COMPUTER GRAPHICS

END TERM LAB FINAL ASSIGNMENT

SUBMITTED BY -

NAME – NABAJIT PAUL

ROLL NO – CSB22034

SUBJECT – COMPUTER GRAPHICS

Report :- 3D Ball Trajectory Simulation with Impact Point On The Ground

Objective :-

The goal of this project was to write a program to simulate and draw the trajectory of a ball, given the coordinates of the point where the ball makes an impact on the ground. The program needed to calculate the ball’s trajectory, including its motion under the influence of gravity, and display the path from the ball's starting point to the point of impact with the ground

Code Walkthrough :-

Let's break down the code into sections and explain each line:-

1. Including Libraries

```
#include <GL/glut.h>
#include <cmath>
#include <iostream>
#include <vector>
```

- **<GL/glut.h>**: This header file is used to access OpenGL functions for creating and managing windows, rendering 3D graphics, and handling user input.
- **<cmath>**: This library provides mathematical functions. We use it for trigonometric calculations (for camera rotation).
- **<iostream>**: Standard input-output stream library for user interaction (taking the user’s input for the bounce point).
- **<vector>**: We use the vector container to store the trajectory points of the ball.

2. **Global Variables**

```
// Ball parameters
float x = -4.0f, y = 1.5f, z = 0.0f; // Initial position of the ball in 3D space
float vx = 0.12f, vy = 0.32f, vz = 0.0f; // Initial velocities in x, y, z directions
float gravity = -0.02f;  // Acceleration due to gravity (negative for downward pull)
float damping = 0.7f;    // Energy loss factor on each bounce

// Ground level
const float groundLevel = 0.0f; // Height of the ground (y = 0)

// Timer for frame updates
int delay = 16; // Delay in milliseconds (~60 FPS)

// Store trajectory points
vector<pair<float, float>> trajectoryPoints; // To store the (x, y) coordinates of the ball's trajectory

// Camera variables
float cameraAngle =  1.0f;  // Camera rotation angle around the Y-axis (in radians)
float cameraRadius = 12.0f;   // Camera radius (distance from the center of the scene)
float cameraHeight = 6.0f;    // Camera height on the Y-axis
```

- These variables are used to store the state of the ball (position, velocity, gravity, and damping), the ground's position (groundLevel), the delay for animation updates, the ball's trajectory, and camera properties (for rotating the camera around the ball).

3. **Drawing the Ball**

```
// Function to draw the ball
void drawBall() {
    glPushMatrix();                         // Save the current transformation state
    glColor3f(1.0f, 0.0f, 0.0f);            // Set ball color to red
    glTranslatef(x, y, z);                  // Translate to the ball's position
    glutSolidSphere(0.15, 30, 30);           // Draw a sphere with radius 0.1
    glPopMatrix();                          // Restore the previous transformation state
}
```

- **glPushMatrix() and glPopMatrix()**: These functions save and restore the current transformation state, ensuring that transformations applied to the ball don't affect other objects.
- **glColor3f(1.0f, 0.0f, 0.0f)**: Sets the color of the ball to red.
- **glTranslatef(x, y, z)**: Translates the object to the ball's position in 3D space.
- **glutSolidSphere(0.15, 30, 30)**: Draws a solid sphere with a radius of 0.15 units (this represents the ball).

4. Drawing the Ground

```
// Function to draw the ground
void drawSmoothGround() {
    glPushMatrix();                          // Save the current transformation state
    void glPushMatrix(void) 0f);             // Set ground color to green

    // Draw smooth ground using a grid of quads
    float gridSize = 0.5f;                   // Size of each grid square
    glBegin(GL_QUADS);                       // Begin drawing quads
    for (float gx = -4.7f; gx < 4.7f; gx += gridSize) { // Loop through x-coordinates
        for (float gz = -1.5f; gz < 1.5f; gz += gridSize) { // Loop through z-coordinates
            glVertex3f(gx, groundLevel, gz);                 // Bottom-left vertex
            glVertex3f(gx + gridSize, groundLevel, gz);      // Bottom-right vertex
            glVertex3f(gx + gridSize, groundLevel, gz + gridSize); // Top-right vertex
            glVertex3f(gx, groundLevel, gz + gridSize);      // Top-left vertex
        }
    }
    glEnd();                                 // End drawing quads
    glPopMatrix();                           // Restore the previous transformation state
}
```

- **glColor3f(0.1f, 0.4f, 0.0f)**: Sets the ground color to green.
- **glBegin(GL_QUADS)**: Starts drawing quadrilaterals for the ground.
- **glVertex3f(gx, groundLevel, gz)**: Defines the four corners of each grid square.
- **glEnd()**: Ends the drawing of quads.

5. **Drawing the Trajectory**

```
// Function to draw the trajectory of the ball
void drawTrajectory() {
    if (trajectoryPoints.size() < 2) return; // If fewer than 2 points, no trajectory to draw

    glColor3f(1.0f, 1.0f, 0.0f);            // Set trajectory color to yellow
    glLineWidth(3.5f);                      // Set line width for the trajectory
    glBegin(GL_LINE_STRIP);                 // Begin drawing a continuous line
    for (auto& point : trajectoryPoints) { // Loop through each stored point
        glVertex3f(point.first, point.second, 0.0f); // Draw point in 3D space
    }
    glEnd();                                // End drawing the line
}
```

- **glColor3f(1.0f, 1.0f, 0.0f): Sets the trajectory color to yellow.**
- **glLineWidth(3.5f): Sets the width of the line to make the trajectory more visible.**
- **glBegin(GL_LINE_STRIP): Begins drawing a continuous line that connects the trajectory points.**
- **glVertex3f(point.first, point.second, 0.0f): Draws each point of the trajectory.**

6. Applying Camera Rotation

```
// Function to apply camera rotation around the ball in a 360-degree orbit
void applyCameraRotation() {
    // Calculate the camera's position based on an angle around the Y-axis
    float cameraX = cameraRadius * cos(cameraAngle);  // X position based on angle
    float cameraZ = cameraRadius * sin(cameraAngle);  // Z position based on angle

    // Set the camera to look at the center (the ball's position)
    glLoadIdentity();                         // Reset the model-view matrix
    gluLookAt(cameraX, cameraHeight, cameraZ,  // Camera position
              x, y, z,                       // Look-at point (ball's position)
              0.0f, 1.0f, 0.0f);             // Up direction (Y-axis)
}
```

- **gluLookAt(cameraX, cameraHeight, cameraZ, x, y, z, 0.0f, 1.0f, 0.0f): Positions the camera to orbit around the ball, constantly looking at the ball's position.**

7. **Main Display Function**

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

    applyCameraRotation();                          // Apply camera rotation

    drawSmoothGround();                         // Draw the ground
    drawTrajectory();                           // Draw the trajectory of the ball
    drawBall();                                 // Draw the ball

    glutSwapBuffers();                          // Swap buffers to display the rendered frame
}
```

- **glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)**: Clears the screen and depth buffer to prepare for the new frame.
- **glutSwapBuffers()**: Swaps the front and back buffers to display the newly rendered frame.

8. **Updating Ball's Position**

```
// Update function called periodically
void update(int value) {
    x += vx;                                    // Update x-position using velocity
    y += vy;                                    // Update y-position using velocity
    z += vz;                                    // Update z-position (not used here)

    vy += gravity;                              // Apply gravity to y-velocity

    trajectoryPoints.emplace_back(x, y);       // Add the current position to the trajectory

    // Check for collision with the ground
    if (y <= groundLevel + 0.1f && vx > 0) { // If the ball hits the ground
        y = groundLevel + 0.1f;                 // Adjust position to prevent sinking
        vy = -vy * damping;                     // Reverse and reduce y-velocity due to bounce

        if (fabs(vy) < 0.01f) {                 // If bounce is negligible
            vy = 0.0f;                          // Stop vertical motion
        }
    }

    // Stop the ball after it reaches a certain x-position
    if (x >= 5.0f) {                            // If the ball reaches the batsman
        vx = vy = vz = 0.0f;                    // Stop all motion
    }

    glutPostRedisplay();                        // Request a redraw
    glutTimerFunc(delay, update, 0);            // Schedule the next update

    cameraAngle += 0.005f; // Increment the camera angle for the next frame (continuous rotation)
    if (cameraAngle > 2 * M_PI) cameraAngle -= 2 * M_PI;  // Keep the angle between 0 and 360 degrees
}
```

- **x += vx; y += vy; z += vz;: Updates the ball's position based on its velocity.**
- **vy += gravity;: Applies the force of gravity to the ball's vertical velocity.**
- **trajectoryPoints.emplace_back(x, y);: Adds the ball's new position to the trajectory.**
- **Bounce logic: When the ball hits the ground, its vertical velocity is reversed, and damping is applied.**

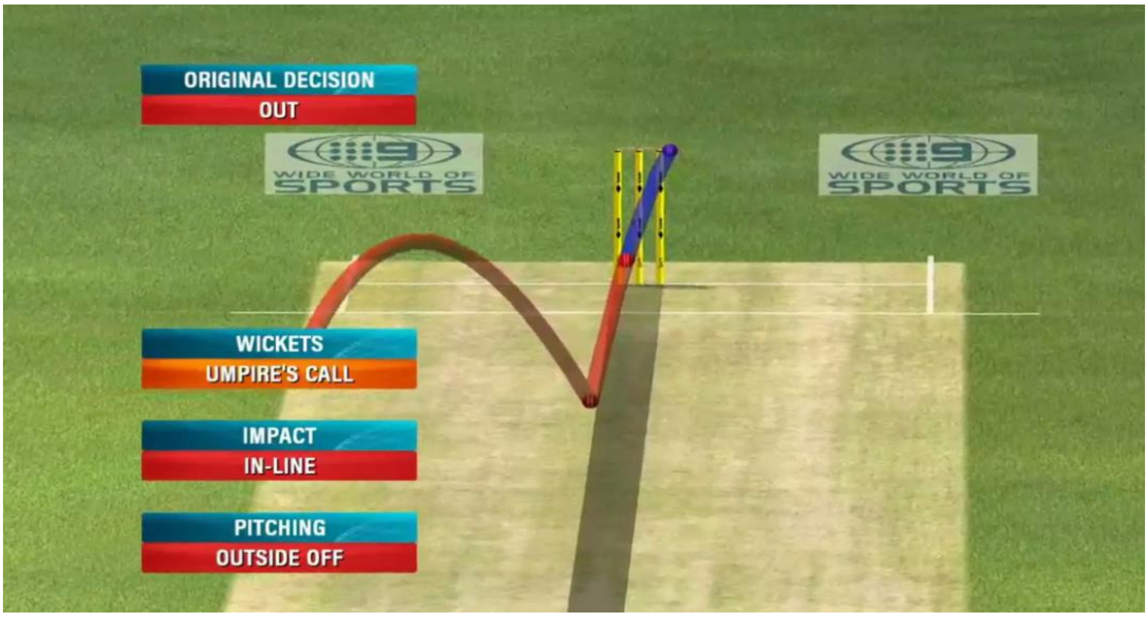
9. Main Function

```
// Main function
int main(int argc, char** argv) {
    cout << "Enter the x-coordinate where the ball should bounce(e.g.,between 2.0 and 5.0 for better bounce): ";
    float bounceX;                          // Take user input for bounce location
    cin >> bounceX;

    // Calculate velocity to hit the bounce point
    float distanceToBounce = bounceX - x;    // Distance from initial x to bounceX
    vx = distanceToBounce / 40.0f;           // Adjust velocity to reach bounce point

    glutInit(&argc, argv);                   // Initialize GLUT
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH); // Enable double buffering and depth
    glutInitWindowSize(800, 600);            // Set window size
    glutCreateWindow("3D Ball Trajectory with Camera Rotation"); // Create window

    init();                                  // Call initialization function

    glutDisplayFunc(display);                // Set display callback
    glutKeyboardFunc(keyboard);              // Register keyboard input callback
    glutTimerFunc(delay, update, 0);         // Set timer callback

    glutMainLoop();                          // Enter GLUT event processing loop
    return 0;                                // Return 0 on successful execution
}
```

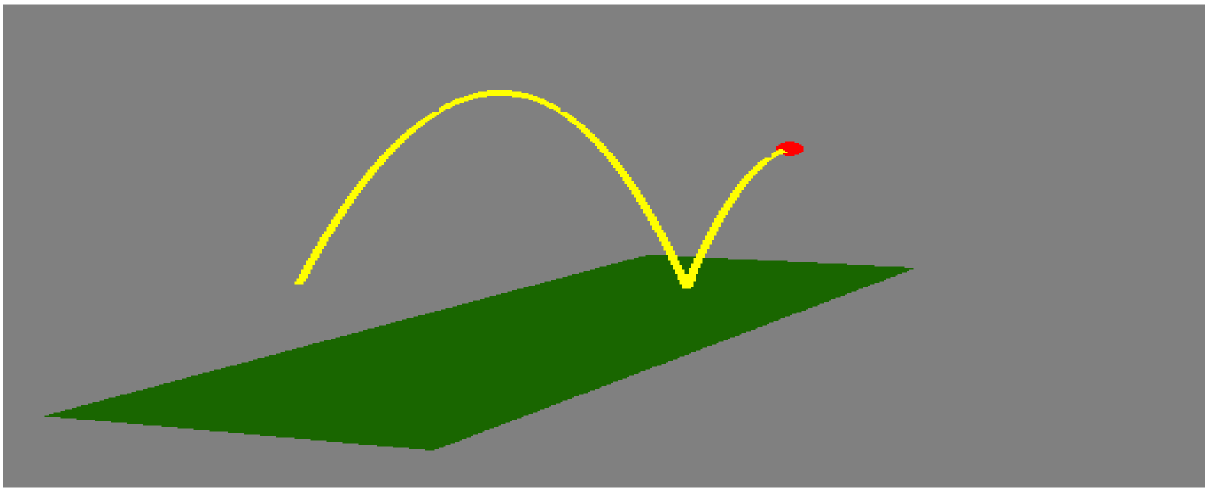
Conclusion

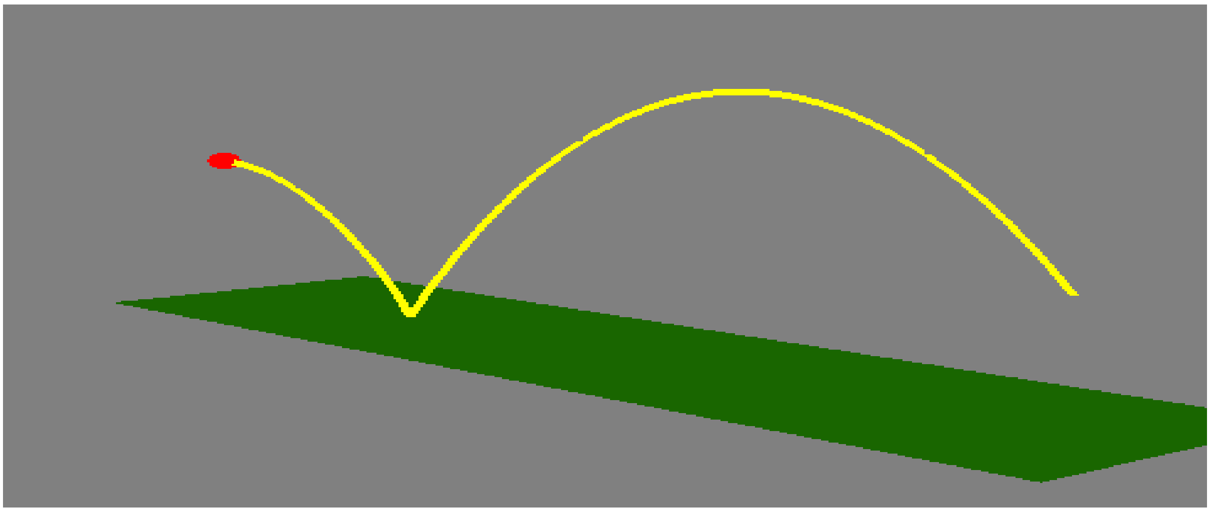
The code effectively simulates and visualizes the 3D trajectory of a ball. It calculates the ball's motion under the influence of gravity, handles user input for the impact point, and uses OpenGL for rendering the ball, ground, and trajectory. The camera is dynamically rotated around the ball to provide a better view of the simulation. The program uses simple physics principles, OpenGL rendering techniques, and event handling to create an interactive simulation that is both educational and visually engaging.

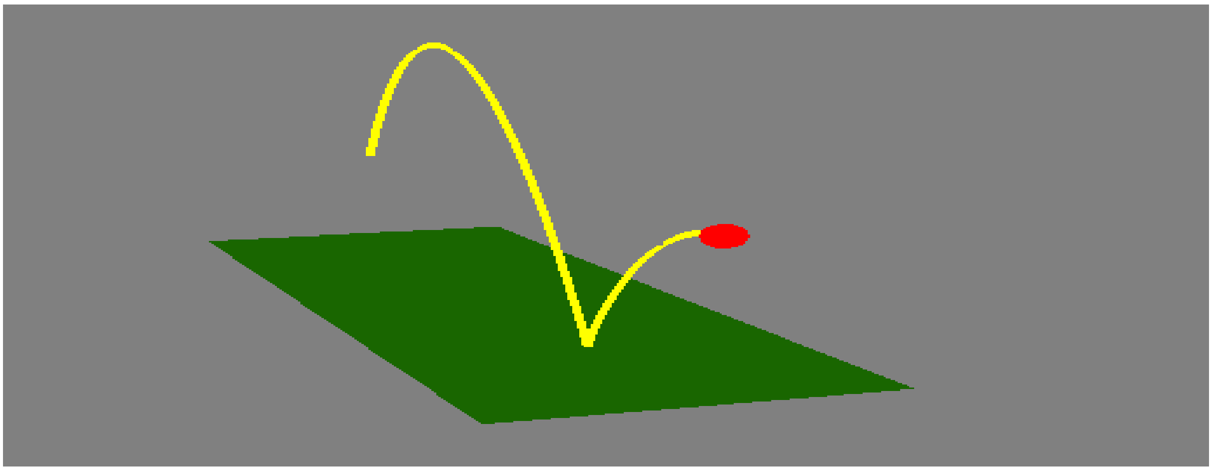
Reference Image Taken To Draw The Trajectory Of The Ball –

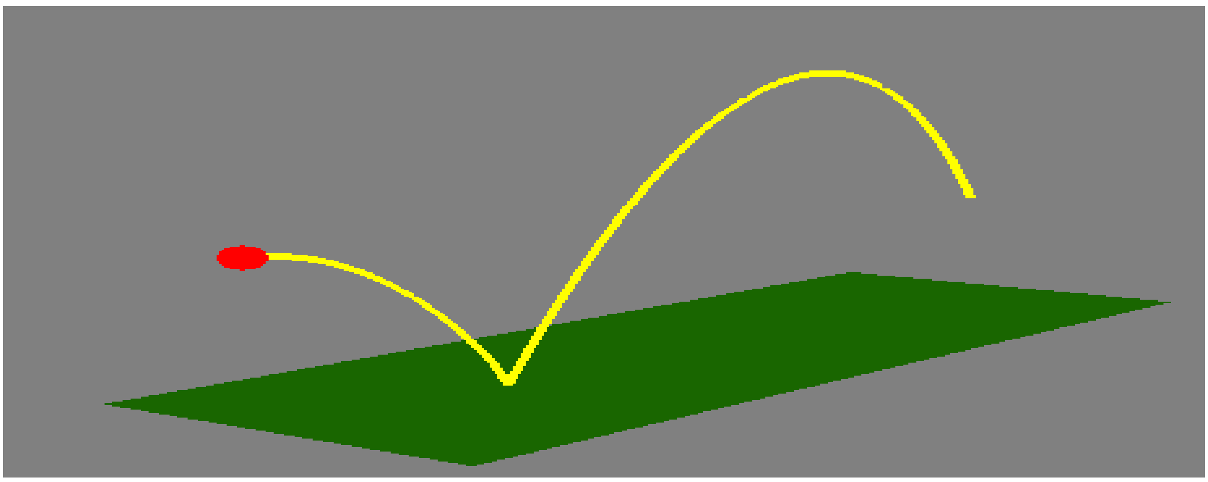


Output Model –









Steps Taken to Write the Code :-

1. **Understanding the Problem: The objective was to calculate and visualize the trajectory of a ball, considering gravity's impact on its motion. The ball's initial position, velocity, and impact point were key factors in determining its path. The program needed to plot the ball's trajectory in a 3D environment and calculate its motion until it reaches the ground at a specified point. The ball should bounce according to the laws of physics, and a rotating camera should provide a dynamic view of the scene.**
2. **Choosing Libraries and Tools:**
   - **OpenGL: This was used to render the 3D graphics of the ball, trajectory, and ground.**
   - **GLUT: A toolkit for managing the window and handling user input, such as camera zoom and simulation reset.**
   - **cmath: Used for performing mathematical operations, including trigonometric calculations for camera rotation.**
   - **iostream: Used for taking user input for the bounce point on the ground.**
3. **Setting Up the Environment:**
   - **I initialized the OpenGL environment, setting the window size and display mode.**
   - **The perspective projection was set up using gluPerspective() to provide a 3D view of the scene.**
   - **Depth testing was enabled to ensure that objects in 3D space are rendered correctly based on their distance from the camera.**
4. **Defining Ball Parameters:**
   - **The initial position of the ball was set at (x = -4.0f, y = 1.5f, z = 0.0f).**
   - **The initial velocities in the x, y, and z directions were defined as vx = 0.12f, vy = 0.32f, and vz = 0.0f.**
   - **The program also included a gravity value, gravity = -0.02f, to simulate the downward pull on the ball.**
   - **A damping factor of 0.7f was used to reduce the velocity after each bounce, simulating energy loss.**
5. **User Input for Impact Point:**
   - **I prompted the user to input the x-coordinate where the ball should make contact with the ground (e.g., between 2.0 and 5.0). This point was used to calculate the ball's required velocity to hit the specified point on the ground.**
   - **The distance between the ball's initial x-position and the input x-coordinate was calculated as distanceToBounce = bounceX - x.**

- The horizontal velocity vx was adjusted to ensure the ball reaches the specified impact point at the correct time. The formula used for this was vx = distanceToBounce / 40.0f.

6. Ball Motion and Trajectory Calculation:
   - The ball's position was updated every frame according to the equations of motion:
     - x += vx, y += vy, and z += vz.
     - Gravity was applied to the y-velocity by incrementing vy with the value of gravity.
   - The ball’s trajectory was stored as a series of (x, y) points in a vector<pair<float, float>> for later rendering.
7. Ball Impact and Bouncing:
   - The program checks if the ball has hit the ground by checking the condition if (y <= groundLevel + 0.1f), and if the ball does touch the ground, its vertical velocity vy is reversed and multiplied by the damping factor.
   - When the ball’s vertical velocity becomes negligible (fabs(vy) < 0.01f), vertical motion is stopped, and the ball remains at the ground level.
   - The program continues to simulate the ball's motion until it reaches the specified x-coordinate for impact (x >= 5.0f), at which point the ball's motion is stopped.
8. Trajectory Rendering:
   - The trajectory of the ball was drawn by connecting the stored points with a yellow line using glBegin(GL_LINE_STRIP).
   - This allowed a visual representation of the ball’s path as it moved from its initial position to the impact point on the ground.
9. Camera and User Controls:
   - A camera was set up to rotate around the ball, using the gluLookAt() function. The camera’s position was calculated based on a rotation angle around the y-axis.
   - The camera angle was continuously updated in each frame to create a smooth 360-degree rotation around the ball.
   - Users could zoom in and out with the 'w' and 's' keys to adjust the camera’s distance from the scene. A limit was imposed to prevent excessive zooming.
10. Display and Animation:
    - The display() function was used to render the scene, including the ball, the ground, and the trajectory.
    - The update() function was used to update the ball's position and velocity, calculate its trajectory, and check for impacts with the ground. The glutTimerFunc() function was used to repeatedly call update() at regular intervals (approximately every 16 milliseconds) to achieve smooth animation.

11. **Reset Function:**
    - **A reset feature was implemented using the 'r' key. This function reset the ball's position and velocity to the initial values, cleared the trajectory, and allowed the simulation to start over.**

Challenges and Solutions

- **Accurate Physics Calculations: One of the challenges was ensuring that the ball's velocity and trajectory were calculated correctly based on the user's input for the bounce point. By using basic kinematics and adjusting the horizontal velocity, I was able to ensure that the ball reached the correct impact point on the ground.**
- **Trajectory Visualization: Storing the ball's trajectory and rendering it as a continuous line in 3D space required careful management of points. The solution was to store only the (x, y) values and use glVertex3f() to plot the path.**
- **Collision Detection and Stopping the Ball: Properly detecting the collision with the ground and stopping the ball once its velocity became negligible was essential for realistic simulation. The ball's vertical velocity was reversed and reduced using the damping factor, and a threshold was set to stop the ball once its movement became negligible.**

Complete Code -

```
#include <GL/glut.h>

#include <cmath>

#include <iostream>

#include <vector>

using namespace std;


// Ball parameters

float x = -4.0f, y = 1.5f, z = 0.0f; // Initial position of the ball in 3D space

float vx = 0.12f, vy = 0.32f, vz = 0.0f; // Initial velocities in x, y, z directions

float gravity = -0.02f;  // Acceleration due to gravity (negative for downward pull)

float damping = 0.7f;    // Energy loss factor on each bounce


// Ground level

const float groundLevel = 0.0f; // Height of the ground (y = 0)


// Timer for frame updates

int delay = 16; // Delay in milliseconds (~60 FPS)


// Store trajectory points

vector<pair<float, float>> trajectoryPoints; // To store the (x, y) coordinates of the ball's trajectory
```

```
// Camera variables

float cameraAngle =  1.0f;  // Camera rotation angle around the Y-axis (in radians)

float cameraRadius = 18.0f; // Camera radius (distance from the center of the scene)

float cameraHeight = 6.0f;    // Camera height on the Y-axis


// Function to draw the ball

void drawBall() {

   glPushMatrix();                       // Save the current transformation state

   glColor3f(1.0f, 0.0f, 0.0f);          // Set ball color to red

   glTranslatef(x, y, z);                // Translate to the ball's position

   glutSolidSphere(0.15, 30, 30);         // Draw a sphere with radius 0.1

   glPopMatrix();                        // Restore the previous transformation state

}


// Function to draw the ground

void drawSmoothGround() {

   glPushMatrix();                       // Save the current transformation state

   glColor3f(0.1f, 0.4f, 0.0f);          // Set ground color to green


   // Draw smooth ground using a grid of quads

   float gridSize = 0.5f;                // Size of each grid square

   glBegin(GL_QUADS);                    // Begin drawing quads

   for (float gx = -4.7f; gx < 4.7f; gx += gridSize) { // Loop through x-coordinates

      for (float gz = -1.5f; gz < 1.5f; gz += gridSize) { // Loop through z-coordinates

         glVertex3f(gx, groundLevel, gz);               // Bottom-left vertex

         glVertex3f(gx + gridSize, groundLevel, gz);     // Bottom-right vertex

         glVertex3f(gx + gridSize, groundLevel, gz + gridSize); // Top-right vertex

         glVertex3f(gx, groundLevel, gz + gridSize);     // Top-left vertex

      }

   }

   glEnd();                          // End drawing quads

   glPopMatrix();                       // Restore the previous transformation state

}


// Function to draw the trajectory of the ball

void drawTrajectory() {

   if (trajectoryPoints.size() < 2) return; // If fewer than 2 points, no trajectory to draw
```

```
    glColor3f(1.0f, 1.0f, 0.0f);         // Set trajectory color to yellow
    glLineWidth(3.5f);                   // Set line width for the trajectory
    glBegin(GL_LINE_STRIP);              // Begin drawing a continuous line
    for (auto& point : trajectoryPoints) { // Loop through each stored point
        glVertex3f(point.first, point.second, 0.0f); // Draw point in 3D space
    }
    glEnd();                             // End drawing the line
}

// Function to apply camera rotation around the ball in a 360-degree orbit
void applyCameraRotation() {
    // Calculate the camera's position based on an angle around the Y-axis
    float cameraX = cameraRadius * cos(cameraAngle);  // X position based on angle
    float cameraZ = cameraRadius * sin(cameraAngle);  // Z position based on angle

    // Set the camera to look at the center (the ball's position)
    glLoadIdentity();                    // Reset the model-view matrix
    gluLookAt(cameraX, cameraHeight, cameraZ,  // Camera position
              x, y, z,                   // Look-at point (ball's position)
              0.0f, 1.0f, 0.0f);         // Up direction (Y-axis)
}

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

    applyCameraRotation();               // Apply camera rotation

    drawSmoothGround();                  // Draw the ground
    drawTrajectory();                    // Draw the trajectory of the ball
    drawBall();                          // Draw the ball

    glutSwapBuffers();                   // Swap buffers to display the rendered frame
}

// Update function called periodically
void update(int value) {
```

```
    x += vx;                        // Update x-position using velocity
    y += vy;                        // Update y-position using velocity
    z += vz;                        // Update z-position (not used here)

    vy += gravity;                  // Apply gravity to y-velocity

    trajectoryPoints.emplace_back(x, y);   // Add the current position to the trajectory

    // Check for collision with the ground
    if (y <= groundLevel + 0.1f && vx > 0) { // If the ball hits the ground
        y = groundLevel + 0.1f;         // Adjust position to prevent sinking
        vy = -vy * damping;             // Reverse and reduce y-velocity due to bounce

        if (fabs(vy) < 0.01f) {         // If bounce is negligible
            vy = 0.0f;                  // Stop vertical motion
        }
    }

    // Stop the ball after it reaches a certain x-position
    if (x >= 5.0f) {                    // If the ball reaches the batsman
        vx = vy = vz = 0.0f;            // Stop all motion
    }

    glutPostRedisplay();                // Request a redraw
    glutTimerFunc(delay, update, 0);    // Schedule the next update

    cameraAngle += 0.005f; // Increment the camera angle for the next frame (continuous rotation)
    if (cameraAngle > 2 * M_PI) cameraAngle -= 2 * M_PI;  // Keep the angle between 0 and 360 degrees
}

// Keyboard input function for camera control
void keyboard(unsigned char key, int x, int y) {
    switch (key) {
        case 'w': // Zoom in
            cameraRadius -= 0.5f;
            if (cameraRadius < 5.0f) cameraRadius = 5.0f; // Limit zoom
            break;
        case 's': // Zoom out
```

```
            cameraRadius += 0.5f;
            if (cameraRadius > 20.0f) cameraRadius = 20.0f; // Limit zoom
            break;
        case 'r': // Reset the simulation
            x = -4.0f; y = 1.5f; vx = 0.12f; vy = 0.32f;
            trajectoryPoints.clear();
            break;
        default:
            break;
    }
    glutPostRedisplay(); // Redraw after camera movement or reset
}

// Initialization function
void init() {
    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);    // Set background color to black
    glEnable(GL_DEPTH_TEST);                 // Enable depth testing
    glMatrixMode(GL_PROJECTION);             // Set matrix mode to projection
    glLoadIdentity();                        // Reset the projection matrix
    gluPerspective(45.0, 1.0, 1.0, 50.0);    // Set perspective projection
    glMatrixMode(GL_MODELVIEW);              // Switch back to model-view matrix
}

// Main function
int main(int argc, char** argv) {
    cout << "Enter the x-coordinate where the ball should bounce(e.g.,between 2.0 and 5.0 for better bounce): ";
    float bounceX;                           // Take user input for bounce location
    cin >> bounceX;

    // Calculate velocity to hit the bounce point
    float distanceToBounce = bounceX - x;    // Distance from initial x to bounceX
    vx = distanceToBounce / 40.0f;           // Adjust velocity to reach bounce point

    glutInit(&argc, argv);                   // Initialize GLUT
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH); // Enable double buffering and depth
    glutInitWindowSize(800, 600);            // Set window size
    glutCreateWindow("3D Ball Trajectory with Camera Rotation"); // Create window
```

```
    init();                                // Call initialization function

    glutDisplayFunc(display);              // Set display callback
    glutKeyboardFunc(keyboard);            // Register keyboard input callback
    glutTimerFunc(delay, update, 0);       // Set timer callback

    glutMainLoop();                        // Enter GLUT event processing loop
    return 0;                              // Return 0 on successful execution
}
```