Computer Graphics (UCS505)

CROSSY ROADS

Branch

B.E. 3rd Year – COE/CSE

Submitted By –

Yash Saxena	102203915
Madhur Mahajan	102203491
Nikhil Garg	102203275

Submitted To –
Dr. Jasmine Kaur

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

Computer Science and Engineering Department

Thapar Institute of Engineering and Technology

Patiala – 147001

INTRODUCTION

Project Overview:

This project is a 3D endless runner inspired by Crossy Roads game implemented using OpenGL and GLUT in C++. The game places a player on a linear or slightly varied path suspended in space. The player's goal is to move forward while avoiding dynamic obstacles like spinning red cubes. **Player Controls:**

- Movement using **W, A, S, D** keys.
- Jumping with **Space**.
- Camera control via **V** (change view) and **C** (toggle rotation).

Obstacles:

- Can be static or spinning cubes.
- Positioned along the player's path.

Graphics & Rendering:

- Uses OpenGL features like lighting, materials and shading .
- Displays 3D shapes like cubes.
- Perspective view adjustment.

Game Logic:

- A timer function updates frame rendering every ~16ms.
- Player's movement and camera angle change based on input.
- Obstacle interactions are updated dynamically.

Utility Visuals:

- Directional arrows with labeled keys (W/A/S/D) guide the player.

Scope of the Project:

Educational Value:

- Demonstrates core OpenGL programming concepts: 3D rendering, transformations and lighting materials.
- Great for learning real-time rendering and animation logic.

Gameplay Elements:

- Simple yet engaging mechanics similar to "Temple Run" or "Subway Surfers".
- Allows testing of user inputs, rendering speed and basic physics (jumping, rotation).

Graphics Features:

- Can be extended with shaders, texture mapping or improved lighting models.

Modularity:

- Uses structured functions and a game class (game.updateGame, game.keyW, etc.), allowing further extension.

USER DEFINED FUNCTIONS

S No.	Function Name	Function Description
1	isCornerPoint	Checks if a position is a corner in the path
2	isAdjacentToCorner	Checks if a position is adjacent to a corner in the path
3	hasObstacle	Checks if a position already has an obstacle
4	isAdjacentToObstacle	Checks if a position is adjacent to another obstacle
5	extendPath	Extends the game path by generating additional segments
6	generateInitialPath	Creates the initial path for the game
7	generateObstacles	Generates obstacles for a path segment
8	onPath	Checks if coordinates are on the current valid path
9	checkObstacleCollision	Determines if player collides with obstacles
10	updateGame	Updates game state based on time delta
11	reset	Resets the game to initial state
12	nextCameraMode	Cycles to the next camera viewing mode
13	toggleCameraRotation	Toggles fixed or rotating camera angle
14	zoomIn	Decreases camera distance (zooms in)
15	zoomOut	Increases camera distance (zooms out)
16	displayText	Renders text on screen at specified position
17	drawTexturedCube	Draws a cube with texture mapping
18	drawCube	Draws a colored cube with material properties
19	drawArrow	Draws directional arrows for player controls
20	drawObstacle	Renders obstacles with appropriate visual effects
21	drawPlayer	Renders the player cube with animations
22	drawSkybox	Creates a skybox environment
23	drawGrid	Draws reference grid on the ground
24	display	Main rendering function called by GLUT
25	timer	Updates game state at regular intervals
26	reshape	Handles window resize events
27	keyboard	Processes keyboard press events
28	keyboardUp	Processes keyboard release events
29	initGL	Initializes OpenGL settings and lighting
30	main	Entry point for the application

CODE SNIPPETS

```
#include <GL/glut.h>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <string>
#include <algorithm>
#include <cmath>
#include <tuple>
#include <stdexcept>
#include <set>

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

class Game {
public:
    // Game state
    int score = 0;
    float playerX = 0, playerY = 1.0, playerZ = 0;
    bool isRolling = false;
    float rollAngle = 0.0f;
    int rollDirection = 0; // 0=none, 1=forward, 2=backward, 3=left, 4=right
    float rollProgress = 0.0f;
    bool gameOver = false;
    bool showDirections = true;
    int maxDistanceTraveled = 0;


    // Jump-movement mechanics
    bool isJumping = false;
    float jumpHeight = 0.0f;
    float jumpProgress = 0.0f;
    static constexpr float MAX_JUMP_HEIGHT = 1.5f;
    static constexpr float JUMP_SPEED = 2.0f;
    float jumpDestX = 0.0f, jumpDestZ = 0.0f;
    float jumpStartX = 0.0f, jumpStartZ = 0.0f;

    // Camera settings
    int cameraMode = 0;
    float cameraDistance = 8.0f;
    float cameraAngle = 45.0f;
    bool fixedCameraAngle = true;

    // Movement controls
    bool keyW = false;
    bool keyS = false;
    bool keyA = false;
    bool keyD = false;
    bool keySpace = false;

    // Game settings
    static constexpr int INITIAL_PATH_LENGTH = 20;
    static constexpr int PATH_SEGMENT_LENGTH = 15;
    static constexpr float ROLL_SPEED = 3.0f;
    static constexpr float CUBE_SIZE = 1.0f;
    static constexpr float PLATFORM_LIFETIME = 3.0f;
static constexpr float PATH_EXTENSION_THRESHOLD = 10.0f;
// FIXED: Changed from 0.9 to 0.3 to increase obstacle spawn rate
static constexpr float OBSTACLE_SPAWN_CHANCE = 0.6f;

// Game elements
// Path tuple: (x, z, lifetime, max_lifetime, isCorner)
std::vector<std::tuple<int, int, float, float, bool>> path;
int maxX = 0, maxZ = 0;
int prevDirection = -1; // -1=initial, 0=x, 1=z

// Obstacle types
enum ObstacleType {
    NONE = 0,
    RISING_BLOCK = 1,
    FALLING_BLOCK = 2,
    SPINNING_BLOCK = 3,
    MOVING_BLOCK = 4
};

struct Obstacle {
    int x, z;
    ObstacleType type;
    float progress;
    bool active;
    float height;
    float rotation;
    float offsetX, offsetZ;
};

std::vector<Obstacle>  obstacles;

// Helper function to check if a position is a corner in the path
bool isCornerPoint(int x, int z) const {
    for (auto& tile : path) {
        int tileX = std::get<0>(tile);
        int tileZ = std::get<1>(tile);
        bool isCorner = std::get<4>(tile);

        if (tileX == x && tileZ == z && isCorner) {
            return true;
        }
    }
    return false;
}

// Helper function to check if a position is adjacent to a corner in the path
bool isAdjacentToCorner(int x, int z) const {
    for (auto& tile : path) {
        int tileX = std::get<0>(tile);
        int tileZ = std::get<1>(tile);
        bool isCorner = std::get<4>(tile);

        if (isCorner &&
            (std::abs(tileX - x) + std::abs(tileZ - z) == 1)) {
            return true;
        }
    }
    return false;
}

// Helper function to check if a position already has an obstacle
bool hasObstacle(int x, int z) const {
        for (const auto& obstacle : obstacles) {
            if (obstacle.x == x && obstacle.z == z && obstacle.active) {
                return true;
            }
        }
        return false;
    }

    // Helper function to check if a position is adjacent to another obstacle
    bool isAdjacentToObstacle(int x, int z) const {
        for (const auto& obstacle : obstacles) {
            if (obstacle.active &&
                (std::abs(obstacle.x - x) + std::abs(obstacle.z - z) == 1)) {
                return true;
            }
        }
        return false;
    }

    // Improved version of extendPath() method that only generates path without obstacles
    void extendPath() {
        try {
            int x = maxX, z = maxZ;
            int currentDirection = prevDirection;

            // Get the last point's details to ensure continuity
            if (!path.empty()) {
                x = std::get<0>(path.back());
                z = std::get<1>(path.back());
            }

            std::vector<std::tuple<int, int, float, float, bool>> newPathSegment;

            for (int i = 0; i < PATH_SEGMENT_LENGTH; ++i) {
                int nextDirection;
                do {
                    // Randomly choose a direction (0 = x, 1 = z)
                    nextDirection = rand() % 2;

                    // Force a direction change if we've been going the same way for too long
                    if (i > 0 && i % 5 == 0) {
                        nextDirection = (currentDirection == 0) ? 1 : 0;
                    }
                } while (nextDirection == currentDirection && i > 0 && rand() % 3 == 0); // Encourage some
turns

                // Determine if this will be a corner point
                bool isCorner = (currentDirection != -1 && currentDirection != nextDirection);

                // Move in the chosen direction
                if (nextDirection == 1)
                    z += 1;
                else
                    x += 1;

                maxX = std::max(maxX, x);
                maxZ = std::max(maxZ, z);

                // Add the new point to the path segment
                newPathSegment.push_back(std::make_tuple(x, z, PLATFORM_LIFETIME,
PLATFORM_LIFETIME, isCorner));
```

```
                currentDirection = nextDirection;
            }

            // Append the new path segment to the main path
            path.insert(path.end(), newPathSegment.begin(), newPathSegment.end());
            prevDirection = currentDirection;

            // Now generate obstacles for the new path segment
            generateObstacles(newPathSegment);
        }
        catch (const std::exception& e) {
            std::cerr << "Error in extendPath: " << e.what() << std::endl;
            throw;
        }
    }

    // Improved version of generateInitialPath() method that only generates path without obstacles
    void generateInitialPath() {
        try {
            path.clear();
            obstacles.clear();
            maxX = maxZ = 0;
            prevDirection = -1;

            int x = 0, z = 0;
            // Add starting point (not a corner)
            path.push_back(std::make_tuple(x, z, PLATFORM_LIFETIME, PLATFORM_LIFETIME,
false));

            int currentDirection = -1;
            int straightCounter = 0; // Used to track how long we've been going straight

            for (int i = 1; i < INITIAL_PATH_LENGTH; ++i) {
                int nextDirection;

                if (i <= 5) {
                    // For the first few steps, ensure we have a clear starting path without corners
                    nextDirection = 0; // Start with an X direction path
                }
                else {
                    // After initial straight segment, introduce possible turns
                    if (straightCounter >= 3) {
                        // Force a turn if we've been going straight for too long
                        nextDirection = (currentDirection == 0) ? 1 : 0;
                        straightCounter = 0;
                    }
                    else {
                        // Otherwise, randomly choose direction with some bias toward continuing
                        if (rand() % 3 == 0) { // 1/3 chance of changing direction
                            nextDirection = (currentDirection == 0) ? 1 : 0;
                            straightCounter = 0;
                        }
                        else {
                            nextDirection = currentDirection == -1 ? (rand() % 2) : currentDirection;
                            straightCounter++;
                        }
                    }
                }

                // Determine if this will be a corner
                bool isCorner = (currentDirection != -1 && currentDirection != nextDirection);
                // Move in the chosen direction
                if (nextDirection == 1)
                    z += 1;
                else
                    x += 1;

                maxX = std::max(maxX, x);
                maxZ = std::max(maxZ, z);

                // Add the new point to the path
                path.push_back(std::make_tuple(x, z, PLATFORM_LIFETIME, PLATFORM_LIFETIME,
isCorner));

                currentDirection = nextDirection;
            }

            prevDirection = currentDirection;

            // Now generate obstacles after the entire path is created
            generateObstacles(path, 6); // Skip the first 6 tiles for a clear starting path
        }
        catch (const std::exception& e) {
            std::cerr << "Error in generateInitialPath: " << e.what() << std::endl;
            throw;
        }
    }
    void generateObstacles(const std::vector<std::tuple<int, int, float, float, bool>>& pathSegment, int
startIndex = 0) {
        try {
            for (size_t i = startIndex; i < pathSegment.size(); ++i) {
                int x = std::get<0>(pathSegment[i]);
                int z = std::get<1>(pathSegment[i]);

                // Skip first 5 tiles for safe zone
                if (i < 5 && startIndex == 0) continue;

                // Skip corners
                if (isCornerPoint(x, z)) continue;

                // Check orthogonal adjacency to any corner
                if (isAdjacentToCorner(x, z)) continue;

                // Check existing obstacles
                if (hasObstacle(x, z)) continue;
                if (isAdjacentToObstacle(x, z)) continue;

                // Find position in full path
                int currentIndex = -1;
                for (size_t idx = 0; idx < path.size(); ++idx) {
                    if (std::get<0>(path[idx]) == x && std::get<1>(path[idx]) == z) {
                        currentIndex = idx;
                        break;
                    }
                }

                // Determine if middle of straight segment
                bool isMiddleStraight = false;
                if (currentIndex > 0 && currentIndex < path.size() - 1) {
                    auto& prevTile = path[currentIndex - 1];
                    auto& nextTile = path[currentIndex + 1];

                    int prevX = std::get<0>(prevTile);
                    int prevZ = std::get<1>(prevTile);
                    int nextX = std::get<0>(nextTile);
                    int nextZ = std::get<1>(nextTile);

                    // Check straight segment in X-direction
                    if (prevX == x - 1 && prevZ == z && nextX == x + 1 && nextZ == z) {
                        isMiddleStraight = true;
                    }
                    // Check straight segment in Z-direction
                    else if (prevZ == z - 1 && prevX == x && nextZ == z + 1 && nextX == x) {
                        isMiddleStraight = true;
                    }
                }

                // Adjust probabilities based on position
                float probability = OBSTACLE_SPAWN_CHANCE;
                if (isMiddleStraight) {
                    probability = 0.8f;  // Higher chance in middle of straight segments
                }

                if ((rand() / static_cast<float>(RAND_MAX)) < probability) {
                    Obstacle newObstacle;
                    newObstacle.x = x;
                    newObstacle.z = z;
                    newObstacle.type = static_cast<ObstacleType>(1 + (rand() % 4));
                    newObstacle.progress = 0.0f;
                    newObstacle.active = true;
                    newObstacle.height = 0.0f;
                    newObstacle.rotation = 0.0f;
                    newObstacle.offsetX = 0.0f;
                    newObstacle.offsetZ = 0.0f;
                    obstacles.push_back(newObstacle);
                }
            }
        }
        catch (const std::exception& e) {
            std::cerr << "Error in generateObstacles: " << e.what() << std::endl;
            throw;
        }
    }

    bool onPath(float x, float z) {
        try {
            int roundedX = std::round(x);
            int roundedZ = std::round(z);

            for (auto& tile : path) {
                int tileX = std::get<0>(tile);
                int tileZ = std::get<1>(tile);
                float tileLife = std::get<2>(tile);

                if (tileLife > 0.0f && tileX == roundedX && tileZ == roundedZ) {
                    return true;
                }
            }
            return false;
        }
        catch (const std::exception& e) {
            std::cerr << "Error in onPath: " << e.what() << std::endl;
            return false;
        }
    }
```

```
bool checkObstacleCollision(float x, float y, float z) {
    try {
        for (auto& obstacle : obstacles) {
            if (!obstacle.active) continue;

            if (std::round(x) == obstacle.x && std::round(z) == obstacle.z) {
                switch (obstacle.type) {
                case RISING_BLOCK:
                case FALLING_BLOCK:
                    if (y <= obstacle.height + 0.5f && y + 0.5f >= obstacle.height - 0.5f) {
                        return true;
                    }
                    break;
                case SPINNING_BLOCK:
                    if (y <= 1.5f) {
                        return true;
                    }
                    break;
                case MOVING_BLOCK:
                    if (y <= 1.0f &&
                        x >= obstacle.x - 0.5f + obstacle.offsetX && x <= obstacle.x + 0.5f + obstacle.offsetX &&
                        z >= obstacle.z - 0.5f + obstacle.offsetZ && z <= obstacle.z + 0.5f + obstacle.offsetZ) {
                        return true;
                    }
                    break;
                }
            }
        }
        return false;
    }
    catch (const std::exception& e) {
        std::cerr << "Error in checkObstacleCollision: " << e.what() << std::endl;
        return false;
    }
}

void updateGame(float deltaTime) {
    if (gameOver) return;

    try {
        // Update platform lifetimes
        for (auto& tile : path) {
            int tileX = std::get<0>(tile);
            int tileZ = std::get<1>(tile);
            float& tileLife = std::get<2>(tile);
            float maxLife = std::get<3>(tile);

            if (tileX < playerX || tileZ < playerZ) {
                tileLife -= deltaTime;
            }

            if (tileLife < 0.0f) tileLife = 0.0f;
        }

        // Update obstacles
        for (auto& obstacle : obstacles) {
            if (!obstacle.active) continue;

            obstacle.progress += deltaTime;

            switch (obstacle.type) {
```

```
    case RISING_BLOCK:
        obstacle.height = std::min(1.0f, obstacle.progress * 0.5f);
        break;
    case FALLING_BLOCK:
    {
        float fallTime = 1.0f;
        if (obstacle.progress < fallTime) {
            obstacle.height = 2.0f - (obstacle.progress / fallTime) * 2.0f;
        }
        else {
            obstacle.height = 0.0f;
        }
        break;
    }
    case SPINNING_BLOCK:
        obstacle.rotation += deltaTime * 180.0f;
        obstacle.height = 0.5f + 0.3f * sin(obstacle.progress * 3.0f);
        break;
    case MOVING_BLOCK:
        obstacle.offsetX = 0.5f * sin(obstacle.progress * 2.0f);
        obstacle.offsetZ = 0.5f * cos(obstacle.progress * 2.0f);
        break;
    }
}

// Handle jumping movement
if (isJumping) {
    jumpProgress += JUMP_SPEED * deltaTime;

    if (jumpProgress >= 1.0f) {
        jumpProgress = 0.0f;
        isJumping = false;
        playerX = jumpDestX;
        playerZ = jumpDestZ;
        jumpHeight = 0.0f;

        if (!onPath(playerX, playerZ)) {
            gameOver = true;
        }

        rollDirection = 0;

        if (!gameOver) {
            float distanceToEnd = std::sqrt(
                std::pow(maxX - playerX, 2) +
                std::pow(maxZ - playerZ, 2)
            );

            if (distanceToEnd < PATH_EXTENSION_THRESHOLD) {
                extendPath();
            }
        }
    }
    else {
        float t = jumpProgress;
        jumpHeight = MAX_JUMP_HEIGHT * std::sin(t * M_PI);
        playerX = jumpStartX + t * (jumpDestX - jumpStartX);
        playerZ = jumpStartZ + t * (jumpDestZ - jumpStartZ);

        if (checkObstacleCollision(playerX, playerY + jumpHeight, playerZ)) {
            gameOver = true;
        }
    }
}
// Handle rolling animation
else if (isRolling) {
    rollProgress += ROLL_SPEED * deltaTime;
    rollAngle = rollProgress * 90.0f;

    if (rollProgress >= 1.0f) {
        isRolling = false;
        rollProgress = 0.0f;
        rollAngle = 0.0f;

        switch (rollDirection) {
        case 1: playerZ -= 1.0f; break;
        case 2: playerZ += 1.0f; break;
        case 3: playerX -= 1.0f; break;
        case 4: playerX += 1.0f; break;
        }

        if (!onPath(playerX, playerZ)) {
            gameOver = true;
        }

        if (!gameOver && checkObstacleCollision(playerX, playerY, playerZ)) {
            gameOver = true;
        }

        rollDirection = 0;

        if (!gameOver) {
            float distanceToEnd = std::sqrt(
                std::pow(maxX - playerX, 2) +
                std::pow(maxZ - playerZ, 2)
            );

            if (distanceToEnd < PATH_EXTENSION_THRESHOLD) {
                extendPath();
            }
        }
    }
}
// Handle player input for movement
else {
    bool canMove = false;
    int newDirection = 0;

    if (keyW) {
        canMove = true;
        newDirection = 1;
        showDirections = false;
    }
    else if (keyS) {
        canMove = true;
        newDirection = 2;
        showDirections = false;
    }
    else if (keyA) {
        canMove = true;
        newDirection = 3;
        showDirections = false;
    }
    else if (keyD) {
```

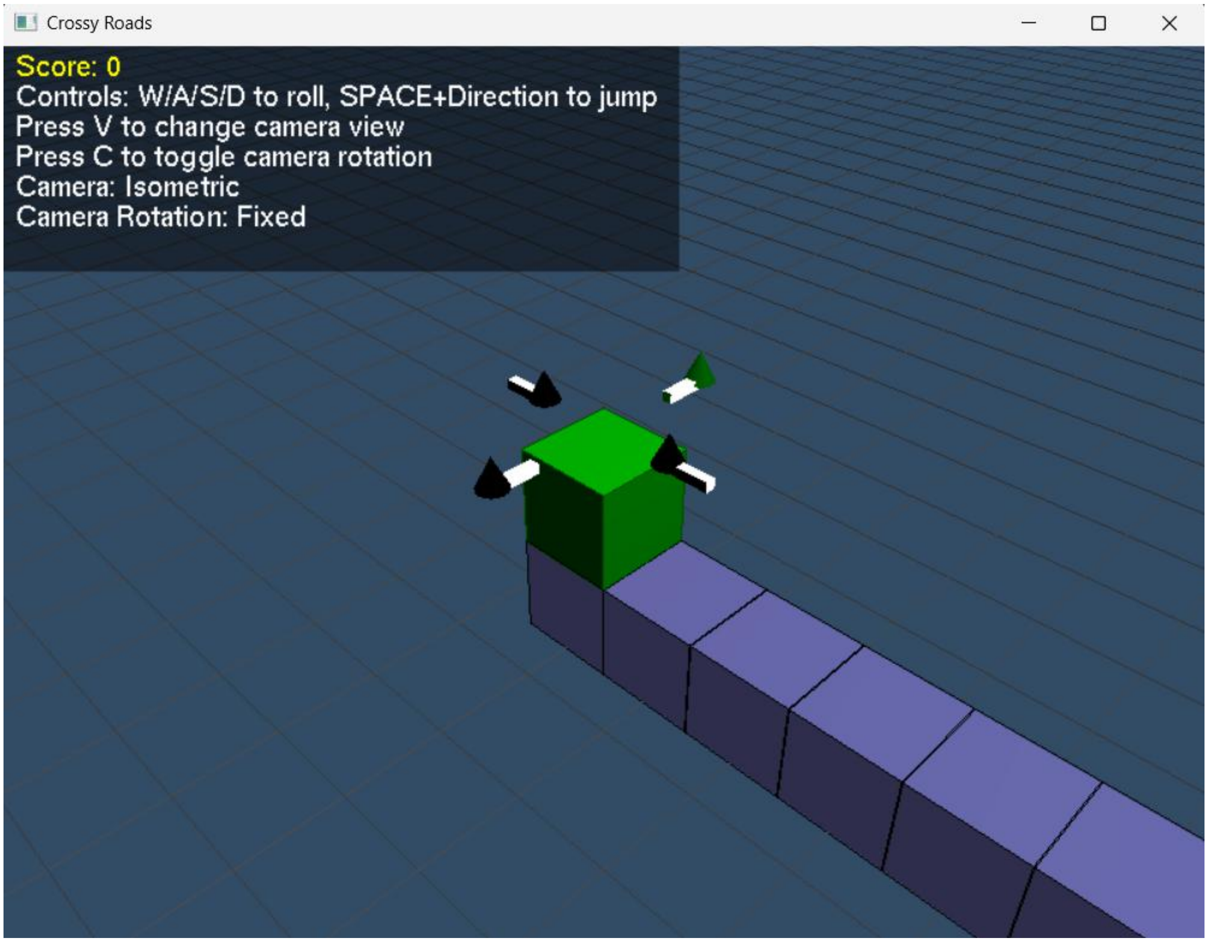
```
                canMove = true;
                newDirection = 4;
                showDirections = false;
            }

            if (canMove) {
                float targetX = playerX, targetZ = playerZ;

                switch (newDirection) {
                case 1: targetZ = playerZ - 1.0f; break;
                case 2: targetZ = playerZ + 1.0f; break;
                case 3: targetX = playerX - 1.0f; break;
                case 4: targetX = playerX + 1.0f; break;
                }

                if (keySpace) {
                    float jumpX = playerX, jumpZ = playerZ;

                    switch (newDirection) {
                    case 1: jumpZ = playerZ - 2.0f; break;
                    case 2: jumpZ = playerZ + 2.0f; break;
                    case 3: jumpX = playerX - 2.0f; break;
                    case 4: jumpX = playerX + 2.0f; break;
                    }

                    isJumping = true;
                    jumpStartX = playerX;
                    jumpStartZ = playerZ;
                    jumpDestX = jumpX;
                    jumpDestZ = jumpZ;
                    rollDirection = newDirection;
                }
                else {
                    isRolling = true;
                    rollDirection = newDirection;
                }
            }
        }

        if (!gameOver) {
            // Calculate current distance traveled (Manhattan distance)
            int currentDistance = static_cast<int>(playerX + playerZ);
            if (currentDistance > maxDistanceTraveled) {
                maxDistanceTraveled = currentDistance;
                score = maxDistanceTraveled;
            }
        }

        if (!fixedCameraAngle) {
            cameraAngle += deltaTime * 10.0f;
            if (cameraAngle > 360.0f) cameraAngle -= 360.0f;
        }
    }
    catch (const std::exception& e) {
        std::cerr << "Error in updateGame: " << e.what() << std::endl;
        gameOver = true;
    }
}

void reset() {
    try {
        score = 0;
```

```
            maxDistanceTraveled = 0;
            gameOver = false;
            isRolling = false;
            isJumping = false;
            jumpHeight = 0.0f;
            jumpProgress = 0.0f;
            rollAngle = 0.0f;
            rollDirection = 0;
            rollProgress = 0.0f;
            showDirections = true;
            prevDirection = -1;

            generateInitialPath();

            playerX = std::get<0>(path[0]);
            playerY = 1.0f;
            playerZ = std::get<1>(path[0]);

            keyW = keyS = keyA = keyD = keySpace = false;
        }
        catch (const std::exception& e) {
            std::cerr << "Error in reset: " << e.what() << std::endl;
            exit(1);
        }
    }

    void nextCameraMode() {
        cameraMode = (cameraMode + 1) % 4;
        if (cameraMode == 3) {
            fixedCameraAngle = true;
        }
    }

    void toggleCameraRotation() {
        fixedCameraAngle = !fixedCameraAngle;
    }

    void zoomIn() {
        cameraDistance = std::max(5.0f, cameraDistance - 1.0f);
    }

    void zoomOut() {
        cameraDistance = std::min(20.0f, cameraDistance + 1.0f);
    }
};

// The rest of the code remains the same...

Game game;
float lastFrameTime = 0.0f;

void displayText(float x, float y, const std::string& text, float r = 1.0f, float g = 1.0f, float b = 1.0f) {
    glColor3f(r, g, b);
    glRasterPos2f(x, y);
    for (char c : text) {
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18,  c);
    }
}

void drawTexturedCube(float x, float y, float z, float size, GLuint texture) {
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D,  texture);
    glPushMatrix();
    glTranslatef(x, y, z);
    glScalef(size, size, size);

    glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-0.5f, -0.5f, 0.5f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(0.5f, -0.5f, 0.5f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(0.5f, 0.5f, 0.5f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-0.5f, 0.5f, 0.5f);
    glEnd();

    glBegin(GL_QUADS);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-0.5f, -0.5f, -0.5f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-0.5f, 0.5f, -0.5f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(0.5f, 0.5f, -0.5f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(0.5f, -0.5f, -0.5f);
    glEnd();

    glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-0.5f, 0.5f, -0.5f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-0.5f, 0.5f, 0.5f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(0.5f, 0.5f, 0.5f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(0.5f, 0.5f, -0.5f);
    glEnd();

    glBegin(GL_QUADS);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-0.5f, -0.5f, -0.5f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(0.5f, -0.5f, -0.5f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(0.5f, -0.5f, 0.5f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-0.5f, -0.5f, 0.5f);
    glEnd();

    glBegin(GL_QUADS);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(0.5f, -0.5f, -0.5f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(0.5f, 0.5f, -0.5f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(0.5f, 0.5f, 0.5f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(0.5f, -0.5f, 0.5f);
    glEnd();

    glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-0.5f, -0.5f, -0.5f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-0.5f, -0.5f, 0.5f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-0.5f, 0.5f, 0.5f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-0.5f, 0.5f, -0.5f);
    glEnd();

    glPopMatrix();
    glDisable(GL_TEXTURE_2D);
}

void drawCube(float x, float y, float z, float size, float r, float g, float b, float alpha = 1.0f) {
    GLfloat mat_ambient[] = { r * 0.3f, g * 0.3f, b * 0.3f, alpha };
    GLfloat mat_diffuse[] = { r, g, b, alpha };
    GLfloat mat_specular[] = { 0.5f, 0.5f, 0.5f, alpha };
    GLfloat mat_shininess = 50.0f;

    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, mat_shininess);
    glPushMatrix();
    glTranslatef(x, y, z);
    glColor4f(r, g, b, alpha);
    glutSolidCube(size);

    if (alpha < 1.0f) {
        glColor4f(0.0f, 0.0f, 0.0f, alpha);
        glutWireCube(size * 1.01f);
    }
    glPopMatrix();
}

void drawArrow(float x, float y, float z, int direction) {
    glPushMatrix();
    glTranslatef(x, y, z);

    GLfloat mat_ambient[] = { 0.5f, 0.4f, 0.1f, 1.0f };
    GLfloat mat_diffuse[] = { 1.0f, 0.8f, 0.0f, 1.0f };
    GLfloat mat_specular[] = { 1.0f, 1.0f, 0.5f, 1.0f };
    GLfloat mat_shininess = 50.0f;

    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, mat_shininess);

    switch (direction) {
    case 1: glRotatef(180, 0, 1, 0); break;
    case 2: break;
    case 3: glRotatef(90, 0, 1, 0); break;
    case 4: glRotatef(-90, 0, 1, 0); break;
    }

    glPushMatrix();
    glScalef(0.1f, 0.1f, 0.4f);
    glutSolidCube(1.0f);
    glPopMatrix();

    glPushMatrix();
    glTranslatef(0, 0, 0.25f);
    glRotatef(-90, 1, 0, 0);
    glutSolidCone(0.15f, 0.3f, 16, 8);
    glPopMatrix();

    glColor3f(0.0f, 0.0f, 0.0f);
    char key;
    switch (direction) {
    case 1: key = 'W'; break;
    case 2: key = 'S'; break;
    case 3: key = 'A'; break;
    case 4: key = 'D'; break;
    default: key = ' '; break;
    }

    glRasterPos3f(0, 0.2f, 0);
    glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, key);

    glPopMatrix();
}

void drawObstacle(const Game::Obstacle& obstacle) {
    if (!obstacle.active) return;
    glPushMatrix();
    glTranslatef(obstacle.x + obstacle.offsetX, obstacle.height, obstacle.z + obstacle.offsetZ);

    // Disable color material tracking
    glDisable(GL_COLOR_MATERIAL);

    // Red material properties
    GLfloat mat_ambient[] = { 0.3f, 0.0f, 0.0f, 1.0f };
    GLfloat mat_diffuse[] = { 1.0f, 0.0f, 0.0f, 1.0f };
    GLfloat mat_specular[] = { 0.5f, 0.5f, 0.5f, 1.0f };
    GLfloat mat_shininess = 50.0f;

    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, mat_shininess);

    if (obstacle.type == Game::SPINNING_BLOCK) {
        glRotatef(obstacle.rotation, 0, 1, 0);
    }

    glutSolidCube(0.8f);

    // Draw wireframe without lighting
    glDisable(GL_LIGHTING);
    glColor3f(0.5f, 0.0f, 0.0f); // Dark red wireframe
    glutWireCube(0.81f);
    glEnable(GL_LIGHTING);

    glEnable(GL_COLOR_MATERIAL);
    glPopMatrix();
}

void drawPlayer() {
    glPushMatrix();
    // Disable color material tracking
    glDisable(GL_COLOR_MATERIAL);
    // Green color materials
    GLfloat mat_ambient[] = { 0.0f, 0.2f, 0.0f, 1.0f };  // Dark green
    GLfloat mat_diffuse[] = { 0.0f, 0.8f, 0.0f, 1.0f };  // Bright green
    GLfloat mat_specular[] = { 0.5f, 1.0f, 0.5f, 1.0f }; // Shiny green highlights
    GLfloat mat_shininess = 50.0f;

    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialf(GL_FRONT, GL_SHININESS, mat_shininess);

    if (game.isJumping) {
        glTranslatef(game.playerX, game.playerY + game.jumpHeight, game.playerZ);
        float rotationAngle = game.jumpProgress * 180.0f;

        switch (game.rollDirection) {
        case 1: glRotatef(-rotationAngle, 1.0f, 0.0f, 0.0f); break;
        case 2: glRotatef(rotationAngle, 1.0f, 0.0f, 0.0f); break;
        case 3: glRotatef(rotationAngle, 0.0f, 0.0f, 1.0f); break;
        case 4: glRotatef(-rotationAngle, 0.0f, 0.0f, 1.0f); break;
        }
    }
    else if (game.isRolling) {
        glTranslatef(game.playerX,  game.playerY,  game.playerZ);
        switch (game.rollDirection) {
        case 1:
            glTranslatef(0, -0.5f, -0.5f);
            glRotatef(-game.rollAngle, 1.0f, 0.0f, 0.0f);
            glTranslatef(0, 0.5f, 0.5f);
            break;
        case 2:
            glTranslatef(0, -0.5f, 0.5f);
            glRotatef(game.rollAngle, 1.0f, 0.0f, 0.0f);
            glTranslatef(0, 0.5f, -0.5f);
            break;
        case 3:
            glTranslatef(-0.5f, -0.5f, 0);
            glRotatef(game.rollAngle, 0.0f, 0.0f, 1.0f);
            glTranslatef(0.5f, 0.5f, 0);
            break;
        case 4:
            glTranslatef(0.5f, -0.5f, 0);
            glRotatef(-game.rollAngle, 0.0f, 0.0f, 1.0f);
            glTranslatef(-0.5f, 0.5f, 0);
            break;
        }
    }
    else {
        glTranslatef(game.playerX,  game.playerY,  game.playerZ);
    }

    // Main player cube
    glutSolidCube(game.CUBE_SIZE);

    // Dark green wireframe
    glDisable(GL_LIGHTING);
    glColor3f(0.0f, 0.3f, 0.0f);
    glutWireCube(game.CUBE_SIZE * 1.01f);
    glEnable(GL_LIGHTING);
    glEnable(GL_COLOR_MATERIAL);
    glPopMatrix();

    if (!game.isRolling && !game.isJumping && !game.gameOver && game.showDirections) {
        drawArrow(game.playerX, game.playerY + 0.7f, game.playerZ - 1.0f, 1);
        drawArrow(game.playerX, game.playerY + 0.7f, game.playerZ + 1.0f, 2);
        drawArrow(game.playerX - 1.0f, game.playerY + 0.7f, game.playerZ, 3);
        drawArrow(game.playerX + 1.0f, game.playerY + 0.7f, game.playerZ, 4);
    }
}

void drawSkybox() {
    glDisable(GL_LIGHTING);
    glColor3f(0.2f, 0.4f, 0.8f);

    glPushMatrix();
    glTranslatef(game.playerX, 0.0f, game.playerZ);
    glutSolidSphere(50.0f, 32, 32);
    glPopMatrix();

    glColor3f(1.0f, 1.0f, 1.0f);
    for (int i = 0; i < 10; i++) {
        glPushMatrix();
        glTranslatef(game.playerX + (i * 10 - 50), 15.0f, game.playerZ + (i % 3 * 10 - 15));
        glutSolidSphere(3.0f, 16, 16);
        glPopMatrix();
    }

    glEnable(GL_LIGHTING);
}

void drawGrid() {
    glBegin(GL_LINES);
    glColor3f(0.3f, 0.3f, 0.3f);
    for (int i = -50; i <= 50; i++) {
        float alpha = 1.0f - (abs(i) / 50.0f);
        glColor3f(0.3f * alpha, 0.3f * alpha, 0.3f * alpha);

        glVertex3f(i, -0.5f, -50);
        glVertex3f(i, -0.5f, 50);

        glVertex3f(-50, -0.5f, i);
        glVertex3f(50, -0.5f, i);
    }
    glEnd();
}

void display() {
    try {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();

        float camX, camY, camZ;
        float lookX, lookY, lookZ;
        float upX = 0, upY = 1, upZ = 0;

        float playerViewY = game.playerY + game.jumpHeight;

        switch (game.cameraMode) {
        case 0:
            camX = game.playerX + game.cameraDistance * cos(game.cameraAngle * M_PI / 180.0f);
            camY = playerViewY + game.cameraDistance * 0.7f;
            camZ = game.playerZ + game.cameraDistance * sin(game.cameraAngle * M_PI / 180.0f);
            lookX = game.playerX;
            lookY  =  playerViewY;
            lookZ = game.playerZ;
            break;
        case 1:
            camX = game.playerX;
            camY = playerViewY + game.cameraDistance;
            camZ = game.playerZ;
            lookX = game.playerX;
            lookY = playerViewY;
            lookZ = game.playerZ;
            upX = 1; upY = 0; upZ = 0;
            break;
        case 2:
            camX = game.playerX + game.cameraDistance;
            camY = playerViewY;
            camZ = game.playerZ;
            lookX = game.playerX;
            lookY = playerViewY;
            lookZ = game.playerZ;
            break;
        case 3:
            camX = game.playerX;
            camY = playerViewY + 3.0f;
            camZ = game.playerZ + 5.0f;
            lookX = game.playerX;
            lookY = playerViewY;
            lookZ = game.playerZ - 5.0f;
            break;
        }

        gluLookAt(camX, camY, camZ, lookX, lookY, lookZ, upX, upY, upZ);

        drawSkybox();
        drawGrid();

        for (auto& tile : game.path) {
            int x = std::get<0>(tile);
            int z = std::get<1>(tile);
            float life = std::get<2>(tile);
            float maxLife = std::get<3>(tile);

            if (life > 0.0f) {
                float alpha = life / maxLife;
                drawCube(x, 0.0f, z, game.CUBE_SIZE, 0.3f, 0.3f, 0.5f, alpha);

                glPushMatrix();
                glTranslatef(x, 0.0f, z);
                glColor4f(0.0f, 0.0f, 0.0f, alpha);
                glutWireCube(game.CUBE_SIZE * 1.01f);
                glPopMatrix();
            }
        }

        for (auto& obstacle : game.obstacles) {
            drawObstacle(obstacle);
        }

        if (!game.gameOver) {
            drawPlayer();
        }

        glMatrixMode(GL_PROJECTION);
        glPushMatrix();
        glLoadIdentity();
        gluOrtho2D(0, 800, 0, 600);
        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
        glLoadIdentity();

        glDisable(GL_DEPTH_TEST);  // Disable depth testing for UI elements
        glColor4f(0.0f, 0.0f, 0.0f, 0.5f);
        glBegin(GL_QUADS);
        glVertex2f(0, 600);
        glVertex2f(450, 600);
        glVertex2f(450, 450);
        glVertex2f(0, 450);
        glEnd();

        displayText(10, 580, "Score: " + std::to_string(game.score), 1.0f, 1.0f, 0.0f);

        if (game.gameOver) {
            displayText(300, 300, "Game Over! Press R to Restart", 1.0f, 0.0f, 0.0f);
        }
        else {
            displayText(10, 560, "Controls: W/A/S/D to roll, SPACE+Direction to jump", 1.0f, 1.0f, 1.0f);
            displayText(10, 540, "Press V to change camera view", 1.0f, 1.0f, 1.0f);
```

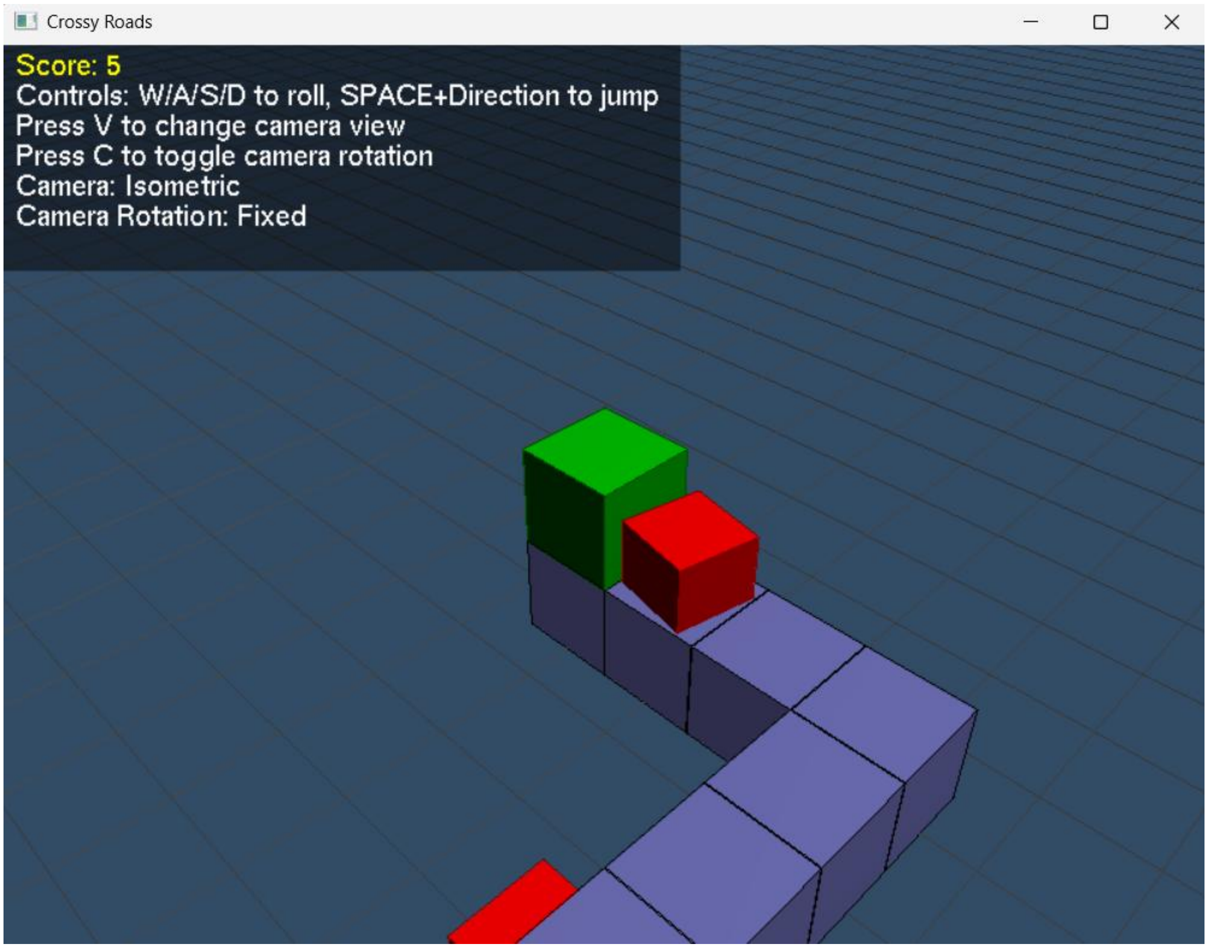
```
            displayText(10, 520, "Press C to toggle camera rotation", 1.0f, 1.0f, 1.0f);

            std::string camMode;
            switch (game.cameraMode) {
            case 0: camMode = "Isometric"; break;
            case 1: camMode = "Top-down"; break;
            case 2: camMode = "Side view"; break;
            case 3: camMode = "First-person"; break;
            }
            displayText(10, 500, "Camera: " + camMode, 1.0f, 1.0f, 1.0f);
            displayText(10, 480, "Camera Rotation: " + std::string(game.fixedCameraAngle ? "Fixed" :
"Rotating"), 1.0f, 1.0f, 1.0f);
        }
        glEnable(GL_DEPTH_TEST);
        glPopMatrix();
        glMatrixMode(GL_PROJECTION);
        glPopMatrix();
        glMatrixMode(GL_MODELVIEW);

        glutSwapBuffers();
    }
    catch (const std::exception& e) {
        std::cerr << "Error in display: " << e.what() << std::endl;
        exit(1);
    }
}

void timer(int) {
    try {
        float currentTime = glutGet(GLUT_ELAPSED_TIME) / 1000.0f;
        float deltaTime = currentTime - lastFrameTime;
        lastFrameTime = currentTime;

        if (deltaTime > 0.1f) deltaTime = 0.1f;

        game.updateGame(deltaTime);
        glutPostRedisplay();
        glutTimerFunc(16, timer, 0);
    }
    catch (const std::exception& e) {
        std::cerr << "Error in timer: " << e.what() << std::endl;
        exit(1);
    }
}

void reshape(int w, int h) {
    try {
        if (h == 0) h = 1;
        float ratio = 1.0f * w / h;
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glViewport(0, 0, w, h);
        gluPerspective(45.0f, ratio, 0.1f, 100.0f);
        glMatrixMode(GL_MODELVIEW);
    }
    catch (const std::exception& e) {
        std::cerr << "Error in reshape: " << e.what() << std::endl;
        exit(1);
    }
}

void keyboard(unsigned char key, int, int) {
    try {
        switch (key) {
        case 'w': case 'W': game.keyW = true; break;
        case 's': case 'S': game.keyS = true; break;
        case 'a': case 'A': game.keyA = true; break;
        case 'd': case 'D': game.keyD = true; break;
        case ' ': game.keySpace = true; break;
        case 'v': case 'V': game.nextCameraMode(); break;
        case 'c': case 'C': game.toggleCameraRotation(); break;
        case '+': case '=': game.zoomIn(); break;
        case '-': case '_': game.zoomOut(); break;
        case 'r': case 'R': game.reset(); break;
        case 27: exit(0); break;
        }
    }
    catch (const std::exception& e) {
        std::cerr << "Error in keyboard: " << e.what() << std::endl;
    }
}

void keyboardUp(unsigned char key, int, int) {
    try {
        switch (key) {
        case 'w': case 'W': game.keyW = false; break;
        case 's': case 'S': game.keyS = false; break;
        case 'a': case 'A': game.keyA = false; break;
        case 'd': case 'D': game.keyD = false; break;
        case ' ': game.keySpace = false; break;
        }
    }
    catch (const std::exception& e) {
        std::cerr << "Error in keyboardUp: " << e.what() << std::endl;
    }
}

void initGL() {
    try {
        const GLubyte* version = glGetString(GL_VERSION);
        if (!version) {
            throw std::runtime_error("OpenGL not properly initialized!");
        }
        std::cout << "OpenGL Version: " << version << std::endl;

        glEnable(GL_DEPTH_TEST);
        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        glClearColor(0.05f, 0.05f, 0.1f, 1.0f);
        glShadeModel(GL_SMOOTH);

        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
        glEnable(GL_COLOR_MATERIAL);

        GLfloat lightPos[] = { 10.0f, 15.0f, 10.0f, 1.0f };
        GLfloat ambientLight[] = { 0.4f, 0.4f, 0.4f, 1.0f };
        GLfloat diffuseLight[] = { 0.8f, 0.8f, 0.8f, 1.0f };
        GLfloat specularLight[] = { 1.0f, 1.0f, 1.0f, 1.0f };

        glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
        glLightfv(GL_LIGHT0, GL_AMBIENT, ambientLight);
        glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuseLight);
        glLightfv(GL_LIGHT0, GL_SPECULAR, specularLight);
        GLfloat fogColor[] = { 0.2f, 0.3f, 0.4f, 1.0f };
        glFogi(GL_FOG_MODE, GL_LINEAR);
        glFogfv(GL_FOG_COLOR, fogColor);
        glFogf(GL_FOG_DENSITY, 0.35f);
        glHint(GL_FOG_HINT, GL_DONT_CARE);
        glFogf(GL_FOG_START, 20.0f);
        glFogf(GL_FOG_END, 40.0f);
        glEnable(GL_FOG);
    }
    catch (const std::exception& e) {
        std::cerr << "Error in initGL: " << e.what() << std::endl;
        throw;
    }
}

int main(int argc, char** argv) {
    try {
        std::srand(static_cast<unsigned  int>(std::time(0)));

        glutInit(&argc, argv);
        if (argc < 1) {
            std::cerr << "GLUT initialization failed!" << std::endl;
            return -1;
        }

        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH | GLUT_ALPHA);
        glutInitWindowSize(800, 600);
        glutCreateWindow("Crossy Roads");

        if (!glutGetWindow()) {
            std::cerr << "Window creation failed!" << std::endl;
            return -1;
        }

        initGL();
        game.generateInitialPath();
        game.playerX = std::get<0>(game.path[0]);
        game.playerZ = std::get<1>(game.path[0]);
        game.playerY = 1.0f;

        glutDisplayFunc(display);
        glutReshapeFunc(reshape);
        glutKeyboardFunc(keyboard);
        glutKeyboardUpFunc(keyboardUp);
        glutTimerFunc(16, timer, 0);

        std::cout << "Game initialized successfully" << std::endl;
        glutMainLoop();
        return 0;
    }
    catch (const std::exception& e) {
        std::cerr << "Fatal error in main: " << e.what() << std::endl;
        return 1;
    }
}
```

();

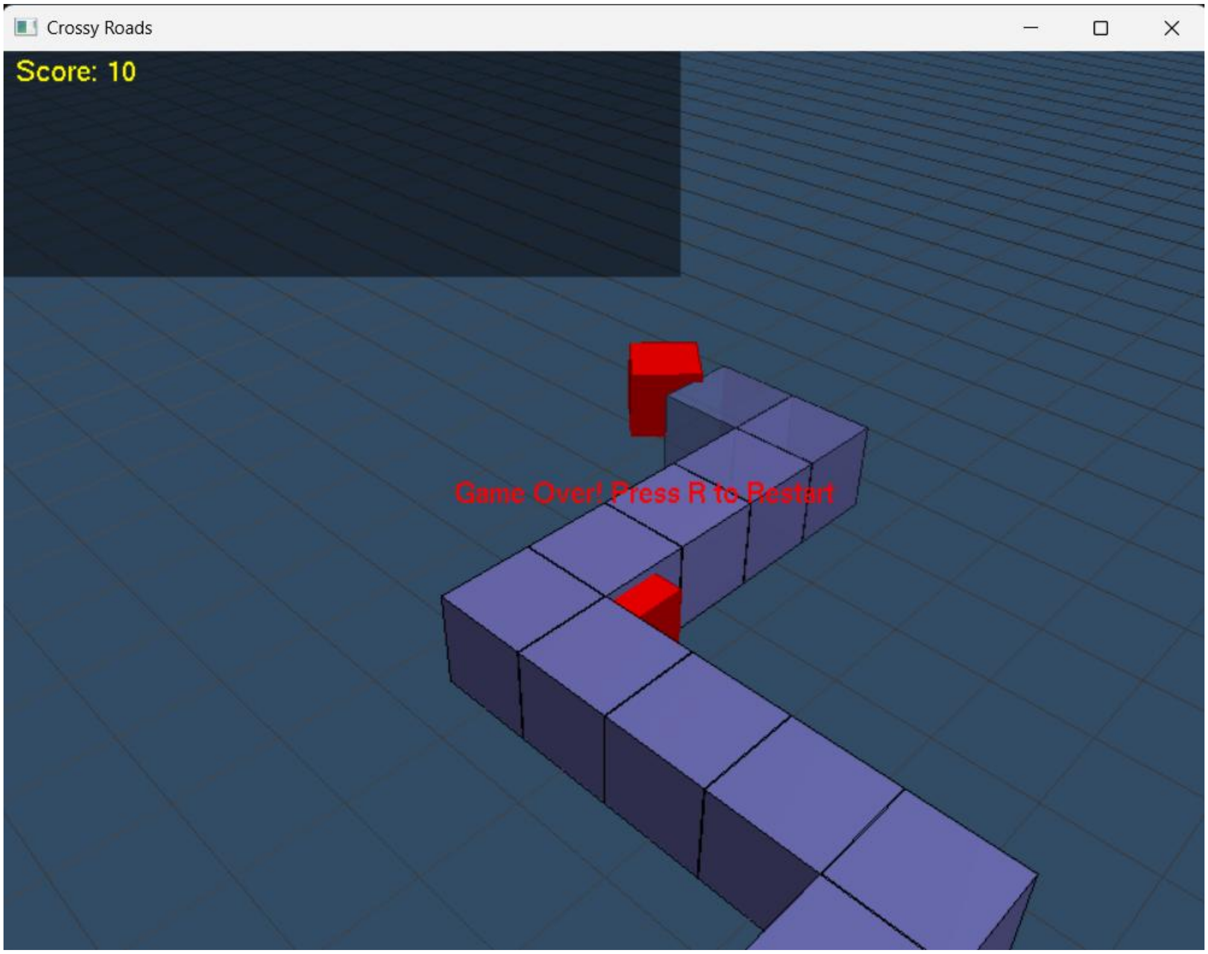
SCREENSHOTS



Score: 0
Controls: W/A/S/D to roll, SPACE+Direction to jump
Press V to change camera view
Press C to toggle camera rotation
Camera: Isometric
Camera Rotation: Fixed



Score: 5
Controls: W/A/S/D to roll, SPACE+Direction to jump
Press V to change camera view
Press C to toggle camera rotation
Camera: Isometric
Camera Rotation: Fixed



Game Over! Press R to Restart

<u>References</u> :

1. *https://nicolbolas.github.io/oldtut/index.html*
2. *https://ogldev.org/*