PREMIER UNIVERSITY, CHATTOGRAM

Department of Computer Science & Engineering

Project Report

On

"Brick Breaker Blitz"

SUBMITTED BY

Name: Sohela Showrin

ID: 2104010202199

Name: Adnan Hassan Nadim

ID: 2104010202200

Name: Rahin Toshmi Ohee

ID: 2104010202204

SUBMITTED TO

Salman Farsi

Lecturer

Department of Computer Science & Engineering

Premier University, Chattogram

2 March, 2025

Table of Contents

List of Figures

1 Introduction

The Brick Breaker Game is a classic arcade-style game developed using OpenGL and C++. The game challenges players to control a paddle at the bottom of the screen, preventing a bouncing ball from falling while breaking bricks positioned at the top. With simple yet engaging gameplay, players must use quick reflexes and strategic movement to clear all the bricks and advance through levels. The game includes multiple levels, each increasing in difficulty by adding more bricks and increasing the ball's speed. Players have a limited number of chances to keep the ball in play. The game also tracks the score, highest score, and previous scores, encouraging players to beat their own records. By utilizing OpenGL, the game efficiently handles rendering, physics, and user interactions, making it a great example of how computer graphics and game mechanics work together to create an interactive experience.

1.1 Purpose

The main purpose of developing this game is to create an engaging and interactive gaming experience while exploring the fundamentals of computer graphics. The project demonstrates how to use OpenGL to render objects, detect collisions, and implement smooth gameplay mechanics. This game is designed to be simple yet challenging, allowing players to test their reaction speed, precision, and strategic thinking. Additionally, it serves as a valuable learning project for those interested in game development, helping to understand

concepts like collision detection, animation, user input handling.

1.2 Motivation

This project is inspired by classic arcade games like Breakout and Arkanoid, which are simple to play but tough to master. These games are popular because they are easy to learn and fun to keep playing. By making this game, we get to explore things like graphics programming, physics-based movement, and how users interact with the game. Adding levels, a scoring system, and game-over conditions makes the game more exciting and competitive. The challenge of increasing difficulty keeps players engaged and motivated to improve. This project not only provides entertainment but also helps in learning important game development concepts in a fun and hands-on way.

2 Objectives

The primary objectives of the game are as follows:

1 To design an interactive game where players control a ball to hit and break bricks while keeping it from dropping off the screen.

2 To integrate the OpenGL graphics library for rendering smooth, dynamic visuals, including the ball, paddle, and brick structures.

3 To develop a system that tracks and displays player scores, fostering competition and improving the gaming experience.

4 To implement advanced collision detection algorithms to ensure realistic and precise interactions between the ball, paddle, and bricks.

5 To establish a continuous game loop that consistently updates the game's logic and renders graphics for a seamless gaming experience.

3 System Specifications

Here are the system specifications (hardware and software) to run the Brick Breaker game smoothly:

3.1 Hardware Specifications

- **Device:** Personal Computer or Laptop
- **Input Device:** Mouse (for controlling the paddle)

3.2 Software Specifications

- **Operating System:** Windows 7/8/10/11 (32-bit or 64-bit)
- **Development Environment:**
  - OpenGL for rendering graphics
  - A C++ compiler / MinGW
- **Libraries and Tools:**
  - OpenGL and C++ Standard Library
  - GLUT or FreeGLUT for handling windowing and input
  - IDE : Code::Blocks

4 Methodology and Features

4.1 Game Progress Flow

- **Starting the Game**: The game begins when the mouse is clicked. The ball then starts moving.
- **Ball Movement**: The ball bounces around the screen, interacting with the walls, the bar, and the bricks.
- **Destroying Bricks**: When the ball hits a brick, the brick breaks and adds points to the score.
- **Losing Chances**: If the ball falls off the screen, one chance is lost. The game ends when all chances are lost.
- **Level Up**: After all bricks in a level are broken, the next level begins. The game continues until all levels are completed or all chances are lost.

4.2 Interactive Game Elements

- **The Bar (Player Control)**: The bar is moved left and right to prevent the ball from falling off the screen. This can be done using the arrow keys or the mouse.
- **The Ball**: The ball moves on its own, and the bar is used to bounce it to hit the bricks.

- **Bricks**: Bricks are placed in the level. When the ball hits a brick, it breaks and scores points.
- **Feedback**: The game displays the score, remaining chances, and messages when levels are completed or the game is lost.

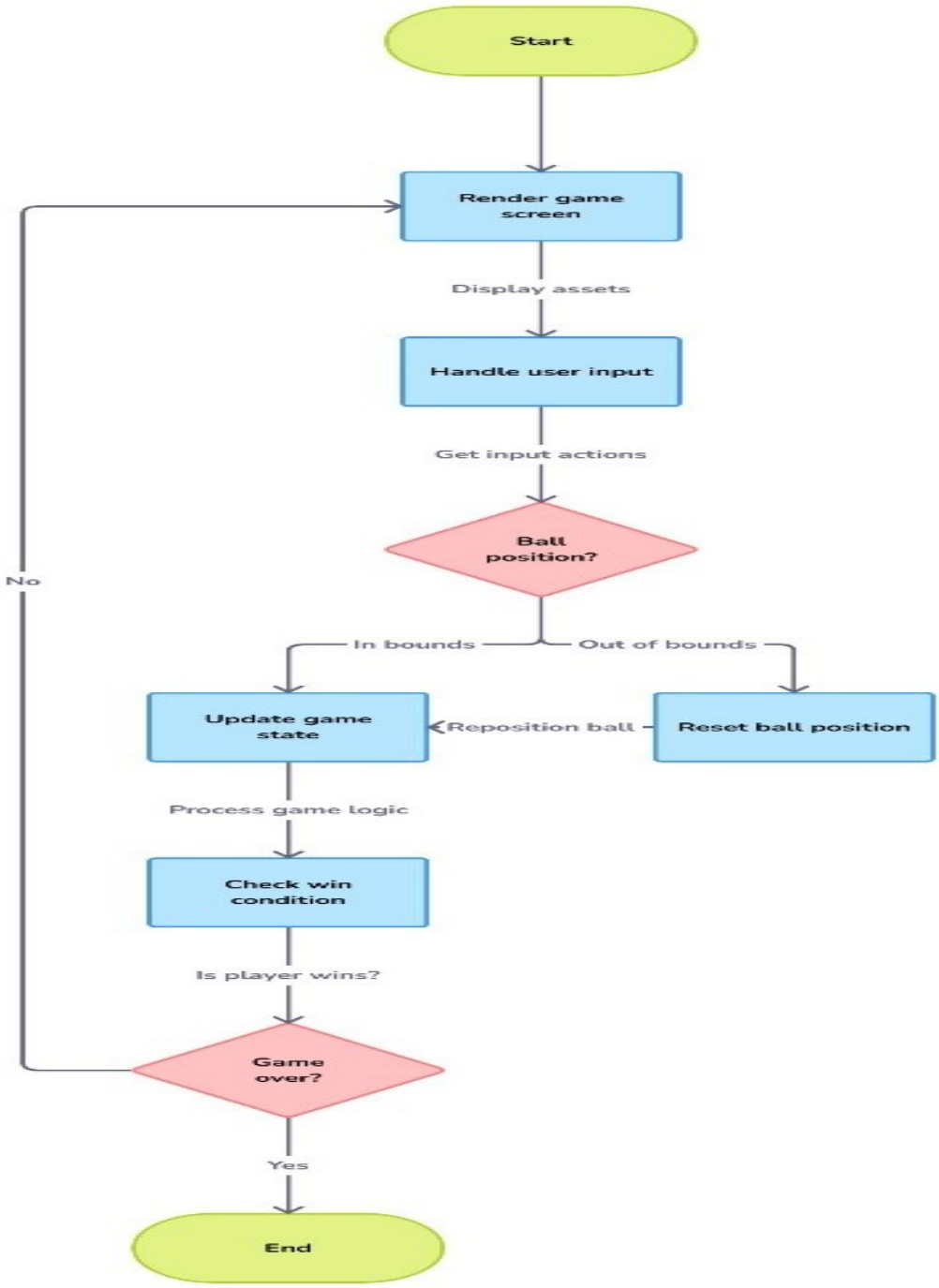

Figure 4.1: Flow Chart

4.3 Scoring Mechanism

- **Earning Points**: Points are earned every time a brick is broken by the ball.
- **Score Display**: The score is shown on the screen and updates as bricks are broken.
- **High Score**: The game keeps track of the highest score and displays it to encourage better performance.

4.4 Level Progression

- **Levels**: The game has multiple levels with increasing difficulty. Each level has more bricks and the ball moves faster.
- **Finishing a Level**: When all bricks are broken in a level, the next level begins.
- **Game End**: The game ends when all levels are completed or when all chances are lost.

4.5 Handling and Response

- **Player Input**: The game responds to the arrow keys or mouse movements to control the bar.
- **Ball and Bar**: When the ball hits the bar, its direction changes and it bounces off.
- **Brick Collision**: When the ball hits a brick, the brick breaks, and the ball moves in a new direction.
- **Game Over and Stage Completion**: If the ball falls off the screen, one chance is lost. If all bricks in a level are destroyed, the next stage begins.

4.6 Real-Time Game Interface

- **Updating the Screen**: The screen is updated continuously to show the ball, bar, bricks, and score.

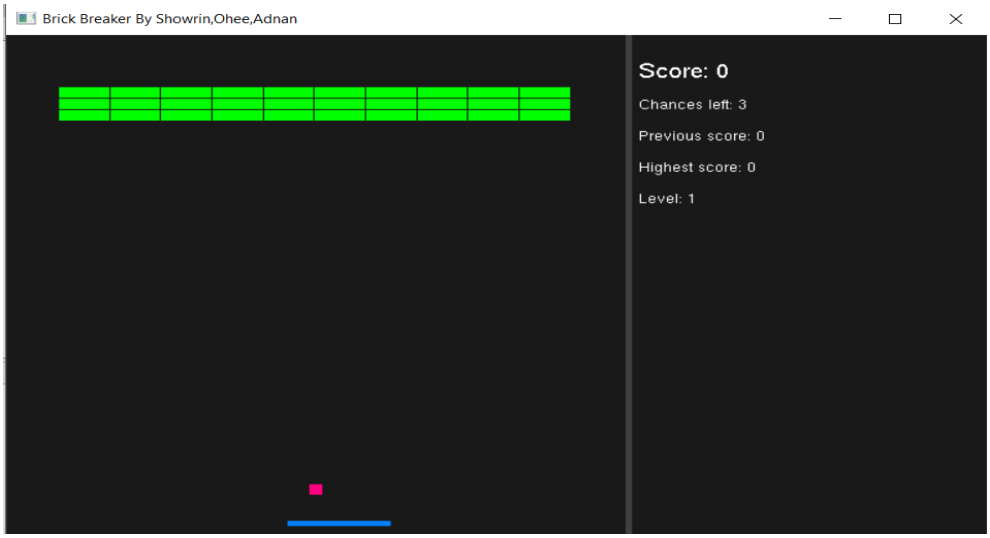

Figure 4.2: Starting interface with chances

Real-Time Updates: The score, ball position, and broken bricks are updated as the game progresses. Important details like the score, remaining chances, highest score, and current level are always visible.

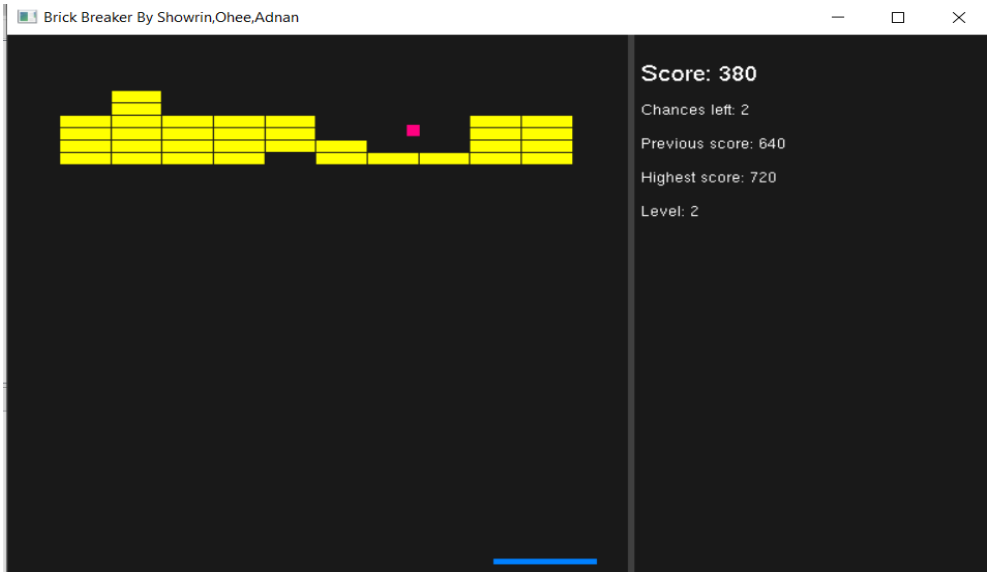

Figure 4.3: Upgradation of scores,level

Messages: Messages such as "Game Over" are displayed to inform about the game's status.

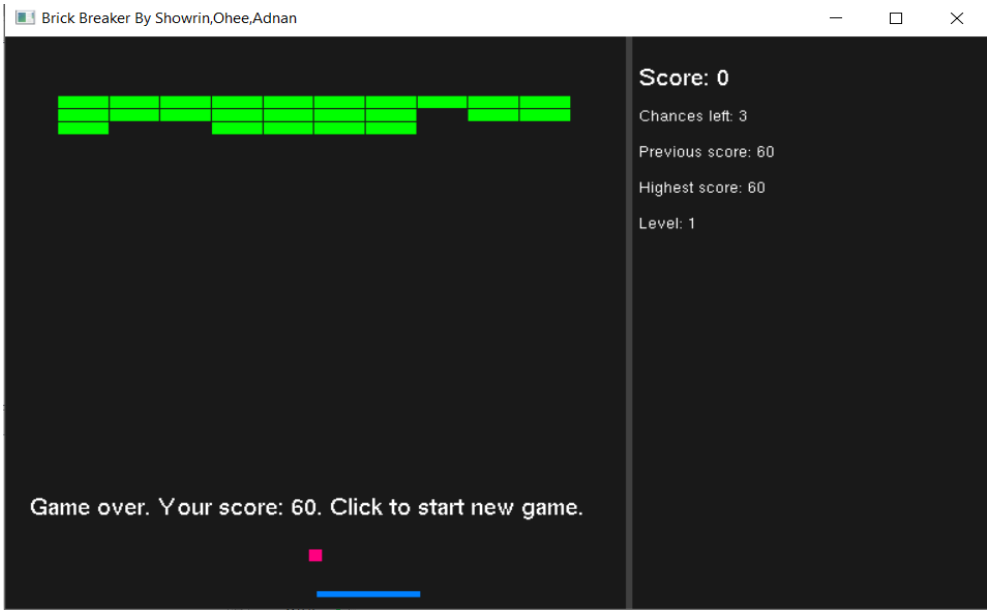

Figure 4.4: Game Over Interface

5 Implementation

5.1 Libraries & Global Variables

5.1.1 Libraries

```
#include <windows.h>
#include <stdio.h>
#include <iostream>
#include <GL/glut.h>
#include <string>
#include <sstream>

using namespace std;

float barX = 200, barY = 465, barWidth = 80, barheight = 5;
float ballX = 235, ballY = 430, ballWH = 10, ballVelX = 0.4, ballVelY = 0.4;
const int brickAmount = 100;
int score = 0, chances = 3, previousScore = 0, highestScore = 0;
bool flag = true, flag2 = true;
int currentLevel = 1;
const int maxLevels = 3;
int bricksPerLevel[maxLevels] = {30, 60, 100}; // Number of bricks for each level
float ballSpeedMultiplier[maxLevels] = {1.0, 1.5, 2.0}; // Ball speed multiplier for each level

struct bricks {
    float x;
    float y;
    float width;
    float height;
    bool isAlive = true;
};
bricks bricksArray[brickAmount];
```

Figure 5.1: Libraries Global Variables

The code begins by including essential libraries:

- **<windows.h>:** Provides access to Windows-specific functions, ensuring the game runs smoothly on this platform.
- **<stdio.h>and <iostream>:** Handle input and output operations, allowing the game to communicate with the player.
- **<GL/glut.h>:** The heart of the game's graphics, enabling the rendering of shapes, colors, and animations using OpenGL.
- **<string>and <sstream>:** Manage text and data formatting, making it easy to display scores, messages, and other information.

These libraries are like the artist's palette, providing the colors and brushes needed to paint the game's world.

5.1.2 Global Variables

The global variables define the core elements of the game, acting as its DNA:

- **Paddle (Bar):**
  - **barX, barY:** The paddle's position on the screen.
  - **barWidth, barheight:** The size of the paddle, determining how wide and thick it is.
- **Ball:**
  - **ballX, ballY:** The ball's position.
  - **ballWH:** The ball's size (width and height).
  - **ballVelX, ballVelY:** The ball's velocity, controlling its speed and direction.
- **Game State:**

  - **score:** Tracks the player's current score.
  - **chances:** The number of lives or attempts the player has.
  - **previousScore, highestScore:** Store historical data to motivate the player to beat their best.
  - **flag, flag2:** Boolean flags to control game states, such as pausing or resetting.

- **Levels and Difficulty:**
  - **currentLevel:** Tracks the current stage of the game.
  - **maxLevels:** The total number of levels.
  - **bricksPerLevel:** An array defining the number of bricks for each level, increasing the challenge as the player progresses.
  - **ballSpeedMultiplier:** Adjusts the ball's speed for each level, making the game faster and more intense.

- **Bricks:**
  - A struct named bricks defines each brick's properties:
    * **x, y:** Position.
    * **width, height:** Size.
    * **isAlive:** A boolean to track whether the brick is still in play.
    * **bricksArray:** An array of bricks, storing all the bricks in the game.

5.2 Brick Formation

```
void createBricks() {
    float brickX = 41, brickY = 50;
    int bricksToCreate = bricksPerLevel[currentLevel - 1];
    for (int i = 0; i < bricksToCreate; i++) {
        if (brickX > 400) {
            brickX = 41;
            brickY += 11;
        }
        bricksArray[i].x = brickX;
        bricksArray[i].y = brickY;
        bricksArray[i].width = 38.66;
        bricksArray[i].height = 10;
        bricksArray[i].isAlive = true; // Ensure bricks are alive when created
        brickX += 39.66;
    }
}
```

Figure 5.2: Brick Formation

1. Initial Setup

- **Starting Position:**
  - **brickX = 41, brickY = 50:** The initial coordinates for the first brick. These values determine where the brick grid begins on the screen.
- **Bricks to Create:**
  - **bricksToCreate = bricksPerLevel[currentLevel - 1]:** The number of bricks to generate depends on the current level.

    For example:

    Level 1: 30 bricks.

    Level 2: 60 bricks.

    Level 3: 100 bricks.

2. Brick Placement Logic

The function uses a loop to place each brick in the grid:

- **Row Completion Check:**
  - **If brickX >400, it means the current row is full, so the function:**
    - Resets brickX to 41 (starting X position).
    - Moves brickY down by 11 units to start a new row.
- **Brick Properties:**
  - **Each brick is assigned:**
    - **Position:** bricksArray[i].x = brickX, bricksArray[i].y = brickY.
    - **Size:** width = 38.66, height = 10.
    - **State:** isAlive = true (the brick is active and can be hit by the ball).
- **Increment Position:**After placing a brick, brickX is increased by 39.66 units to position the next brick beside it.

3. The Grid Pattern

The bricks are arranged in a grid pattern:

- **Rows and Columns:**
  - Bricks are placed horizontally until the row is full (brickX ¿ 400).
  - Then, the function moves to the next row by increasing brickY.
- **Spacing:**The spacing between bricks is determined by the increment in brickX (39.66 units) and brickY (11 units).

5.3 Stage Wise Brick Theme

The renderBricks() function is the artist of the game, bringing the bricks to life with vibrant colors that change based on the current level. It's not just about rendering bricks—it's about creating a visual journey for the player.

```
void renderBricks() {
    int bricksToRender = bricksPerLevel[currentLevel - 1];
    glBegin(GL_QUADS);
    for (int i = 0; i < bricksToRender; i++) {
        if (bricksArray[i].isAlive) {
            // Set brick color based on level
            if (currentLevel == 1) {
                glColor3ub(0, 255, 0); // Green bricks for level 1
            } else if (currentLevel == 2) {
                glColor3ub(255, 255, 0); // Yellow bricks for level 2
            } else if (currentLevel == 3) {
                glColor3ub(255, 0, 0); // Red bricks for level 3
            }
            // Render the brick
            glVertex2f(bricksArray[i].x, bricksArray[i].y);
            glVertex2f(bricksArray[i].x + bricksArray[i].width, bricksArray[i].y);
            glVertex2f(bricksArray[i].x + bricksArray[i].width, bricksArray[i].y + bricksArray[i].height);
            glVertex2f(bricksArray[i].x, bricksArray[i].y + bricksArray[i].height);
        }
    }
    glEnd();
}
```

Figure 5.3: Brick Formation

1. Setting the Stage

- **Bricks to Render:**
  - **bricksToRender = bricksPerLevel[currentLevel - 1]:** The number of bricks to display depends on the current level.
- **OpenGL Rendering:**
  - **glBegin(GL$_{QUADS}$)** : $Signals the start of rendering quadrilateral shapes (the bricks)$.
  - **glEnd():** Signals the end of rendering.

2. Level-Based Color Themes The function uses the currentLevel variable to determine the brick colors, creating a visual progression as the player advances:

- **Level 1: Green Bricks:**
  - **glColor3ub(0, 255, 0):** Bright green bricks symbolize the beginning of the journey—fresh, calm, and approachable.
- **Level 2: Yellow Bricks:**
  - **glColor3ub(255, 255, 0):** Yellow bricks represent the middle stage—energetic and challenging, urging the player to stay focused.
- **Level 3: Red Bricks:**
  - **glColor3ub(255, 0, 0):** Red bricks signify the final challenge—intense, urgent, and demanding the player's best effort.

3. Rendering Each Brick For each brick that is still alive (isAlive == true), the function:

- **Sets the Color:** Based on the current level.
- **Draws the Brick:**
  - **Uses glVertex2f() to define the four corners of the brick:**
    - **Bottom-left:** (bricksArray[i].x, bricksArray[i].y).
    - **Bottom-right:** (bricksArray[i].x + bricksArray[i].width, bricksArray[i].y).
    - **Top-right:** (bricksArray[i].x + bricksArray[i].width, bricksArray[i].y + bricksArray[i].height).
    - **Top-left:** (bricksArray[i].x, bricksArray[i].y + bricksArray[i].height).

5.4 Ball Motion Mechanics

The interaction between the ball and bricks is the core mechanic that drives the gameplay. It determines how the ball bounces, how bricks are destroyed, and how the player progresses through the levels.

```
void moveBall() {
    int bricksToCheck = bricksPerLevel[currentLevel - 1];
    bool allBricksBroken = true;
    for (int i = 0; i < bricksToCheck; i++) {
        if (bricksArray[i].isAlive) {
            allBricksBroken = false;
            break;
        }
    }

    if (allBricksBroken) {
        if (currentLevel < maxLevels) {
            currentLevel++;
            ballVelX *= ballSpeedMultiplier[currentLevel - 1];
            ballVelY *= ballSpeedMultiplier[currentLevel - 1];
            createBricks();
        } else {
            currentLevel++;
            ballVelX = 0;
            ballVelY = 0;
            congratulatoryMessage();
            glutPostRedisplay();
            return;
        }
    }
```

Figure 5.4: Ball Motion Mechanics

1. Collision Detection The checkCollision() function is the gatekeeper of interactions. It checks if two objects (like the ball and a brick) overlap by comparing their positions and sizes. If any of the following conditions are true, there is no collision:

- The ball is above the brick.
- The ball is below the brick.
- The ball is to the left of the brick.
- The ball is to the right of the brick.

If none of these conditions are met, the function returns true, indicating a collision. This simple yet powerful logic ensures that the ball interacts seamlessly with bricks and other

game elements.

```
ballX += ballVelX;
for (int i = 0; i < bricksToCheck; i++) {
    if (bricksArray[i].isAlive == true) {
        if (checkCollision(ballX, ballY, ballWH, ballWH, bricksArray[i].x, bricksArray[i].y, bricksArray[i].width, bricksArray[i].height) ==
            ballVelX = -ballVelX;
            bricksArray[i].isAlive = false;
            score += 10;
            break;
        }
    }
}
ballY -= ballVelY;
for (int i = 0; i < bricksToCheck; i++) {
    if (bricksArray[i].isAlive == true) {
        if (checkCollision(ballX, ballY, ballWH, ballWH, bricksArray[i].x, bricksArray[i].y, bricksArray[i].width, bricksArray[i].height) ==
            ballVelY = -ballVelY;
            bricksArray[i].isAlive = false;
            score += 10;
            break;
        }
    }
}
if (ballX < 0) {
    ballVelX = -ballVelX;
} else if (ballX + ballWH > 480) {
    ballVelX = -ballVelX;
}
if (ballY < 0) {
    ballVelY = -ballVelY;
} else if (ballY + ballWH > 480) {
    if (chances <= 1) {
        // Redirect to previous level
        if (currentLevel > 1) {
            currentLevel--;
```

Figure 5.5: Ball Motion Mechanics

2. Ball Motion Mechanics The moveBall() function handles the ball's movement and its interactions with bricks, walls, and the paddle.

- **Level Completion Check:**The function first checks if all bricks in the current level are broken. If they are, it progresses to the next level, increases the ball's speed, and regenerates the bricks. If all levels are completed, it displays a congratulatory message and stops the ball.

```
        if (chances <= 1) {
            // Redirect to previous level
            if (currentLevel > 1) {
                currentLevel--;
                ballVelX = 0.4 * ballSpeedMultiplier[currentLevel - 1];
                ballVelY = 0.4 * ballSpeedMultiplier[currentLevel - 1];
                createBricks();
            } else {
                // Reset game if already at level 1
                barX = 200;
                barY = 465;
                ballX = 235;
                ballY = 430;
                ballVelX = 0;
                ballVelY = 0;
                previousScore = score;
                if (highestScore < score) {
                    highestScore = score;
                }
                chances = 3;
                score = 0;
                flag = false;
                Sleep(1000);
                message(flag);
            }
        } else {
            chances--; // Decrease chances
            ballX = 235;
            ballY = 430;
            if (ballVelY < 0) {
                ballVelY = -ballVelY;
            }
            Sleep(1000);
        }
    }
    if (checkCollision(ballX, ballY, ballWH, ballWH, barX, barY, barWidth, barheight) == true) {
        ballVelY = -ballVelY;
    }
    glutPostRedisplay();
}
```

Figure 5.6: Ball Motion Mechanics

- **Ball Movement:**The ball's position is updated based on its velocity (ballVelX and ballVelY). This creates the illusion of continuous motion.
- **Brick Collision:**
  - **When the ball collides with a brick:**
    - The ball's direction is reversed (ballVelX or ballVelY is negated).

    * The brick is marked as isAlive = false, making it disappear.

    * The player's score increases by 10 points.

- **Wall Collision:**If the ball hits the left, right, or top walls, its direction is reversed to keep it within the game area.

  - **If the ball hits the bottom wall:** The player loses a chance (life).

  - If no chances are left, the game either resets or moves the player back to the previous level.

- **Paddle Collision:**When the ball hits the paddle, its vertical direction is reversed, sending it back into play.

3. Game Progression and Difficulty The game dynamically adjusts its difficulty based on the player's progress:

- **Level Advancement:**When all bricks in a level are destroyed, the player moves to the next level. The ball's speed increases, making the game more challenging.

- **Chances and Resets:**If the player loses all chances, the game either resets to Level 1 or moves the player back to the previous level, depending on the current level.

5.5 Ball & Brick Interaction

```
bool checkCollision(float aX, float aY, float aW, float aH, float bX, float bY, float bW, float bH) {
    if (aY + aH < bY)
        return false;
    else if (aY > bY + bH)
        return false;
    else if (aX + aW < bX)
        return false;
    else if (aX > bX + bW)
        return false;
    else
        return true;
}
```

Figure 5.7: Ball & Brick Interaction

The checkCollision() function is the gatekeeper of all interactions in the game. It determines whether two objects—like the ball and a brick—are colliding by comparing their positions and sizes. This function is essential for making the game feel responsive and realistic. The function takes
the positions and dimensions of two objects (Object A and Object B) as input. It then checks if these objects overlap by evaluating four conditions:

- If Object A is completely above Object B, there is no collision.
- If Object A is completely below Object B, there is no collision.
- If Object A is completely to the left of Object B, there is no collision.

- If Object A is completely to the right of Object B, there is no collision.

If none of these conditions are true, the objects must be overlapping, and the function returns true, indicating a collision. This simple yet powerful logic ensures that the ball interacts seamlessly with bricks, walls, and the paddle, creating a dynamic and engaging gameplay experience. Without this function, the game would lack the precision and responsiveness that make it fun and challenging. It's the invisible hand that guides every interaction, making the game world feel alive.

5.6 End Game Notification

```
void message(bool a) {
    if (a == false) {
        glColor3ub(255, 255, 255); // White text
        glRasterPos2f(20, 400);
        stringstream ss;
        ss << previousScore;
        string s = "Game over. Your score: " + ss.str() + ". Click to start new game.";
        int len = s.length();
        for (int i = 0; i < len; i++) {
            glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, s[i]);
        }
    }
}

void completeMessage(bool a) {
    if (a == false) {
        glColor3ub(255, 255, 255); // White text
        glRasterPos2f(20, 400);
        stringstream ss;
        ss << score;
        string s = "STAGE COMPLETE. Your score: " + ss.str() + ". Click to start new game.";
        int len = s.length();
        for (int i = 0; i < len; i++) {
            glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, s[i]);
        }
    }
}
```

Figure 5.8: End Game Notification

```
void congratulatoryMessage() {
    glColor3ub(255, 255, 0); // Yellow text
    glRasterPos2f(20, 400);
    string s = "Congratulations! You have successfully completed all the levels.";
    int len = s.length();
    for (int i = 0; i < len; i++) {
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, s[i]);
    }
}
```

Figure 5.9: End Game Notification

The end game notification system is the emotional core of the game, providing feedback to the player when they either lose, complete a level, or conquer the entire game. These messages are not just text on the screen—they are the culmination of the player's efforts, celebrating their achievements or encouraging them to try again.

- The message() function displays a game over message in white text, showing the final score and prompting the player to start a new game. It turns defeat into motivation.
- The completeMessage() function appears when a level is completed. It congratulates the player, shows their score, and invites them to the next challenge, rewarding their progress.
- The congratulatoryMessage() function celebrates the player's success when all levels

are completed. Written in yellow text, it marks the end of their journey with a sense of accomplishment.

5.7 Mouse Based Interaction

```
void mouse(int button, int state, int x, int y) {
    switch (button) {
        case GLUT_LEFT_BUTTON:
            if (state == GLUT_DOWN) {
                flag = true;
                if (ballVelX <= 0 && ballVelY <= 0) {
                    ballVelX = 0.3;
                    ballVelY = 0.3;
                }
                glutIdleFunc(moveBall);
            }
            break;
        default:
            break;
    }
}
```

Figure 5.10: Mouse Based Interaction

The mouse() function handles mouse clicks to start or restart the game. When the left mouse button is pressed (GLUT_LEFT_BUTTON), it sets the game flag to true and initializes the ball's velocity if it's stationary. This triggers the moveBall() function, setting the game in motion.

5.8 Keyboard Based Interaction

```
void keyboard(int key, int x, int y) {
    switch (key) {
        case GLUT_KEY_LEFT:
            barX -= 50;
            if (barX < 0) {
                barX = 0;
            }
            glutPostRedisplay();
            break;
        case GLUT_KEY_RIGHT:
            barX += 50;
            if (barX + barWidth > 480) {
                barX = 480 - barWidth;
            }
            glutPostRedisplay();
            break;
        default:
            break;
    }
}
```

Figure 5.11: Keyboard Based Interaction

The keyboard() function allows the player to move the paddle using the arrow keys. Pressing the left arrow moves the paddle left, while the right arrow moves it right. The paddle's position is constrained to stay within the game area, ensuring it doesn't go off-screen.

5.9 Game Screen Renderer

```
void myDisplay(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0, 0.0, 0.0);

    // Bar
    glBegin(GL_QUADS);
    glColor3ub(0, 128, 255); // Blue bar
    glVertex2f(barX, barY);
    glVertex2f(barX + barWidth, barY);
    glVertex2f(barX + barWidth, barY + barheight);
    glVertex2f(barX, barY + barheight);
    glEnd();

    // Ball
    glBegin(GL_QUADS);
    glColor3ub(255, 0, 128); // Pink ball
    glVertex2f(ballX, ballY);
    glVertex2f(ballX + ballWH, ballY);
    glVertex2f(ballX + ballWH, ballY + ballWH);
    glVertex2f(ballX, ballY + ballWH);
    glEnd();

    // Sidebar
    glBegin(GL_QUADS);
    glColor3ub(64, 64, 64); // Dark gray sidebar
    glVertex2f(480, 0);
    glVertex2f(480, 480);
    glVertex2f(485, 480);
    glVertex2f(485, 0);
    glEnd();
```

Figure 5.12: Game Screen Renderer

The myDisplay() function is the visual engine of the game, responsible for rendering all elements on the screen. It starts by clearing the screen and setting the background color. Then, it draws the paddle (bar), the ball, and the sidebar using OpenGL's glBegin(GL_QUADS) and glVertex2f() functions, defining their shapes, positions, and colors.

```
    // Render bricks
    renderBricks();

    // Display score, chances, and other information
    print(score);
    message(flag);
    completeMessage(flag2);

    if (currentLevel > maxLevels) {
        congratulatoryMessage();
    }

    glutSwapBuffers();
}
```

Figure 5.13: Game Screen Renderer

Next, it calls renderBricks() to display the bricks, followed by print(score) to show the player's score, remaining chances, and other game information. If the game is over or a level is completed, it displays appropriate messages using message(), completeMessage(), or congratulatoryMessage().

Finally, glutSwapBuffers() updates the screen, ensuring smooth and flicker-free visuals.

6 Application

The Brick Breaker game is not just an entertainment tool but also serves practical purposes in education, skill development, and technology. Below are its key applications:

1. **Entertainment** The game can be a fun way to relax or pass the time. It's simple and easy to play, making it great for anyone looking for quick entertainment.

2. **Stress Relief** Playing the game helps people unwind after a long day, as it has no complicated rules and provides a calming experience with colorful visuals and smooth gameplay.

3. **Improving Hand-Eye Coordination** The game can help improve coordination between the player's eyes and hands. Players need to move the paddle to hit the ball and break the bricks.

4. **Mobile Gaming** The game can be made into a mobile app, allowing people to play anytime and anywhere, making it ideal for short breaks.

5. **Learning Tool** The game could be adapted to teach math, language, or other subjects. For example, each brick could represent a question or word that players need to solve or translate to break.

6. **Challenge and Motivation** The game has different levels of difficulty, which can challenge players to keep improving their skills and motivate them to reach higher scores.

7. **Team Building Activities** A multiplayer version can allow players to work together or compete, which could be used in team-building events or as a fun social game.

8. **Game Design Practice** Aspiring game developers can use Brick Breaker as a simple project to learn how to design games, implement physics, and create user interfaces.

9. **Data Visualization** Instead of traditional charts or graphs, you can use the brick-breaking game to represent data. For example, each brick could represent a data point, and breaking them could show changes over time.

10. **Fitness and Movement** The game could be played using physical movement (for example, using motion sensors or gestures), encouraging exercise while playing.

These are just a few ways the Brick Breaker game can be applied. It's a versatile game with potential for many fun and educational uses!

7 Future Scope

1. Enhanced Graphics: Introduce textures, animations, and particle effects to make the game visually richer and more engaging.

2. More Levels and Themes: Expand the game with additional levels, each featuring unique themes, brick patterns, and challenges.

3. Power-Ups and Special Abilities: Add power-ups like multi-ball, paddle size changes, or temporary invincibility to add depth to gameplay.

4. Sound and Music: Integrate sound effects for collisions, level completion, and background music to enhance the player experience.

5. Multiplayer Mode: Introduce a competitive or cooperative multiplayer mode, allowing players to challenge friends or work together to clear levels.

8 Limitations

1. Limited Levels: With only three levels, the game may feel short or repetitive for players looking for extended challenges.

2. Keyboard and Mouse Dependency: The game relies solely on keyboard and mouse inputs, making it less accessible for players who prefer touchscreen or controller-based controls.

3. No Sound Effects: The absence of audio feedback, such as collision sounds or background music, reduces the overall engagement and immersion.

4. Uncontrollable Ball Speed at Higher Levels: As the player progresses to higher levels, the ball's speed increases significantly due to the ballSpeedMultiplier. While this adds difficulty, it can make the game unpredictable and hard to control, especially for casual players. The rapid acceleration might lead to frustration rather than a satisfying challenge.

9 Conclusion

This brick-breaker game project, built using OpenGL and GLUT, is a functional and enjoyable implementation of a classic arcade game. The code effectively handles core mechanics like ball movement, collision detection, and level progression. However, it has some limitations, such as abrupt ball speed increases at higher levels, a fixed number of levels, and no sound effects or power-ups.

Despite these limitations, the project provides a strong foundation for future improvements. By adding features like gradual speed adjustments, power-ups, and more levels, it can evolve into a more polished and engaging game. In its current state, it's a great starting point for further development and demonstrates the potential for creating a fun and challenging experience.

Contribution

Report Contribution

Chapter	Showrin (2199)	Ohee (2204)	Adnan (2200)
Introduction, Purpose, Motivation	■		
Objectives			■
System Specification	■		
Methodology & Features	■	■	■
Implementation	■	■	■
Application	■		
Future Scope		■	
Limitations		■	
Conclusion		■	

Code Contribution

Code	Showrin (2199)	Ohee (2204)	Adnan (2200)
Full source code	■	■	■