructions
        const char pressR[] = "PRESS 'R' TO RETURN TO MENU";
        glColor3f(0.8f, 0.8f, 1.0f);
        glRasterPos2f(windowWidth / 2 - 100, windowHeight / 2 + 80);
                               for    (const    char*    c    =    pressR;    *c;    c++)
glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, *c);
    }

    // Restore the projection matrix
    glMatrixMode(GL_PROJECTION); glPopMatrix(); glMatrixMode(GL_MODELVIEW);
}


void renderMenu(void) {
    // Setup 2D overlay
    glMatrixMode(GL_PROJECTION); glPushMatrix(); glLoadIdentity();
    gluOrtho2D(0, windowWidth, windowHeight, 0);
    glMatrixMode(GL_MODELVIEW); glLoadIdentity();

    float time = glutGet(GLUT_ELAPSED_TIME) * 0.001f;

    // Best scores section
    glColor3f(currentColors.textR, currentColors.textG, currentColors.textB);
    const char bestScoreLabel[] = "BEST SCORE";
    glRasterPos2f(20, 30);
                  for     (const     char*     c     =     bestScoreLabel;     *c;     c++)
glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, *c);

    // Show difficulty scores
    const char difficultyNames[][10] = { "Easy", "Medium", "Hard" };
    for (int i = 0; i < 3; i++) {
        char timeStr[50];
        if (bestScores[i] != -1) sprintf_s(timeStr, "%s: %02d:%02d", difficultyNames[i],
bestScores[i] / 60, bestScores[i] % 60);
        else sprintf_s(timeStr, "%s: --:--", difficultyNames[i]);

                  glColor3f(currentColors.textR  *  0.9f,  currentColors.textG  *  0.9f,
currentColors.textB * 0.9f);
        glRasterPos2f(20, 50 + i * 20);
                               for    (const    char*    c    =    timeStr;    *c;    c++)
glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, *c);
    }

    // Theme toggle switch
    float toggleX = windowWidth - 80, toggleY = 20;

    // Draw switch background and handle
    glColor4f(0.2f, 0.2f, 0.3f, 0.8f);
    glBegin(GL_QUADS);
    glVertex2f(toggleX, toggleY); glVertex2f(toggleX + 60, toggleY);
    glVertex2f(toggleX + 60, toggleY + 30); glVertex2f(toggleX, toggleY + 30);
    glEnd();

    float handlePos = currentTheme == THEME_DARK ? toggleX + 5 : toggleX + 35;
    glColor4f(currentTheme == THEME_DARK ? 0.3f : 0.9f, currentTheme == THEME_DARK ?
0.5f : 0.9f,
        currentTheme == THEME_DARK ? 0.9f : 0.5f, 1.0f);
    glBegin(GL_QUADS);
    glVertex2f(handlePos, toggleY + 5); glVertex2f(handlePos + 20, toggleY + 5);
    glVertex2f(handlePos + 20, toggleY + 25); glVertex2f(handlePos, toggleY + 25);
    glEnd();

    // Theme label
    glColor3f(currentColors.textR, currentColors.textG, currentColors.textB);
    const char themeLabel[] = "Theme";
    glRasterPos2f(toggleX + 10, toggleY + 45);
    for (const char* c = themeLabel; *c; c++)
glutBitmapCharacter(GLUT_BITMAP_HELVETICA_10, *c);

    // Main panel
    float panelWidth = 400, panelHeight = 450;
    float panelX = windowWidth / 2 - panelWidth / 2, panelY = windowHeight / 2 -
panelHeight / 2;

    // Panel background and border
    glColor4f(currentColors.uiR, currentColors.uiG, currentColors.uiB, 0.8f);
    glBegin(GL_QUADS);
    glVertex2f(panelX, panelY); glVertex2f(panelX + panelWidth, panelY);
    glVertex2f(panelX + panelWidth, panelY + panelHeight); glVertex2f(panelX, panelY +
panelHeight);
    glEnd();

    glColor4f(currentColors.accentR, currentColors.accentG, currentColors.accentB,
0.6f);
    glLineWidth(2.0f);
    glBegin(GL_LINE_LOOP);
    glVertex2f(panelX, panelY); glVertex2f(panelX + panelWidth, panelY);
    glVertex2f(panelX + panelWidth, panelY + panelHeight); glVertex2f(panelX, panelY +
panelHeight);
    glEnd();
    glLineWidth(1.0f);

    // Title
    const char title[] = "COSMIC LIGHT WEAVER";
    float titleX = panelX + panelWidth / 2 - 130, titleY = panelY + 80;

    // Title background
    glColor4f(currentColors.uiR + 0.1f, currentColors.uiG + 0.1f, currentColors.uiB +
0.1f, 0.5f);
    glBegin(GL_QUADS);
    glVertex2f(titleX - 30, titleY - 25); glVertex2f(titleX + 280, titleY - 25);
    glVertex2f(titleX + 280, titleY + 25); glVertex2f(titleX - 30, titleY + 25);
    glEnd();
    // Title text with scaling
    float scaleFactor = 1.0f + 0.1f * sin(time * 2.0f);
    glColor3f(currentColors.textR, currentColors.textG, currentColors.textB);
    glPushMatrix();
    glTranslatef(titleX + 130, titleY, 0);
    glScalef(scaleFactor, scaleFactor, 1.0f);
    glTranslatef(-(titleX + 130), -titleY, 0);

    glRasterPos2f(titleX, titleY);
     for (const char* c = title; *c; c++) glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18,
*c);
    glPopMatrix();

    // Menu options
    const char* options[] = {
        "Easy", "Medium", "Hard",
        currentTheme == THEME_DARK ? "Switch to Light Theme" : "Switch to Dark Theme",
        "Start Mission", "Exit"
    };

     float optionY = panelY + 150, optionSpacing = 45, optionX = panelX + panelWidth / 2
- 80;

    // Draw options
    for (int i = 0; i < MENU_COUNT; i++) {
        if (i == selectedOption) {
            // Selected option
                    glColor3f(currentColors.textR  +  0.2f,  currentColors.textG  +  0.2f,
currentColors.textB + 0.2f);
            glPushMatrix();
            glTranslatef(optionX, optionY + i * optionSpacing, 0);
            glScalef(1.2f, 1.2f, 1.0f);
            glTranslatef(-optionX, -(optionY + i * optionSpacing), 0);
            glRasterPos2f(optionX, optionY + i * optionSpacing);
                               for   (const   char*   c   =   options[i];   *c;   c++)
glutBitmapCharacter(GLUT_BITMAP_HELVETICA_14, *c);
            glPopMatrix();

            // Underline
            float textWidth = strlen(options[i]) * 9;
            glLineWidth(2.0f);
            glBegin(GL_LINES);
            glVertex2f(optionX, optionY + i * optionSpacing + 5);
            glVertex2f(optionX + textWidth, optionY + i * optionSpacing + 5);
            glEnd();
            glLineWidth(1.0f);
        }
        else {
            // Non-selected options
                    glColor3f(currentColors.textR  *  0.7f,  currentColors.textG  *  0.7f,
currentColors.textB * 0.7f);
            glRasterPos2f(optionX, optionY + i * optionSpacing);
                               for   (const   char*   c   =   options[i];   *c;   c++)
glutBitmapCharacter(GLUT_BITMAP_HELVETICA_14, *c);
        }
        // Difficulty indicator
        if (i < 3 && i == currentDifficulty) {
            glColor3f(0.3f, 0.8f, 0.3f);
            glPointSize(8.0f);
             glBegin(GL_POINTS); glVertex2f(optionX + 120, optionY + i * optionSpacing);
glEnd();
            glPointSize(1.0f);
        }
    }

    // Instructions
    const char instructions[] = "Use arrow keys to navigate, Enter to select";
             glColor3f(currentColors.textR   *   0.8f,   currentColors.textG   *   0.8f,
currentColors.textB * 0.8f);
    glRasterPos2f(panelX + panelWidth / 2 - 120, panelY + panelHeight - 30);
                    for     (const     char*     c     =     instructions;     *c;     c++)
glutBitmapCharacter(GLUT_BITMAP_HELVETICA_10, *c);

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

void renderGame(void) {
    // Render game elements in proper order
    renderSpace(); renderTrail(); renderCoins(); renderExit(); renderPlayer();
    // Overlay UI
    renderHUD();
    // Game state overlays (win/lose screens)
    if (currentState == GAME_WIN || currentState == GAME_LOSE) renderGameState();
}

void display(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    // Background effects
    renderBackgroundEffects(); renderStarsAndNebulas(); renderParticles();
    // Render game or menu based on state
    if (currentState == GAME_MENU) renderMenu(); else renderGame();
    glutSwapBuffers();
}

void reshape(int w, int h) {
    windowWidth = w; windowHeight = h;
    glViewport(0, 0, (GLsizei)w, (GLsizei)h);
    glMatrixMode(GL_PROJECTION); glLoadIdentity();
    gluOrtho2D(0.0, GRID_WIDTH * CELL_SIZE, GRID_HEIGHT * CELL_SIZE, 0.0);
    glMatrixMode(GL_MODELVIEW);
}

void keyboard(unsigned char key, int x, int y) {
    if (currentState == GAME_MENU) {
        switch (key) {
        case 13: // Enter key
            switch (selectedOption) {
            case MENU_EASY: currentDifficulty = DIFFICULTY_EASY; break;
            case MENU_MEDIUM: currentDifficulty = DIFFICULTY_MEDIUM; break;
            case MENU_HARD: currentDifficulty = DIFFICULTY_HARD; break;
            case MENU_THEME:
                currentTheme = (currentTheme == THEME_DARK) ? THEME_LIGHT : THEME_DARK;
                updateThemeColors(); break;
            case MENU_START: updateDifficultySettings(); startNewGame(); break;
            case MENU_EXIT: exit(0); break;
            }
            break;
        case 't': case 'T': // Theme toggle shortcut
            currentTheme = (currentTheme == THEME_DARK) ? THEME_LIGHT : THEME_DARK;
            updateThemeColors(); break;
        case 'q': case 'Q': case 27: exit(0); break; // ESC key
        }
        glutPostRedisplay(); return;
    }

    if (currentState == GAME_WIN || currentState == GAME_LOSE) {
        if (key == 'r' || key == 'R') { currentState = GAME_MENU; glutPostRedisplay(); }
        else if (key == 'q' || key == 'Q' || key == 27) exit(0);
        return;
    }

    // Game controls
    float newX = player.x, newY = player.y;
    switch (key) {
    case 'w': case 'W': newY -= 1.0f; break;
    case 's': case 'S': newY += 1.0f; break;
    case 'a': case 'A': newX -= 1.0f; break;
    case 'd': case 'D': newX += 1.0f; break;
    case 27: currentState = GAME_MENU; break; // ESC key
    }

    if (isValidMove(newX, newY)) {
        player.x = newX; player.y = newY;
        addTrailPoint(player.x, player.y);
        checkCoinCollision(); checkWinCondition();
    }
    glutPostRedisplay();
}

void specialKeys(int key, int x, int y) {
    if (currentState == GAME_MENU) {
        switch (key) {
        case GLUT_KEY_UP:
                selectedOption = (MenuOption)((selectedOption == 0) ? MENU_COUNT - 1 :
selectedOption - 1); break;
        case GLUT_KEY_DOWN:
            selectedOption = (MenuOption)((selectedOption + 1) % MENU_COUNT); break;
        }
        glutPostRedisplay(); return;
    }

    if (currentState != GAME_PLAYING) return;

    float newX = player.x, newY = player.y;
    switch (key) {
    case GLUT_KEY_UP: newY -= 1.0f; break;
    case GLUT_KEY_DOWN: newY += 1.0f; break;
    case GLUT_KEY_LEFT: newX -= 1.0f; break;
    case GLUT_KEY_RIGHT: newX += 1.0f; break;
    }

    if (isValidMove(newX, newY)) {
        player.x = newX; player.y = newY;
        addTrailPoint(player.x, player.y);
        checkCoinCollision(); checkWinCondition();
    }
    glutPostRedisplay();
}

void update(int value) {
    float time = glutGet(GLUT_ELAPSED_TIME) * 0.001f;

    if (currentState == GAME_PLAYING) {
        // Decrease player light
        float decayRate;
        switch (currentDifficulty) {
        case DIFFICULTY_EASY: decayRate = LIGHT_DECAY_RATE * 0.6f; break;
        case DIFFICULTY_MEDIUM: decayRate = LIGHT_DECAY_RATE * 1.0f; break;
        case DIFFICULTY_HARD: decayRate = LIGHT_DECAY_RATE * 1.5f; break;
        default: decayRate = LIGHT_DECAY_RATE;
        }
        player.light -= decayRate;

        // Update trail intensities and remove faded points
        for (int i = 0; i < trailLength; i++) trail[i].intensity -= 0.2f;
        int i = 0;
        while (i < trailLength) {
            if (trail[i].intensity <= 0) {
                            memmove(&trail[i], &trail[i + 1], (trailLength - i - 1) *
sizeof(TrailPoint));
                trailLength--;
            }
            else i++;
        }

        // Twinkle stars
        for (int i = 0; i < MAX_STARS; i++)
                if (rand() % 30 == 0) stars[i].brightness = 0.3f + ((float)rand() /
RAND_MAX) * 0.7f;

        // Animate nebulas
        for (int i = 0; i < MAX_NEBULAS; i++)
            nebulas[i].a = (0.05f + 0.03f * sin(time * nebulas[i].pulse_speed));

        // Check lose condition
        if (player.light <= 0 || gameTime >= timeLimit) currentState = GAME_LOSE;
    }

    // Update particles in all game states
    for (int i = 0; i < MAX_PARTICLES; i++) {
        // Age particles
        particles[i].age += 0.5f;
        // Reset dead particles
        if (particles[i].age >= particles[i].lifespan) {
            // Spawn from sides
            if (rand() % 2 == 0) {
                // Left or right
                particles[i].x = rand() % 2 == 0 ? 0 : windowWidth;
                particles[i].y = rand() % windowHeight;
                 particles[i].vx = particles[i].x == 0 ? (0.2f + (rand() % 20) / 100.0f)
:
                    -(0.2f + (rand() % 20) / 100.0f);
                particles[i].vy = (float)(rand() % 60 - 30) / 300.0f;
            }
            else {
                // Top or bottom
                particles[i].y = rand() % 2 == 0 ? 0 : windowHeight;
                particles[i].x = rand() % windowWidth;
                 particles[i].vy = particles[i].y == 0 ? (0.2f + (rand() % 20) / 100.0f)
:
                    -(0.2f + (rand() % 20) / 100.0f);
                particles[i].vx = (float)(rand() % 60 - 30) / 300.0f;
            }

            // Reset properties
            particles[i].size = 1.0f + (rand() % 30) / 10.0f;
            particles[i].color[0] = 0.1f + (rand() % 30) / 100.0f;
            particles[i].color[1] = 0.2f + (rand() % 40) / 100.0f;
            particles[i].color[2] = 0.5f + (rand() % 50) / 100.0f;
            particles[i].alpha = 0.1f + (rand() % 40) / 100.0f;
            particles[i].age = 0; particles[i].lifespan = 50.0f + rand() % 100;
        }

        // Move particles with wave motion
        particles[i].x += particles[i].vx + sin(time + particles[i].y * 0.01f) * 0.2f;
        particles[i].y += particles[i].vy + cos(time + particles[i].x * 0.01f) * 0.2f;
    }

    glutPostRedisplay();
    glutTimerFunc(100, update, 0);
}

void updateTimer(int value) {
    if (currentState == GAME_PLAYING) gameTime++;
    glutTimerFunc(1000, updateTimer, 0);
}

int main(int argc, char** argv) {
    srand((unsigned int)time(NULL));
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(windowWidth, windowHeight);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Cosmic Light Weaver");
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutSpecialFunc(specialKeys);
    glutTimerFunc(100, update, 0);
    glutTimerFunc(1000, updateTimer, 0);
    glutMainLoop();
    return 0;
}
```

Output Screenshots

Main Menu:



Main Menu(with Light Theme):



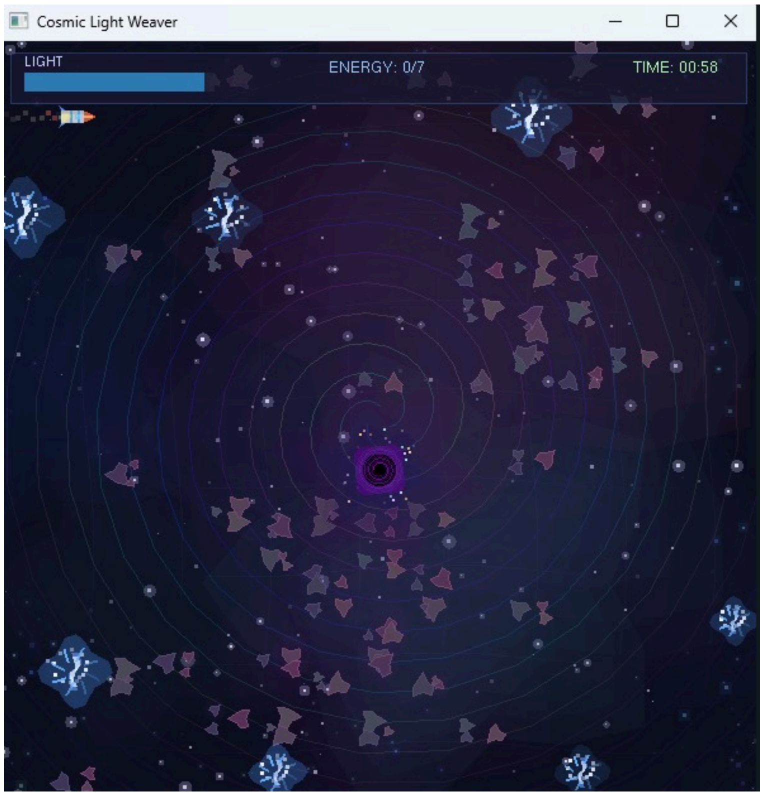
BEST SCORE
Easy: 00:15
Medium: 00:22
Hard: 00:10

COSMIC LIGHT WEAVER

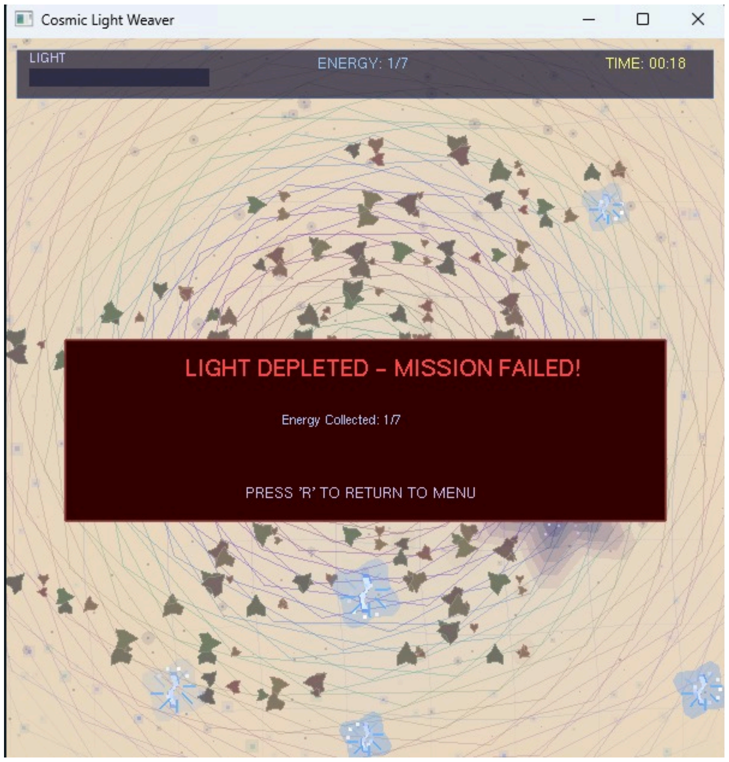
Easy
Medium
Hard
Switch to Dark Theme
Start Mission
Exit

Use arrow keys to navigate, Enter to select

Gameplay:



Mission Failed — Light Source Exhausted:

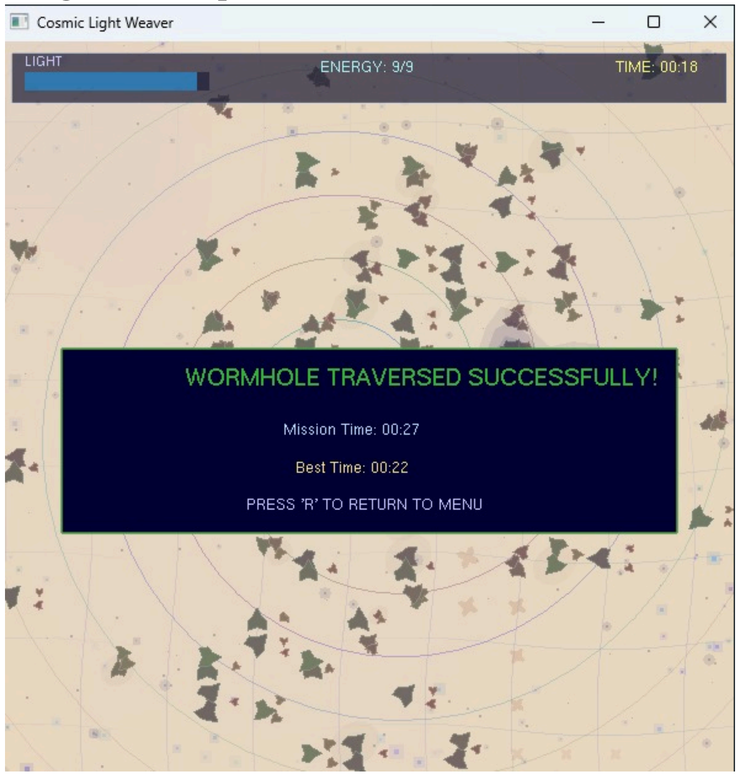


LIGHT DEPLETED - MISSION FAILED!

Energy Collected: 1/7

PRESS 'R' TO RETURN TO MENU

Victory! Wormhole Navigation Complete:



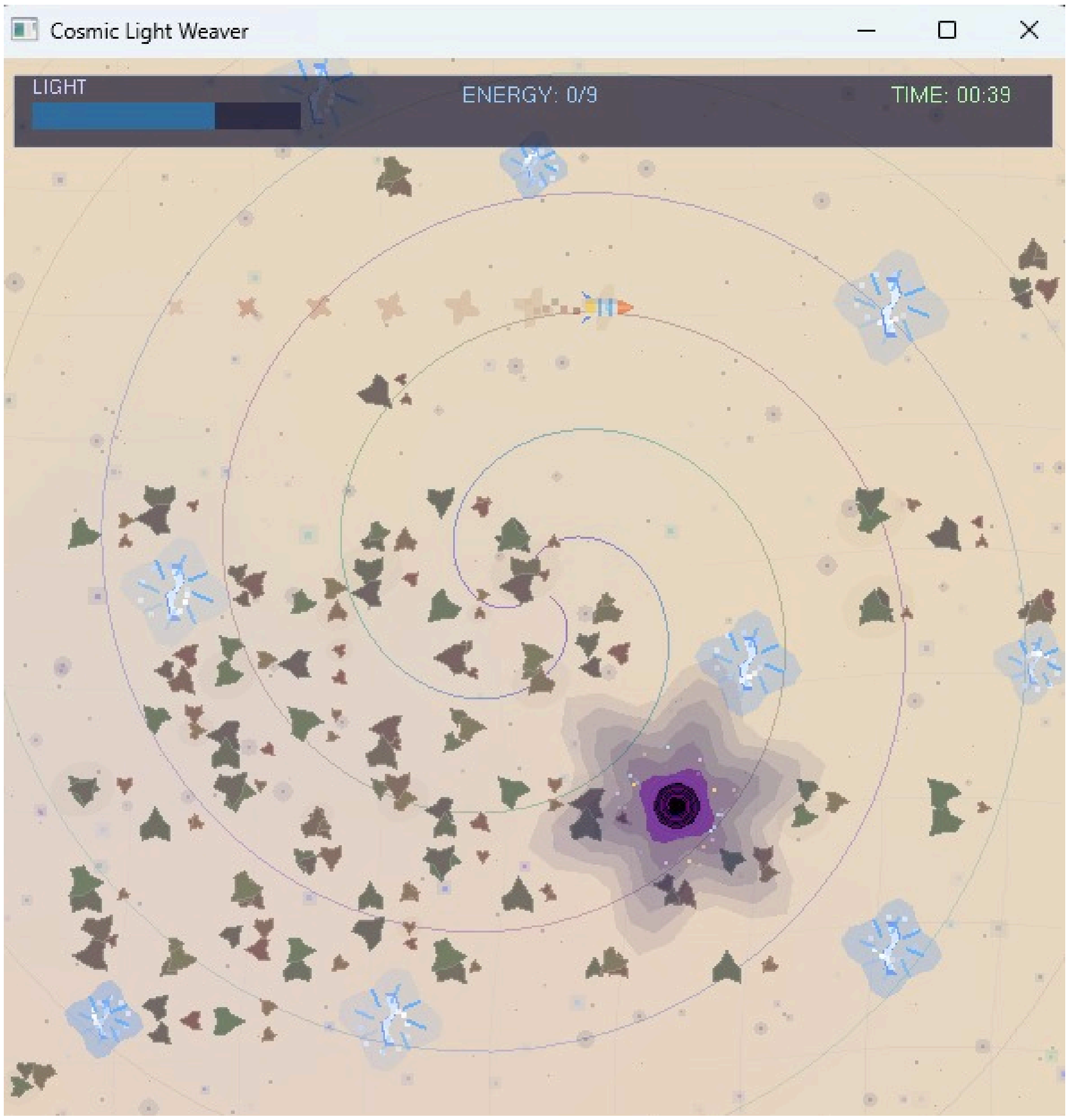
WORMHOLE TRAVERSED SUCCESSFULLY!

Mission Time: 00:27

Best Time: 00:22

PRESS 'R' TO RETURN TO MENU

Gameplay in progress:



****