This code simulates a continuously bouncing ball in a 2D environment using OpenGL and GLUT. Let's break it down step by step:

**1. Libraries and Global Variables:**

#include <GL/glut.h>

#include <cmath>

* **GLUT (OpenGL Utility Toolkit)** is a library used to create windows, handle user input, and manage events for OpenGL applications.
* **cmath** is used for mathematical functions like cosf, sinf, and constants like M\_PI (π).

**Global Variables:**

float x = -0.5f, y = 0.0f; // Ball position in 2D space

float vx = 0.001f, vy = 0.02f; // Ball velocity (horizontal and vertical)

const float gravity = -0.0005f; // Gravity force acting on the ball

const float radius = 0.05f; // Ball's radius

* The ball starts at the position (-0.5f, 0.0f).
* The initial velocities vx (horizontal) and vy (vertical) define the ball's speed and direction.
* gravity simulates the downward force (acts in the negative y direction).
* radius defines the size of the ball.

**2. Drawing the Ball:**

void drawBall(float x, float y) {

int segments = 100;

glBegin(GL\_TRIANGLE\_FAN); // OpenGL primitive to create a filled circle

glColor3f(1.0f, 0.2f, 0.2f); // Set color to red

glVertex2f(x, y); // Central point of the circle

for (int i = 0; i <= segments; i++) {

float angle = i \* 2.0f \* M\_PI / segments; // Angle for each segment

glVertex2f(x + cosf(angle) \* radius, y + sinf(angle) \* radius); // Circle points

}

glEnd();

}

* drawBall uses **GL\_TRIANGLE\_FAN** to draw a circle. The center is at (x, y) and 100 segments are used to approximate the circle.
* The color is set to red using glColor3f(1.0f, 0.2f, 0.2f).
* Each point on the circle is calculated using trigonometry (using cosf and sinf), and the circle is rendered by connecting these points to the center.

**3. Display Function:**

void display() {

glClear(GL\_COLOR\_BUFFER\_BIT); // Clear the screen

drawBall(x, y); // Draw the ball at the current position

glutSwapBuffers(); // Swap buffers to update the screen

}

* Clears the screen and redraws the ball at the updated position.
* glutSwapBuffers is used to swap the front and back buffers, displaying the rendered image on the screen.

**4. Update Function (Animation Logic):**

void update(int value) {

vy += gravity; // Apply gravity to vertical velocity

y += vy; // Update vertical position

x += vx; // Update horizontal position

// Bounce off bottom wall (y <= -1.0f + radius)

if (y - radius <= -1.0f) {

y = -1.0f + radius;

vy = -vy; // Reverse vertical velocity (bounce)

}

// Bounce off left and right walls (x >= 1.0f - radius or x <= -1.0f + radius)

if (x + radius >= 1.0f) {

x = 1.0f - radius;

vx = -vx; // Reverse horizontal velocity (bounce)

}

if (x - radius <= -1.0f) {

x = -1.0f + radius;

vx = -vx; // Reverse horizontal velocity (bounce)

}

glutPostRedisplay(); // Request redraw

glutTimerFunc(16, update, 0); // Call update every ~16ms (~60 FPS)

}

* **Gravity**: vy (vertical velocity) is continuously decreased by gravity, simulating the downward pull.
* **Position Update**: The x and y positions are updated based on the current velocities vx and vy.
* **Collision Detection**:
  + If the ball touches the bottom (y - radius <= -1.0f), its vertical velocity is reversed, simulating a bounce.
  + The same principle is applied for the left and right walls, reversing the horizontal velocity when the ball hits the sides.
* glutPostRedisplay triggers a redraw of the window, and glutTimerFunc(16, update, 0) calls the update function every 16 milliseconds to update the animation, aiming for approximately 60 frames per second.

**5. Initialization Function:**

void init() {

glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // Set background to black

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(-1.0, 1.0, -1.0, 1.0); // Set orthogonal projection (2D space)

}

* **Background Color**: The background is set to black using glClearColor.
* **Projection**: gluOrtho2D sets the 2D orthographic view, defining the range for x and y to be from -1.0 to 1.0.

**6. Main Function (Application Setup):**

int main(int argc, char\*\* argv) {

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB); // Double buffering and RGB color model

glutInitWindowSize(600, 600); // Set window size

glutCreateWindow("Continuous Bouncing Ball"); // Create window with title

init(); // Initialize OpenGL settings

glutDisplayFunc(display); // Register display callback

glutTimerFunc(0, update, 0); // Start update loop

glutMainLoop(); // Enter the GLUT main loop

return 0;

}

* Initializes GLUT, sets display mode, and creates a window.
* Registers the display function to render the scene and the update function for animation.
* Calls glutMainLoop, which enters the event-processing loop (handles input, redraws, etc.).

Here are some potential viva questions with answers related to the code, including possible modifications and enhancements that an examiner might suggest during the practical viva:

### 1. ****Q: How would you modify the ball’s behavior to simulate a more realistic bounce?****

**Answer:**  
To make the ball's bounce more realistic, you can implement energy loss during each bounce (also known as damping). This means the ball will lose some vertical velocity every time it hits the ground, eventually coming to a stop. To implement this, you can reduce the vertical velocity (vy) after every bounce.

**Modification:**

// Apply damping factor to vertical velocity on bounce

if (y - radius <= -1.0f) {

y = -1.0f + radius;

vy = -vy \* 0.9f; // Apply a damping factor (90% of the velocity)

}

This will make the ball lose 10% of its vertical speed after each bounce, simulating energy loss (e.g., due to friction or air resistance).

### 2. ****Q: How can you modify this program to handle multiple bouncing balls?****

**Answer:**  
To simulate multiple bouncing balls, you would need to store the position, velocity, and other properties (like radius) of each ball in an array or vector. Then, in the update and drawBall functions, you would loop over this array to update and render each ball.

**Modification:**

// Define a structure to store ball properties

struct Ball {

float x, y;

float vx, vy;

};

// Create an array or vector of balls

std::vector<Ball> balls = {

{-0.5f, 0.0f, 0.001f, 0.02f}, // Ball 1

{0.5f, 0.5f, -0.001f, -0.02f} // Ball 2

};

void update(int value) {

for (Ball &ball : balls) {

ball.vy += gravity; // Apply gravity

ball.y += ball.vy; // Update position

ball.x += ball.vx; // Update horizontal position

// Bounce logic for each ball

if (ball.y - radius <= -1.0f) {

ball.y = -1.0f + radius;

ball.vy = -ball.vy; // Bounce off the bottom

}

if (ball.x + radius >= 1.0f) {

ball.x = 1.0f - radius;

ball.vx = -ball.vx; // Bounce off the right wall

}

if (ball.x - radius <= -1.0f) {

ball.x = -1.0f + radius;

ball.vx = -ball.vx; // Bounce off the left wall

}

}

glutPostRedisplay();

glutTimerFunc(16, update, 0); // ~60 FPS

}

void display() {

glClear(GL\_COLOR\_BUFFER\_BIT);

for (Ball &ball : balls) {

drawBall(ball.x, ball.y); // Draw each ball

}

glutSwapBuffers();

}

This modification will handle multiple balls and make each one bounce independently.

### 3. ****Q: Can you make the ball change colors when it hits the wall?****

**Answer:**  
Yes, you can modify the drawBall function to change the color of the ball based on which wall it collides with. For example, we can change the ball's color to blue when it hits the left wall, red when it hits the right wall, and green when it hits the bottom.

**Modification:**

void drawBall(float x, float y, bool hitLeft, bool hitRight, bool hitBottom) {

int segments = 100;

glBegin(GL\_TRIANGLE\_FAN);

// Change color based on the wall hit

if (hitBottom) {

glColor3f(0.0f, 1.0f, 0.0f); // Green for bottom

} else if (hitLeft) {

glColor3f(0.0f, 0.0f, 1.0f); // Blue for left wall

} else if (hitRight) {

glColor3f(1.0f, 0.0f, 0.0f); // Red for right wall

} else {

glColor3f(1.0f, 0.2f, 0.2f); // Default red color

}

glVertex2f(x, y); // Central point of the circle

for (int i = 0; i <= segments; i++) {

float angle = i \* 2.0f \* M\_PI / segments; // Angle for each segment

glVertex2f(x + cosf(angle) \* radius, y + sinf(angle) \* radius); // Circle points

}

glEnd();

}

In the update function, track which wall the ball hits and pass this information to drawBall.

void update(int value) {

bool hitLeft = false, hitRight = false, hitBottom = false;

// Update position and check collisions

for (Ball &ball : balls) {

ball.vy += gravity;

ball.y += ball.vy;

ball.x += ball.vx;

if (ball.y - radius <= -1.0f) {

ball.y = -1.0f + radius;

ball.vy = -ball.vy;

hitBottom = true;

}

if (ball.x + radius >= 1.0f) {

ball.x = 1.0f - radius;

ball.vx = -ball.vx;

hitRight = true;

}

if (ball.x - radius <= -1.0f) {

ball.x = -1.0f + radius;

ball.vx = -ball.vx;

hitLeft = true;

}

drawBall(ball.x, ball.y, hitLeft, hitRight, hitBottom); // Draw with color change

}

glutPostRedisplay();

glutTimerFunc(16, update, 0); // ~60 FPS

}

### 4. ****Q: How would you simulate energy loss (damping) after each bounce?****

**Answer:**  
To simulate energy loss, you can apply a damping factor to the ball's velocity after every bounce. This would reduce the speed of the ball gradually, causing it to eventually come to rest.

**Modification:**

// Apply damping factor to vertical and horizontal velocities on bounce

if (y - radius <= -1.0f) {

y = -1.0f + radius;

vy = -vy \* 0.9f; // Apply damping to vertical velocity (90% of the original)

}

if (x + radius >= 1.0f || x - radius <= -1.0f) {

x = (x + radius >= 1.0f) ? 1.0f - radius : -1.0f + radius;

vx = -vx \* 0.9f; // Apply damping to horizontal velocity

}

The ball will now lose 10% of its velocity after every bounce, making it eventually slow down and stop.

### 5. ****Q: Can you implement a feature that lets the user control gravity while the simulation is running?****

**Answer:**  
Yes, you can allow the user to interact with the simulation and change gravity in real-time using keyboard input. For example, the user can press keys to increase or decrease gravity.

**Modification:**

void keyboard(unsigned char key, int x, int y) {

if (key == '+') {

gravity -= 0.0001f; // Increase gravity

}

if (key == '-') {

gravity += 0.0001f; // Decrease gravity

}

}

int main(int argc, char\*\* argv) {

// Other initialization code

glutKeyboardFunc(keyboard); // Register keyboard function

glutMainLoop();

}

This will allow the user to press + to increase gravity and - to decrease it.

These modifications cover typical exam viva questions that could test your understanding of physics simulation, OpenGL usage, and interactivity in a graphical application.