COMPUTER GRAPHICS LAB REPORT (GAME PROJECT)

Name	Roll No.	Class	Section
Pavel Das	002310501037	BCSE III	A2
Sandip Naskar	002310501038	BCSE III	A2
Arka Dutta	002310501039	BCSE III	A2
Saksam Saraff	002310501040	BCSE III	A2

Objective / Aim

The objective of this Computer Graphics lab project is to design and implement an interactive **Egg Catcher** game in a two-dimensional raster grid using the Qt framework in C++. The game simulates eggs falling from the top of the screen and a basket controlled by the player at the bottom of the screen.

In this updated version, advanced features such as **wind**, **focus mode** and a persistent **leaderboard** are added to make the gameplay more challenging, realistic and competitive.

The main aims are:

- To apply basic concepts of computer graphics such as drawing primitives, coordinate systems and rasterization to design the game elements on a raster grid.
- To use event-driven programming and timer-based animation to create smooth motion for falling eggs and the animated basket.
- To implement game mechanics such as scoring, lives, egg types (normal, bad, life), wind-based horizontal drift, focus mode and difficulty scaling.
- To implement high score persistence and a leaderboard that stores and displays top scores with player names across multiple runs and devices.

Contents

1. Introduction

Computer graphics deals with generating and manipulating visual content using computers. A classic introductory application of 2D computer graphics is the creation of small arcade-style games which use geometric shapes, raster grids and basic animation.

In this project, an **Egg Catcher** game is implemented where eggs fall from the top of the window and the player controls a basket at the bottom using the keyboard. The player must catch normal and life eggs while avoiding bad (negative) eggs. Each egg type affects score and lives differently.

The project is implemented using **Qt Creator** and the Qt framework in C++. Qt provides:

- A widget-based GUI framework with a main window class.
- A powerful painting system (`QPainter`) for drawing shapes and text.
- Timer support (`QTimer`) for driving the game loop.
- Event handling for keyboard input (`QKeyEvent`).
- Convenience classes for file handling (`QFile`, `QTextStream`), directories (`QDir`) and platform-specific paths (`QStandardPaths`).

The latest version of the game extends the original basic version with:

- A **menu screen** with player name input and buttons.
- A **leaderboard screen** that displays the top scores and player names.
- A **wind system** that randomly enables wind, pushing falling eggs sideways and adding difficulty.
- A **focus mode** which increases game intensity and multiplies scoring when the player reaches higher scores.

2. Problem Statement and Game Description

2.1 Problem Statement

Design and implement a two-dimensional Egg Catcher game in a raster grid using Qt and C++. The game must:

- Display falling eggs from predefined columns at the top of the grid.
- Allow the player to control a basket at the bottom using keyboard input.

- Implement different egg types: normal, bad and life eggs, each with their own effects on score and lives.
- Gradually increase difficulty via gravity, spawn rates and wind.
- Enter a special focus mode at high scores that changes visuals and the scoring system.
- Maintain a persistent per-device high score and a global leaderboard of top scores.

2.2 Game Rules and Gamification Elements

The rule set of the game is as follows:

- The player starts with **3 lives** and **score = 0**.
- **Normal eggs** (white) increase the score when caught.
- **Life eggs** (pink) increase the score and may grant extra lives (up to a maximum of 5 lives).
- **Bad eggs** (red) decrease lives and may decrease score.
- If a non-bad egg falls to the bottom without being caught, the player loses a life.
- When lives reach zero, the game enters the Game Over state.
- Above a certain score (50 points), a cyclic **focus mode** is activated where gameplay becomes faster and scoring is boosted.
- A high score is stored locally for the current device, and a leaderboard of top scores with player names is managed via a `LeaderboardManager`.

3. Tools and Technologies Used

3.1 Software and Libraries

- **Qt Creator IDE**: for designing the user interface, editing source files and building the project.
- **Qt Framework**: provides GUI widgets, painting, event handling, timers, file I/O and utility classes.
- **C++17**: core programming language used for game logic.

3.2 Graphics Concepts

- 2D raster grid and integer cell coordinates.
- Drawing primitives (rectangles, pixel blocks, ellipses, paths).
- Basic animation using timer-driven re-rendering.
- Simple particle systems for egg splats and wind dust.

4. System Design

4.1 Class Structure and Member Variables

The core logic is implemented in the `MainWindow` class, which derives from `QMainWindow`. The constructor uses a member-initializer list to set up game state, including the leaderboard manager, wind, focus mode and UI flags:

Listing 1: MainWindow constructor header and initializer list.

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent),
      ui(new Ui::MainWindow),
      leaderboardManager(), // Initialize LeaderboardManager
      gameTimer(nullptr),
      grid_box(6),
      highScore(0),
      grid_size(600),
      cols(0),
      rows(0),
      globalTime(0.0f),
      globalSpawnTimer(0.0f),
      spawnInterval(1.0f),
      lastEdgeSpawnTime(-100.0f),
      edgeSpawnCooldown(4.0f),
      currentColumnIndex(0),
      basket(0, 0),
      prevBasketX(0.0f),
      basketXVelocity(0.0f),
      basketTargetVel(0.0f),
      basketAccel(25.0f),
      basketMaxVel(50.0f),
      moveLeft(false),
      moveRight(false),
      fixedDelta(1.0f / 120.0f),
      accumulator(0.0f),
      gameOver(false),
      gameRunning(false),
      showMenu(true),            // Initial state: Menu
      showLeaderboard(false),    // Not showing leaderboard initially
      score(0),
      lives(3),
      flashAlpha(0.0f),
      flashFadeSpeed(2.5f),
      windActive(false),
      windStrength(0.0f),
      windTimer(0.0f),
      windCooldown(8.0f),
      timeSinceLastWind(0.0f),
      focusMode(false)
{
    QString dirPath =
        QStandardPaths::writableLocation(QStandardPaths::
           AppDataLocation);
    qDebug() << dirPath;
    ui->setupUi(this);
    setFocusPolicy(Qt::StrongFocus);
    // ...
}
```

4.2 Grid and Background Generation

The grid is sized from the `frame` widget and a green field background with grid lines is drawn using `QPainter`:

Listing 2: Grid and background initialization.

```
if (ui->frame) {
    grid_size = ui->frame->frameSize().width();
    if (grid_size <= 0)
        grid_size = 600;
}
loadHighScore(); // Load local high score for HUD
cols = qMax(40, grid_size / grid_box);
rows = qMax(30, grid_size / grid_box);

// background (simple green field)
```

```
background = QPixmap(grid_size, grid_size);
background.fill(QColor(48, 120, 48)); // nicer green
{
    QPainter g(&background);
    g.setPen(QPen(QColor(25, 100, 25, 80), 1));
    for (int i = 0; i <= cols; ++i)
        g.drawLine(i * grid_box, 0, i * grid_box, grid_size);
    for (int j = 0; j <= rows; ++j)
        g.drawLine(0, j * grid_box, grid_size, j * grid_box);
}
ui->frame->setPixmap(background);
```

The basket is initially placed near the bottom centre:

Listing 3: Initial basket position and sound setup.

```
score = 0;
lives = 3;
basket = QPointF(cols / 2.0f, rows - 3.0f);
prevBasketX = basket.x();

soundCatch.setSource(QUrl::fromLocalFile(
    "C:/Projects/EggCatcher/sfx/catch.wav"));
soundCatch.setVolume(0.8f);
soundLose.setSource(QUrl::fromLocalFile(
    "C:/Projects/EggCatcher/sfx/lose.wav"));
soundLose.setVolume(0.9f);
```

4.3 Drop Columns Layout

Eggs fall from four fixed lanes (columns) to give a lane-based gameplay style:

Listing 4: Drop column initialization.

```
dropColumns.clear();
int mid = cols / 2;
dropColumns = {mid - 25, mid - 5, mid + 5, mid + 25};
std::sort(dropColumns.begin(), dropColumns.end());

columnTimers.resize(dropColumns.size());
columnDelays.resize(dropColumns.size());
for (int i = 0; i < dropColumns.size(); ++i) {
    columnTimers[i] = 0.0f;
    columnDelays[i] = 3.0f +
        QRandomGenerator::global()->bounded(2.0f);
}
```

5. Persistent High Score and Device ID

5.1 Per-device Unique ID

To identify a device and ensure consistent leaderboard entries, the code creates and stores a unique device ID using `QUuid` in the `AppDataLocation` directory:

Listing 5: Generating and storing a device ID.

```
QString MainWindow::getDeviceID()
{
    QString path = QStandardPaths::writableLocation(
                       QStandardPaths::AppDataLocation)
                   + "/device_id.txt";

    QFile f(path);

    // Already exists → load
    if (f.exists() && f.open(QIODevice::ReadOnly | QIODevice::Text))
        {
        QString id = f.readAll().trimmed();
        f.close();
        if (!id.isEmpty())
            return id;
    }

    // Create new
    QString newID =
        QUuid::createUuid().toString(QUuid::WithoutBraces);

    if (f.open(QIODevice::WriteOnly | QIODevice::Text)) {
        f.write(newID.toUtf8());
        f.close();
    }

    return newID;
}
```

5.2 Local High Score Storage

The game stores the player name and local high score in a `player_info.txt` file under the app data directory:

Listing 6: Loading high score and player name.

```
void MainWindow::loadHighScore()
{
    QString dirPath =
        QStandardPaths::writableLocation(
            QStandardPaths::AppDataLocation);
    QDir().mkpath(dirPath);

    QString filePath = dirPath + "/player_info.txt";
    QFile file(filePath);

    if (!file.exists()) {
        playerName = "Player";
        highScore = 0;
        return;
    }

    if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
        return;

    QTextStream in(&file);
    playerName = in.readLine().trimmed();
    highScore = in.readLine().toInt();
    file.close();

    if (playerName.isEmpty())
        playerName = "Player";
}
```

Saving the high score is similarly straightforward:

Listing 7: Saving high score and player name.

```
void MainWindow::saveHighScore()
{
    QString dirPath =
        QStandardPaths::writableLocation(
            QStandardPaths::AppDataLocation);

    QDir().mkpath(dirPath);

    QString filePath = dirPath + "/player_info.txt";
    QFile file(filePath);

    if (!file.open(QIODevice::WriteOnly | QIODevice::Text)) {
        qDebug() << "FAILED TO CREATE FILE:" << filePath;
        return;
    }

    QTextStream out(&file);
    out << playerName << "\n" << highScore;
    file.close();

    qDebug() << "Saved highscore to:" << filePath;
}
```

6. Menu, Leaderboard and Game States

6.1 Menu UI Setup

The constructor creates four UI elements on top of the frame: a play button, leaderboard button, name input and a back-to-menu button.

Listing 8: Menu widgets and styling.

```
// --- MENU UI Setup ---
playButton = new QPushButton("PLAY", ui->frame);
leaderboardButton = new QPushButton("LEADERBOARD", ui->frame);
nameInput = new QLineEdit(ui->frame);
backToMenuButton = new QPushButton("Back to Menu", ui->frame);

// Set retro-looking styles
QString buttonStyle =
    "QPushButton { font-size: 20px; color: yellow; "
    "background-color: #2a2a2a; border: 2px solid white; "
    "padding: 10px; } "
    "QPushButton:hover { background-color: #444444; }";
QString inputStyle =
    "QLineEdit { font-size: 18px; color: white; "
    "background-color: #2a2a2a; border: 2px solid #555555; "
    "padding: 8px; }";
```

```

playButton->setStyleSheet(buttonStyle);
leaderboardButton->setStyleSheet(buttonStyle);
backToMenuButton->setStyleSheet(buttonStyle);
nameInput->setStyleSheet(inputStyle);
nameInput->setMaxLength(10);
nameInput->setText(playerName);

// Initial positioning
playButton->setGeometry(220, 370, 160, 50);
leaderboardButton->setGeometry(220, 430, 160, 50);
nameInput->setGeometry(200, 310, 200, 40);
backToMenuButton->setGeometry(220, 570, 160, 50);
```

Signals are connected to appropriate slots:

Listing 9: Menu button signal connections.

```
connect(playButton, &QPushButton::clicked,
        this, &MainWindow::startGameButtonClicked);
connect(leaderboardButton, &QPushButton::clicked,
        this, &MainWindow::showLeaderboardButtonClicked);
connect(backToMenuButton, &QPushButton::clicked, [this](){
    showLeaderboard = false;
    showMenu = true;
});
```

6.2 Menu Screen Rendering

The menu screen displays the title and the player name label, while showing the menu widgets and hiding score/lives labels.

Listing 10: Drawing the menu screen.

```
void MainWindow::drawMenu()
{
    QPixmap pix = background;
    QPainter p(&pix);
    p.setRenderHint(QPainter::Antialiasing, true);

    p.setPen(Qt::yellow);
    p.setFont(QFont("Comic Sans MS", 36, QFont::Bold));
    p.drawText(pix.rect().adjusted(0, -400, 0, 0),
               Qt::AlignCenter, "EGG CATCHER");
    p.setPen(Qt::white);
    p.setFont(QFont("Arial", 18));
    p.drawText(200, 280, "Player Name:");

    p.end();
    ui->frame->setPixmap(pix);

    nameInput->show();
    playButton->show();
    leaderboardButton->show();
    backToMenuButton->hide();
    ui->scoreLabel->hide();
    ui->livesLabel->hide();
}
```

6.3 Switching to Leaderboard Screen

When the leaderboard button is pressed, the game loads scores through the `leaderboardManager`. A loading spinner is shown while scores are being fetched, and then the top 5 entries are displayed.

Listing 11: Switching to leaderboard state.

```
void MainWindow::showLeaderboardButtonClicked() {

    showMenu = false;
    nameInput->hide();
    playButton->hide();
    leaderboardButton->hide();
    showLeaderboard = true;
    loadingLeaderboard = true;

    QTimer::singleShot(10, this, [this]() {
        leaderboardManager.loadScores();
        loadingLeaderboard = false;
    });
}
```

The `drawLeaderboard()` function first checks if loading is still in progress; if yes, a spinner and “Loading Leaderboard...” text are drawn. Once loading finishes, the scores are rendered:

Listing 12: Rendering the leaderboard screen.

```
void MainWindow::drawLeaderboard()
{
    QPixmap pix = background;
    QPainter p(&pix);
    p.setRenderHint(QPainter::Antialiasing, true);

    /* IF LOADING → SHOW SPINNER AND RETURN */
    if (loadingLeaderboard) {

        p.fillRect(pix.rect(), QColor(0, 0, 0, 150));

        p.setRenderHint(QPainter::Antialiasing, true);
        p.setPen(QPen(Qt::yellow, 6, Qt::SolidLine, Qt::RoundCap));

        int cx = pix.width() / 2;
        int cy = pix.height() / 2;
        int r  = 40;

        p.drawArc(cx - r, cy - r, r * 2, r * 2,
                  loaderAngle * 16, 120 * 16);

        loaderAngle += 10;
        if (loaderAngle >= 360) loaderAngle = 0;

        p.setPen(Qt::white);
        p.setFont(QFont("Arial", 20, QFont::Bold));
        p.drawText(0, cy + 80, pix.width(), 40,
                   Qt::AlignCenter, "Loading Leaderboard...");

        p.end();
        ui->frame->setPixmap(pix);
        return;
    }

    /* LOADING FINISHED → DRAW FULL LEADERBOARD */
    QVector<ScoreEntry> topScores = leaderboardManager.loadScores();

    p.setPen(Qt::cyan);
    p.setFont(QFont("Comic Sans MS", 30, QFont::Bold));
    p.drawText(pix.rect().adjusted(0, -500, 0, 0),
               Qt::AlignCenter, "TOP EGG CATCHERS");
    int yPos = 200;
    p.setFont(QFont("Arial", 20, QFont::Bold));
    p.setPen(Qt::white);

    p.drawText(150, yPos, "RANK");
    p.drawText(250, yPos, "SCORE");
    p.drawText(400, yPos, "NAME");
    yPos += 30;

    for (int i = 0; i < 5; ++i) {
        p.setPen(i == 0 ? Qt::yellow : Qt::white);
        p.setFont(QFont("Arial", 18));

        QString rank = QString::number(i + 1);
        QString scoreText = "---";
        QString nameText = "Empty";

        if (i < topScores.size()) {
            scoreText = QString::number(topScores[i].score);
            nameText = topScores[i].name;
        }

        p.drawText(150, yPos, rank);
        p.drawText(250, yPos, scoreText);
        p.drawText(400, yPos, nameText);
        yPos += 40;
    }

    p.end();
    ui->frame->setPixmap(pix);

    nameInput->hide();
    playButton->hide();
    leaderboardButton->hide();
    backToMenuButton->show();
    ui->scoreLabel->hide();
    ui->livesLabel->hide();
}
```

6.4 Keyboard Input and Game Reset

Keyboard input is used both for gameplay (A/D or Arrow keys) and for game state transitions (R to restart, M to go back to the menu):

Listing 13: Key press handling with menu/game over logic.

```cpp
void MainWindow::keyPressEvent(QKeyEvent *event)
{
    if (event->isAutoRepeat())
        return;

    if (gameOver && event->key() == Qt::Key_R) {
        resetGame();
        return;
    } else if (gameOver && event->key() == Qt::Key_M) {
        gameRunning = false;
        showMenu = true;
        nameInput->show();
        playButton->show();
        leaderboardButton->show();
        backToMenuButton->hide();
        return;
    }

    if (!gameRunning || gameOver)
        return;

    if (event->key() == Qt::Key_A || event->key() == Qt::Key_Left)
        moveLeft = true;
    else if (event->key() == Qt::Key_D || event->key() == Qt::
       Key_Right)
        moveRight = true;
}
```

Listing 14: Game reset routine.

```cpp
void MainWindow::resetGame()
{
    ui->scoreLabel->show();
    ui->livesLabel->show();
    score = 0;
    lives = 3;
    eggs.clear();
    basket = QPointF(cols / 2.0f, rows - 3.0f);
    prevBasketX = basket.x();
    basketXVelocity = 0.0f;
    basketTargetVel = 0.0f;
    gameOver = false;
    moveRight = false;
    moveLeft = false;
    accumulator = 0.0f;
    flashColor = QColor();
    frameClock.restart();
    gameRunning = true;
    showMenu = false;
    showLeaderboard = false;

    nameInput->hide();
    playButton->hide();
    leaderboardButton->hide();
    backToMenuButton->hide();

    gameTimer->start();

    windActive = false;
    windStrength = 0.0f;
    windTimer = 0.0f;
    timeSinceLastWind = 0.0f;
    focusMode = false;
}
```

When the game ends, `handleGameOver()` updates high score and submits scores to the leaderboard:

Listing 15: Handling game over and leaderboard submission.

```
void MainWindow::handleGameOver()
{
    highScore = std::max(score, highScore);

    saveHighScore();
    if (score >= 0) {
        QString deviceID = getDeviceID();
        qDebug() << playerName << " " << highScore;
        leaderboardManager.addScore(deviceID, playerName, highScore)
            ;
    }
}
```

7. Main Game Loop and Fixed Timestep

7.1 Timer-based Game Loop

The core game loop is driven by a `QTimer` that calls `gameTick()` approximately 60 times per second:

Listing 16: Timer setup in constructor.

```
// Timer
gameTimer = new QTimer(this);
gameTimer->setTimerType(Qt::PreciseTimer);
connect(gameTimer, &QTimer::timeout, this, &MainWindow::gameTick);
gameTimer->start(1000 / 60);  //  16.67 ms per frame

frameClock.start();
```

7.2 State-controlled `gameTick()`

The `gameTick()` method behaves like a high-level state machine:

Listing 17: Main gameTick function.

```
void MainWindow::gameTick()
{
    // State machine for screens
    if (showMenu) {
        drawMenu();
        return;
    }
    if (showLeaderboard) {
        drawLeaderboard();
        return;
    }

    if (gameOver) {
        handleGameOver();
        drawGameOver();
        return;
    }
```

```
// Elapsed real time
float dt = frameClock.restart() / 1000.0f;
dt = qBound(0.001f, dt, 0.05f);   // clamp: min 1ms, max 50ms
accumulator += dt;

// Run physics in fixed steps
const float fixedStep = fixedDelta;  // 1/120 s
while (accumulator >= fixedStep) {
    updatePhysics(fixedStep);
    accumulator -= fixedStep;
}

float alpha = accumulator / fixedStep;
drawGame(alpha);

const int targetFrameMS = 16; // ~60fps
int elapsed = frameClock.elapsed();
if (elapsed < targetFrameMS)
    QThread::msleep(targetFrameMS - elapsed);

static int frameCount = 0;
static float fpsTimer = 0;
frameCount++;
fpsTimer += dt;
if (fpsTimer >= 1.0f) {
    qDebug() << "FPS:" << frameCount;
    frameCount = 0;
    fpsTimer = 0;
}

if (scoreAnimTimer > 0.0f) {
    scoreAnimTimer -= dt;
    float t = 1.0f - scoreAnimTimer / 0.2f;
    scoreScale = 1.0f + 0.5f * (1.0f - t * t); // ease-out
} else {
    scoreScale = 1.0f;
    scoreChanged = false;
}

if (livesPulseTimer > 0.0f) {
        livesPulseTimer -= dt;
    }
}
```

8. Focus Mode and Wind System

8.1 Focus Mode Logic

Focus mode is a special high-intensity state that activates after the player reaches a score of 50. It cycles on and off based on the score difference.

Inside `updatePhysics(float dt)`:

Listing 18: Focus mode computation in updatePhysics().

```
bool newFocus = false;
if (score >= 50) {
    int t = score - 50;
    int m = t % 150;
    if (m < 100)
        newFocus = true;
}
focusMode = newFocus;
```

When focus mode is active, gravity and fall speed increase and the egg spawn interval decreases slightly, making the game faster and more intense.

8.2 Wind Activation and Strength

The wind system periodically enables horizontal forces on eggs. It uses a cooldown and random chance to determine when a gust starts, and then it picks a direction and magnitude:

Listing 19: Wind state update in updatePhysics().

```
timeSinceLastWind += dt;

if (!windActive && timeSinceLastWind >= windCooldown) {
    if (QRandomGenerator::global()->bounded(1000) < 2) {
        windActive = true;
        windTimer = QRandomGenerator::global()->bounded(1200, 2500)
           / 1000.0f;
        timeSinceLastWind = 0.0f;

        float minStrength = 2.0f;
        float maxStrength = 6.0f;
        if (focusMode) {
            minStrength = 4.0f;
            maxStrength = 10.0f;
        }

        float magnitude = QRandomGenerator::global()->bounded(
                                (int)(minStrength * 100),
                                (int)(maxStrength * 100)
                          ) / 100.0f;
        int direction =
            (QRandomGenerator::global()->bounded(0, 2) == 0) ? -1 :
               1;
        windStrength = direction * magnitude;
    }
}

if (windActive) {
    windTimer -= dt;
    if (windTimer <= 0.0f) {
        windActive = false;
        windStrength = 0.0f;
    }
}
```

8.3 Wind Dust Particles

Wind dust particles simulate small pieces of dust travelling across the screen. They are spawned randomly across a portion of the grid when wind is active:

Listing 20: Wind dust spawning.

```
if (windActive && std::abs(windStrength) > 0.05f) {
    int count = 8;  // good balance

    for (int i = 0; i < count; i++) {
        WindParticle wp;

        wp.pos = QPointF(
            QRandomGenerator::global()->bounded(cols),
            QRandomGenerator::global()->bounded(int(rows * 0.7f))
            );

        float dir = (windStrength > 0.0f) ? 1.0f : -1.0f;

        wp.vel = QPointF(
            dir * (0.4f + QRandomGenerator::global()->bounded(120) /
                100.0f),
            (QRandomGenerator::global()->bounded(-20, 21) / 100.0f)
            );

        wp.maxLife = 0.6f +
            (QRandomGenerator::global()->bounded(40) / 100.0f);
        wp.lifetime = wp.maxLife;
        wp.alpha = 1.0f;

        windParticles.push_back(wp);
    }
}
```

They are updated each frame and fade out over time:

Listing 21: Wind dust particle update.

```
QVector<WindParticle> wpSurvivors;
for (auto &wp : windParticles) {
    wp.pos += wp.vel * (dt * 60.0f);
    wp.lifetime -= dt;
    wp.alpha = qMax(0.0f, wp.lifetime / wp.maxLife);

    if (wp.lifetime > 0)
        wpSurvivors.push_back(wp);
}
windParticles = wpSurvivors;
```

8.4 Wind Streaks

In addition to dust, the game uses text arrows ("»»" or "««") to show strong wind streaks:

Listing 22: Wind streak spawn and update.

```
if (windActive && std::abs(windStrength) > 0.05f) {
    // Random chance per physics tick to avoid too many streaks
    if (QRandomGenerator::global()->bounded(100) < 30) {
        WindStreak ws;

        ws.pos = QPointF(
            QRandomGenerator::global()->bounded(cols),
            QRandomGenerator::global()->bounded(rows)
            );

        ws.maxLife = 0.8f;
        ws.lifetime = ws.maxLife;
        ws.alpha = 1.0f;

        windStreaks.push_back(ws);
    }
}

QVector<WindStreak> streakAlive;
for (auto &ws : windStreaks) {
    ws.lifetime -= dt;
    ws.alpha = qMax(0.0f, ws.lifetime / ws.maxLife);
    if (ws.lifetime > 0)
        streakAlive.push_back(ws);
}
windStreaks = streakAlive;
```

9. Egg Spawn, Physics and Difficulty

9.1 Egg Spawning and Types

Eggs are spawned periodically at the drop columns. The type is chosen randomly with different probabilities for normal, bad and life eggs:

Listing 23: Egg spawning in updatePhysics().

```
if (globalSpawnTimer >= spawnInterval) {
    int col = dropColumns[currentColumnIndex];
    bool isEdgeCol =
        (col == dropColumns.front() || col == dropColumns.back());
    bool canSpawn = true;

    float dynamicEdgeCooldown =
        qMax(0.6f, edgeSpawnCooldown - 0.03f * score);
    if (isEdgeCol && (globalTime - lastEdgeSpawnTime <
       dynamicEdgeCooldown))
        canSpawn = false;

    if (canSpawn) {
        Egg e;
        e.pos = QPointF(float(col), 0.0f);
        e.yVelocity = 0.0f;
        e.state = "falling";

        int r = QRandomGenerator::global()->bounded(100);
        if (r < 75) {
            e.type = "normal";
            e.color = QColor(Qt::white);
        } else if (r < 95) {
            e.type = "bad";
            e.color = QColor(200, 50, 50);
        } else {
            e.type = "life";
            e.color = QColor(255, 105, 180);
        }

        eggs.append(e);
        if (isEdgeCol)
            lastEdgeSpawnTime = globalTime;
    }

    currentColumnIndex =
        (currentColumnIndex + 1) % dropColumns.size();
    globalSpawnTimer = 0.0f;
    spawnInterval = 0.8f +
        QRandomGenerator::global()->bounded(0.6f);
}
```

9.2 Basket Movement

Basket motion is based on a target velocity and a smoothed acceleration towards that velocity:

Listing 24: Basket target velocity and integration.

```
if (moveLeft && !moveRight)
    basketTargetVel = -basketMaxVel;
else if (moveRight && !moveLeft)
    basketTargetVel = basketMaxVel;
else
    basketTargetVel = 0.0f;

```

```
float blend = 1.0f - qExp(-basketAccel * dt);
Q_UNUSED(blend);

basketXVelocity += (basketTargetVel - basketXVelocity) *
                   qMin(1.0f, dt * basketAccel);

basket.setX(basket.x() + basketXVelocity * dt);
basket.setX(std::clamp((float)basket.x(), 0.0f,
                       float(cols - 1)));
prevBasketX = basket.x();
```

9.3 Gravity, Difficulty and Wind Influence

Eggs are affected by gravity and wind. Gravity and maximum fall speed scale with score, and spawn interval becomes smaller as score increases. Focus mode further adjusts these parameters:

Listing 25: Egg physics and difficulty scaling.

```
float baseGravity = 10.0f + score * 0.05f;
float maxFallSpeed = 22.0f;

spawnInterval = qMax(0.6f, 1.0f - score * 0.01f);

if (focusMode) {
    baseGravity *= 1.15f;
    maxFallSpeed *= 1.15f;
    spawnInterval = qMax(0.5f, spawnInterval - 0.05f);
}
```

9.4 Per-egg Physics and Game Logic

Each egg is updated according to its type and possibly affected by wind. The collision with the basket determines score changes and life changes.

Listing 26: Per-egg update with wind and collisions.

```
QVector<Egg> survivors;
bool caughtAny = false;
bool lostAny = false;
bool lostLifeAny = false;
bool gainedLifeAny = false;

```

```
for (auto &egg : eggs) {
    egg.prevY = egg.pos.y();
    float gravity = baseGravity;
    if (egg.type == "life") gravity *= 0.5f;
    if (egg.type == "bad")  gravity *= 1.2f;

    if (egg.state == "falling") {
        egg.yVelocity += gravity * dt;
        egg.yVelocity = qMin(egg.yVelocity, maxFallSpeed);
        egg.pos.setY(egg.pos.y() + egg.yVelocity * dt);

        if (windActive) {
            egg.pos.setX(egg.pos.x() + windStrength * dt);
            egg.pos.setX(std::clamp((float)egg.pos.x(), 0.0f,
                                    float(cols - 1)));
        }

        int basketWidth = 16;
        int basketHeight = 6;

        QRectF basketRect(
            basket.x() - basketWidth / 2.0f,
            basket.y() - 0.5f,
            basketWidth,
            basketHeight + 1.5f
            );

        QRectF eggRect(egg.pos.x(), egg.pos.y(), 1.0f, 1.0f);

        if (eggRect.intersects(basketRect)) {
            egg.state = "caught";
            egg.animTimer = 0;
            caughtAny = true;

            int scoreDelta = 0;

            if (!focusMode) {
                if (egg.type == "normal" || egg.type == "life")
                    scoreDelta = 2;
                else if (egg.type == "bad")
                    scoreDelta = -2;
            } else {
                if (egg.type == "normal" || egg.type == "life")
                    scoreDelta = 5;
                else if (egg.type == "bad")
                    scoreDelta = 0;
            }

            score += scoreDelta;

            if (egg.type == "bad") {
                lives = std::max(0, lives - 1);
                lostLifeAny = true;
                flashColor = QColor(255, 0, 0);
                flashAlpha = 0.0f;
                flashTimer = 0.0f;
            }
            else if (egg.type == "life") {
                int oldLives = lives;
                lives = std::min(5, lives + 1);
                if (lives > oldLives) gainedLifeAny = true;
                flashColor = QColor(0, 255, 0);
                flashAlpha = 0.0f;
                flashTimer = 0.0f;
            }
        }
        else if (egg.pos.y() >= rows - 1) {
            egg.state = "splat";
            egg.animTimer = 0;

            if (egg.type != "bad") {
                lives = std::max(0, lives - 1);
                lostLifeAny = true;
            }
        }

        survivors.push_back(egg);
    }
    else if (egg.state == "caught") {
        egg.animTimer += dt;
        egg.scale = 1.0f - egg.animTimer * 3.0f;
        egg.alpha = 1.0f - egg.animTimer * 2.0f;
        if (egg.animTimer < 0.5f)
            survivors.push_back(egg);
    }
    else if (egg.state == "splat" && egg.animTimer < 1) {
        int numParticles = 12;
        int scale = 1000;
        for (int i = 0; i < numParticles; ++i) {
            int angleDeg =
                QRandomGenerator::global()->bounded(360);
            double rad = angleDeg * M_PI / 180.0;

            int speed =
                QRandomGenerator::global()->bounded(500, 1500);

            Particle p;
            p.pos = QPoint(int(egg.pos.x() * scale),
                           int(egg.pos.y() * scale));
            p.velocity = QPoint(int(cos(rad) * speed),
                                int(sin(rad) * speed));
            p.lifetime =
                QRandomGenerator::global()->bounded(30, 60);
            p.alpha = 255;
            p.color = egg.color;
            particles.append(p);
        }
    }
}
```

After processing eggs and particles, the global score, lives and high score are updated:

Listing 27: Score, lives and high score update.

```
eggs = survivors;
if (caughtAny) score++;
if (lostAny) lives--;
if (score > highScore) highScore = score;
if (lives <= 0) gameOver = true;

if (flashColor.isValid()) {
    flashTimer += dt;
    float fadeInDur = 0.2f;
    float fadeOutDur = 0.5f;

    if (flashTimer < fadeInDur)
        flashAlpha = flashTimer / fadeInDur;
    else if (flashTimer < fadeInDur + fadeOutDur)
        flashAlpha =
            1.0f - (flashTimer - fadeInDur) / fadeOutDur;
    else {
        flashColor = QColor();
        flashAlpha = 0.0f;
    }
}

if (caughtAny && !gameOver) {
    scoreAnimTimer = 0.2f;
    scoreScale = 1.5f;
    scoreChanged = true;
}

if ((lostLifeAny || gainedLifeAny) && !gameOver) {
    livesPulseTimer = 0.3f;
    livesChanged = true;
}

QVector<Particle> aliveParticles;
int scale = 1000;
for (auto &p : particles) {
    p.pos += p.velocity / 60;
    p.lifetime--;
    p.alpha = std::max(0,
        (p.lifetime * 255) / 60);
    if (p.lifetime > 0)
        aliveParticles.push_back(p);
}
particles = aliveParticles;
```

10. Visual Effects, Egg Shape and HUD

10.1 Egg Shape Rendering

Each egg is rendered using the member function `drawEggShape()`, which implements a pixel-art-style egg with outline and splat state:

Listing 28: Member egg drawing function.

```
void MainWindow::drawEggShape(QPainter &p,
                              const Egg &egg,
                              float cellSize)
{
    p.setRenderHint(QPainter::Antialiasing, false);

    QPointF center((egg.pos.x() + 0.5f) * cellSize,
                   (egg.pos.y() + 0.5f) * cellSize);

    float baseW = cellSize * 2.5f * 1.5f;
    float baseH = cellSize * 2.5f * 2.0f;
    float w = baseW * egg.scale;
    float h = baseH * egg.scale;

    QColor fillColor = egg.color;
    fillColor.setAlphaF(egg.alpha);
    QColor outlineColor = Qt::yellow;
    outlineColor.setAlphaF(egg.alpha);

    int step = 1;

    auto plot = [&](int gx, int gy) {
        p.fillRect(center.x() + gx,
                   center.y() + gy,
                   step, step, fillColor);
    };

    for (int yi = -h / 2; yi <= h / 2; yi += step)
    {
        float yf = float(yi) / (h / 2);

        float modifier = (yf < 0)
            ? (1.0f - 0.3f * yf * yf)
            : (1.0f + 0.1f * yf);

        int xSpan = int((w / 2) *
                        sqrt(1 - yf * yf) *
                        modifier);

        for (int xi = -xSpan; xi <= xSpan; xi += step)
        {
            plot(xi, yi);
        }
    }

    p.setBrush(outlineColor);
    for (int yi = -h / 2; yi <= h / 2; yi += step)
    {
        float yf = float(yi) / (h / 2);
        float modifier = (yf < 0)
            ? (1.0f - 0.3f * yf * yf)
            : (1.0f + 0.1f * yf);
        int xSpan = int((w / 2) *
                        sqrt(1 - yf * yf) *
                        modifier);

        // left & right edges
        p.fillRect(center.x() - xSpan,
                   center.y() + yi,
                   step, step, outlineColor);
        p.fillRect(center.x() + xSpan,
                   center.y() + yi,
                   step, step, outlineColor);
    }

    if (egg.state == "splat")
    {
        int splatW = int(w);
        int splatH = int(h * 0.4f);
        QRectF splatRect(center.x() - splatW / 2,
                         center.y() - splatH / 2,
                         splatW, splatH);
        p.fillRect(splatRect, fillColor);
        p.setPen(QPen(outlineColor, 2.0));
        p.drawRect(splatRect);
    }
}
```

10.2 Drawing the Game World

The `drawGame()` function assembles all visual elements: background, wind effects, basket, eggs, HUD and particles. In focus mode, the background is darkened.

Listing 29: Beginning of drawGame() and focus mode handling.

```
void MainWindow::drawGame(float alpha)
{
    QPixmap framePix = background;

    // In focus mode, darken the world
    if (focusMode) {
        framePix.fill(Qt::black);
    }

    QPainter painter(&framePix);
    painter.setRenderHint(QPainter::Antialiasing, true);

    float basketRenderX =
        prevBasketX + (basket.x() - prevBasketX) * alpha;
    float basketRenderY = basket.y();

    int basketWidthCells = 16;
    int basketHeightCells = 6;

    QColor basketFill(205, 133, 63);
    QColor basketOutline(120, 60, 20);

    // FLASH RENDERING
    if (flashColor.isValid() && flashAlpha > 0.0f) {
        QColor overlay = flashColor;
        overlay.setAlphaF(flashAlpha * 0.5f);
        painter.fillRect(framePix.rect(), overlay);
    }
```

Wind dust particles are rendered as small semi-transparent circles:

Listing 30: Drawing wind dust.

```
if (windActive && !windParticles.isEmpty()) {
    for (auto &wp : windParticles) {

        QColor dust(230, 230, 230);
        dust.setAlphaF(0.2f + wp.alpha * 0.8f);
        float px = wp.pos.x() * grid_box;
        float py = wp.pos.y() * grid_box;

        float size = grid_box * 0.30f;

        painter.setBrush(dust);
        painter.setPen(Qt::NoPen);
        painter.drawEllipse(QRectF(px, py, size, size));
    }
}
```

Wind streak arrows are drawn as rotated text:

Listing 31: Drawing wind streak arrows.

```
if (windActive && !windStreaks.isEmpty()) {

    painter.setFont(QFont("Arial", grid_box * 0.9f,
                          QFont::Bold));

    for (auto &ws : windStreaks) {

        bool right = (windStrength > 0);
        QString arrow = right ? ">>>>" : "<<<<";

        float px = ws.pos.x() * grid_box;
        float py = ws.pos.y() * grid_box;

        painter.save();

        painter.translate(px, py);
        painter.rotate(right ? 20 : -20);

        QColor col(230, 230, 230);
        col.setAlphaF(ws.alpha * 0.8f);

        painter.setPen(col);
        painter.drawText(0, 0, arrow);

        painter.restore();
    }
}
```

The basket is drawn as a series of grid cells with a curved rim and a motion trail:

Listing 32: Basket drawing and trail.

```
painter.setBrush(basketFill);
painter.setPen(Qt::NoPen);

for (int y = 0; y < basketHeightCells; ++y) {
    float rowY = basketRenderY + y;
    int taper = std::min(y / 1, basketWidthCells / 6);
    int startX = -basketWidthCells / 2 + taper;
    int endX = basketWidthCells / 2 - taper;
    for (int x = startX; x <= endX; ++x) {
        float cx = basketRenderX + x;
        float px = cx * grid_box;
        float py = rowY * grid_box;
        painter.fillRect(px, py, grid_box, grid_box, basketFill);
    }
}

painter.setBrush(Qt::NoBrush);
QPen rimPen(basketOutline);
rimPen.setWidthF(4.0);
rimPen.setCapStyle(Qt::RoundCap);
rimPen.setJoinStyle(Qt::RoundJoin);
painter.setPen(rimPen);

QPainterPath rimPath;
QPointF leftPoint(
    (basketRenderX - basketWidthCells / 2.0f) * grid_box,
    basketRenderY * grid_box);
QPointF rightPoint(
    (basketRenderX + basketWidthCells / 2.0f) * grid_box,
    basketRenderY * grid_box);
QPointF control(
    basketRenderX * grid_box,
    (basketRenderY - basketHeightCells * 0.5f) * grid_box);
rimPath.moveTo(leftPoint);
rimPath.quadTo(control, rightPoint);
painter.drawPath(rimPath);

// Basket trail
int trailLength = 6;
```

```
for (int i = 1; i <= trailLength; ++i) {
    int fade = qMax(10, 120 - i * 18);
    QColor trailColor(160, 82, 45, fade);
    float trailX =
        basketRenderX - basketXVelocity * (i * 0.02f);
    painter.fillRect(
        (trailX - basketWidthCells / 2.0f) * grid_box,
        basketRenderY * grid_box,
        basketWidthCells * grid_box,
        basketHeightCells * grid_box,
        trailColor);
}
```

Eggs are rendered by interpolating their y-position with `alpha` for smooth motion:

Listing 33: Egg rendering loop.

```
for (auto &egg : eggs) {
    Egg renderEgg = egg;
    renderEgg.pos.setY(
        egg.prevY + (egg.pos.y() - egg.prevY) * alpha);
    drawEggShape(painter, renderEgg, (float)grid_box);
}
```

The HUD displays score, high score, focus mode banner and hearts for lives:

Listing 34: HUD rendering with score, high score and lives.

```
painter.setFont(QFont("Comic Sans MS", 24, QFont::Bold));
QColor scoreColor(255, 215, 0);
if (focusMode) scoreColor = QColor(0, 255, 255);

painter.setPen(scoreColor);
painter.save();
painter.translate(QPointF(30, 45));
painter.scale(scoreScale, scoreScale);
painter.drawText(QPointF(0, 0),
                 QString("Score: %1").arg(score));
painter.restore();

// High score
QFont highFont("Arial", 18, QFont::Bold);
painter.setFont(highFont);
painter.setPen(QColor(200, 200, 255));
painter.drawText(30, 75,
                  QString("High Score: %1").arg(highScore));

// Focus Mode banner
if (focusMode) {
    painter.setFont(QFont("Arial", 18, QFont::Bold));
    painter.setPen(QColor(0, 255, 255));
    painter.drawText(framePix.rect(),
                     Qt::AlignTop | Qt::AlignHCenter,
                     "FOCUS MODE  x5 SCORE");
}

// Lives as hearts
int heartSize = 24;
float pulseScale =
    1.0f + 0.5f * (livesPulseTimer / 0.3f);
for (int i = 0; i < lives; ++i) {
    int x = framePix.width() - 40 - i * (heartSize + 5);
    int y = 20;

    QPainterPath heartPath;
    heartPath.moveTo(x + heartSize / 2.0, y + heartSize / 5.0);
    heartPath.cubicTo(x + heartSize / 2.0, y, x, y,
                      x, y + heartSize / 3.0);
    heartPath.cubicTo(x, y + heartSize * 0.8,
                      x + heartSize / 2.0, y + heartSize,
                      x + heartSize / 2.0, y + heartSize * 0.9);
    heartPath.cubicTo(x + heartSize / 2.0, y + heartSize,
                      x + heartSize, y + heartSize * 0.8,
                      x + heartSize, y + heartSize / 3.0);
    heartPath.cubicTo(x + heartSize, y,
                      x + heartSize / 2.0, y,
                      x + heartSize / 2.0, y + heartSize / 5.0);

    painter.save();
    painter.translate(x + heartSize / 2.0, y + heartSize / 2.0);
    painter.scale(pulseScale, pulseScale);
    painter.translate(-(x + heartSize / 2.0),
                      -(y + heartSize / 2.0));
    painter.setBrush(Qt::red);
    painter.setPen(Qt::NoPen);
    painter.drawPath(heartPath);
    painter.restore();
}
```

Finally, egg splat particles are drawn as small circles:

Listing 35: Drawing egg splat particles.

```
painter.setPen(Qt::NoPen);
for (auto &p : particles) {
    QColor c = p.color;
    c.setAlpha(p.alpha);
    painter.setBrush(c);
    painter.setPen(Qt::NoPen);
    int size = grid_box / 3;
    painter.drawEllipse(
        QPointF(p.pos.x() / 1000.0,
                p.pos.y() / 1000.0) * grid_box,
        size, size);
}

painter.end();
ui->frame->setPixmap(framePix);
```

11. Screenshots and Figures

In the final report, the following screenshots should be captured from the running Qt application and inserted into the Overleaf project. Save them with the exact filenames given below and upload them to Overleaf.

11.1 Menu Screen

File name: `game_start.png`

Screenshot showing the main menu with the game title, player name input, PLAY button and LEADERBOARD button.

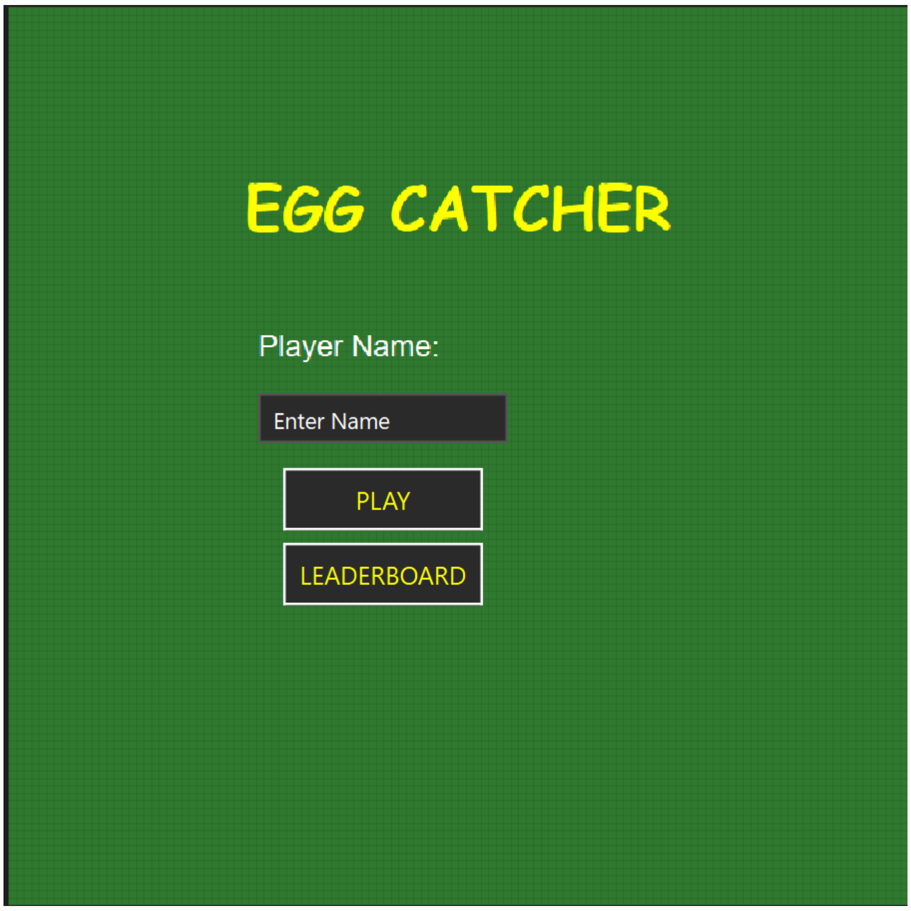


Figure 1: Main menu of the Egg Catcher game.

11.2 Normal Gameplay

File name: `gameplay_normal.png`

Screenshot showing normal gameplay with eggs falling more or less straight down (without obvious wind) and the basket catching them.

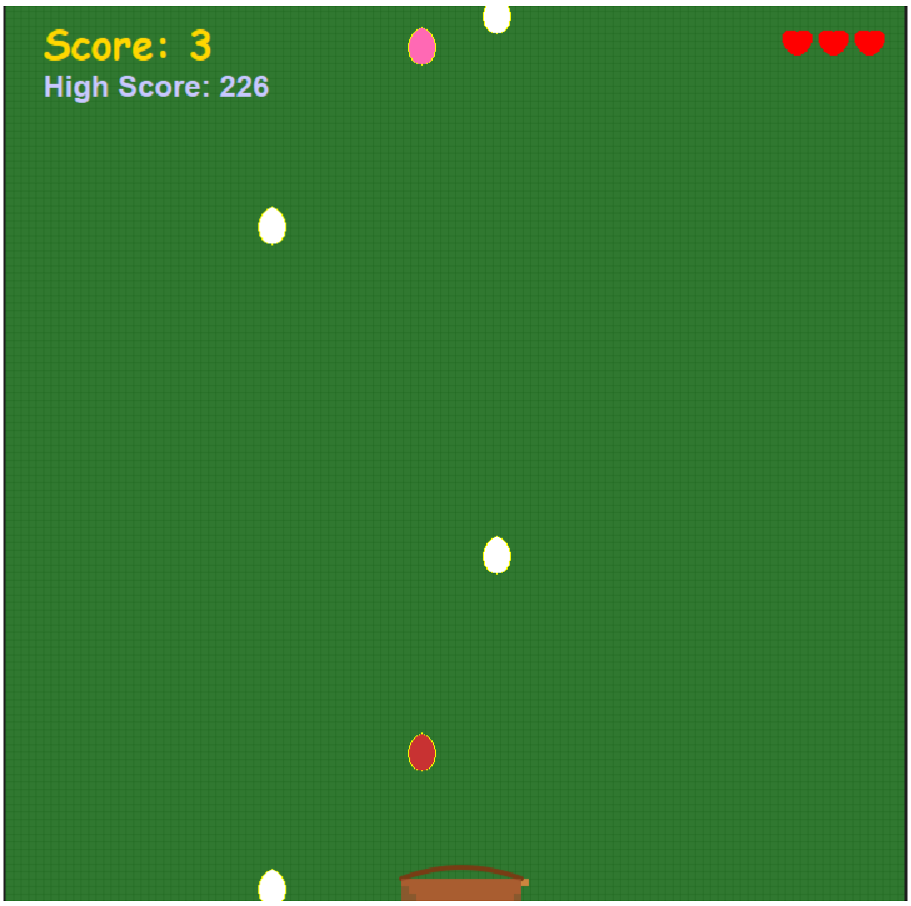


Figure 2: Normal gameplay with falling eggs and basket movement.

11.3 Windy Gameplay

File name: `gameplay_wind.png`

Screenshot where wind is active; wind dust particles and streak arrows ("»»" or "««") should be visible and eggs should appear slightly slanted or drifting sideways.

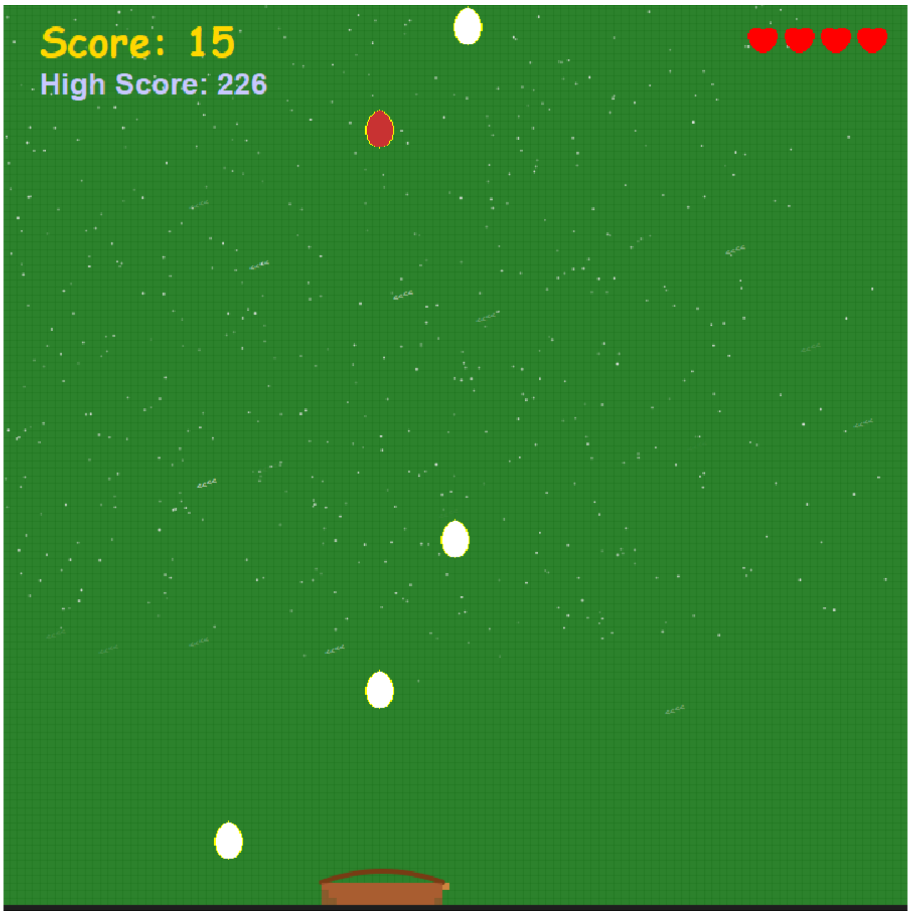


Figure 3: Gameplay during wind, showing drifting eggs and wind effects.

11.4 Bad and Life Eggs

File name: `gameplay_bad_life.png`

Screenshot showing both bad (red) eggs and life (pink) eggs on screen, along with the basket and current HUD.

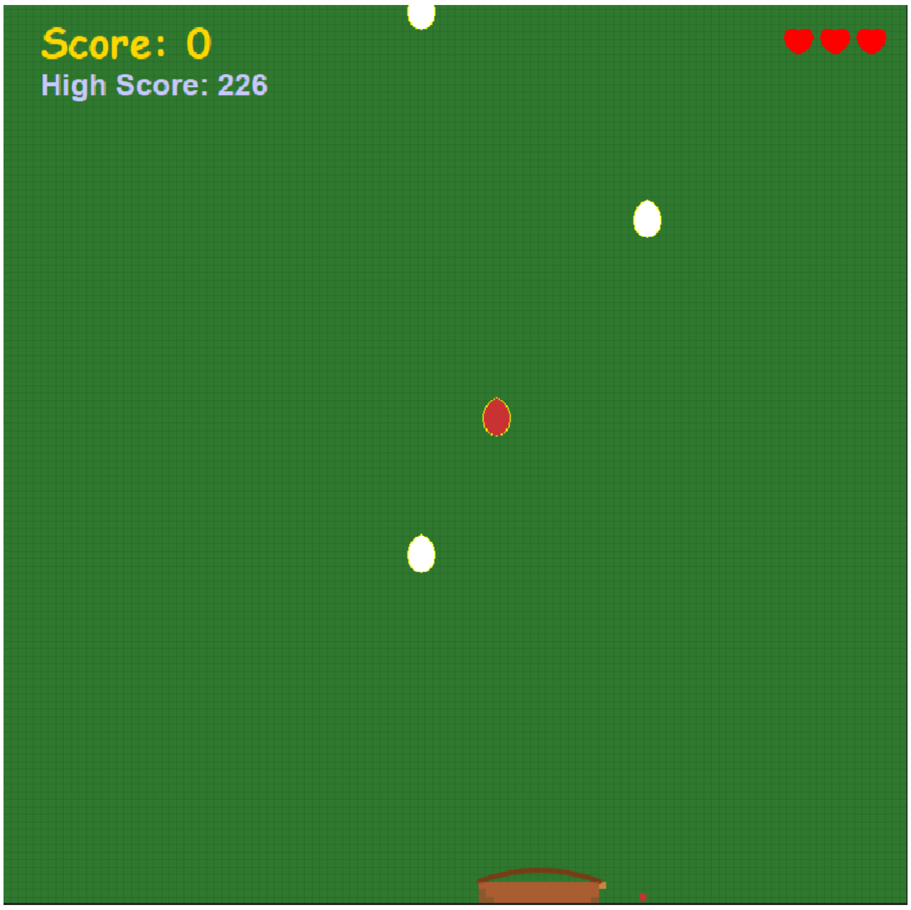


Figure 4: Bad eggs and life eggs visible together during gameplay.

11.5 Game Over Screen

File name: `game_over.png`

Screenshot of the Game Over screen where final score is shown and instructions for restarting or returning to the menu are visible.

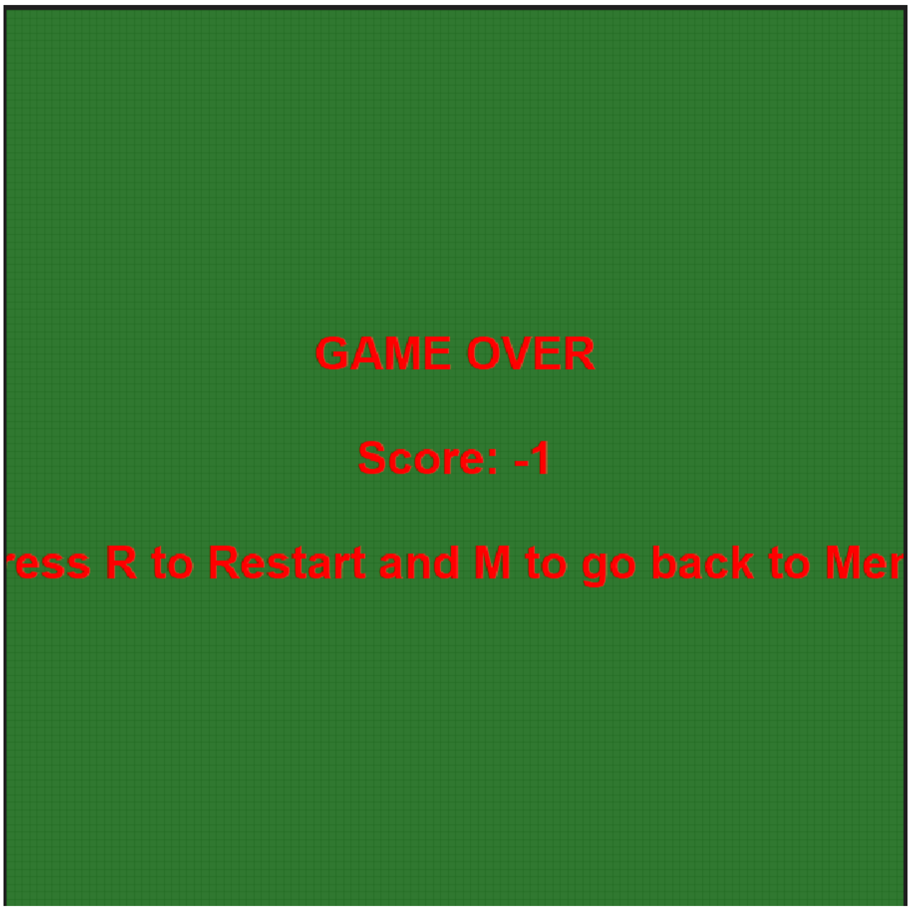
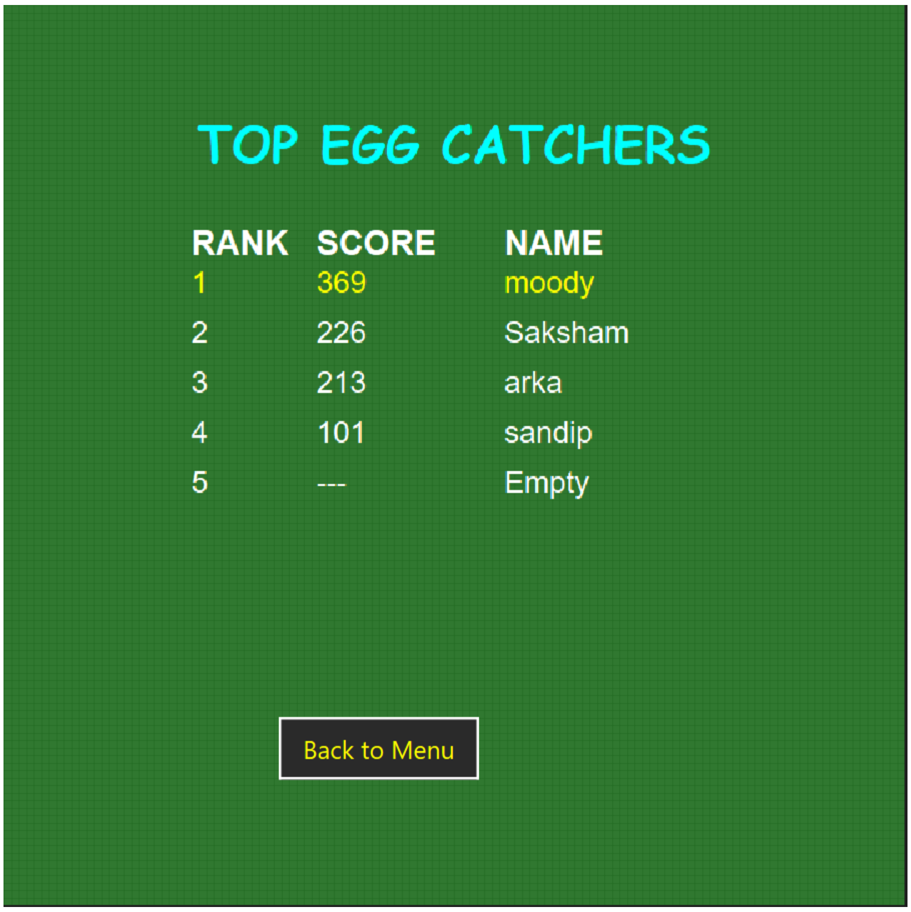


Figure 5: Game over screen with final score.

11.6 Leaderboard Screen

File name: `leaderboard.png`

Screenshot of the leaderboard screen showing at least a few top entries with rank, score and name.


TOP EGG CATCHERS

RANK	SCORE	NAME
1	369	moody
2	226	Saksham
3	213	arka
4	101	sandip
5	---	Empty

Figure 6: Leaderboard screen of top Egg Catcher scores.

12. Flowchart of Game Logic

A flowchart summarising the high-level logic of the game (menu, leaderboard, playing, game over) should be created separately and saved as `flowchart.png`. Insert it as follows:

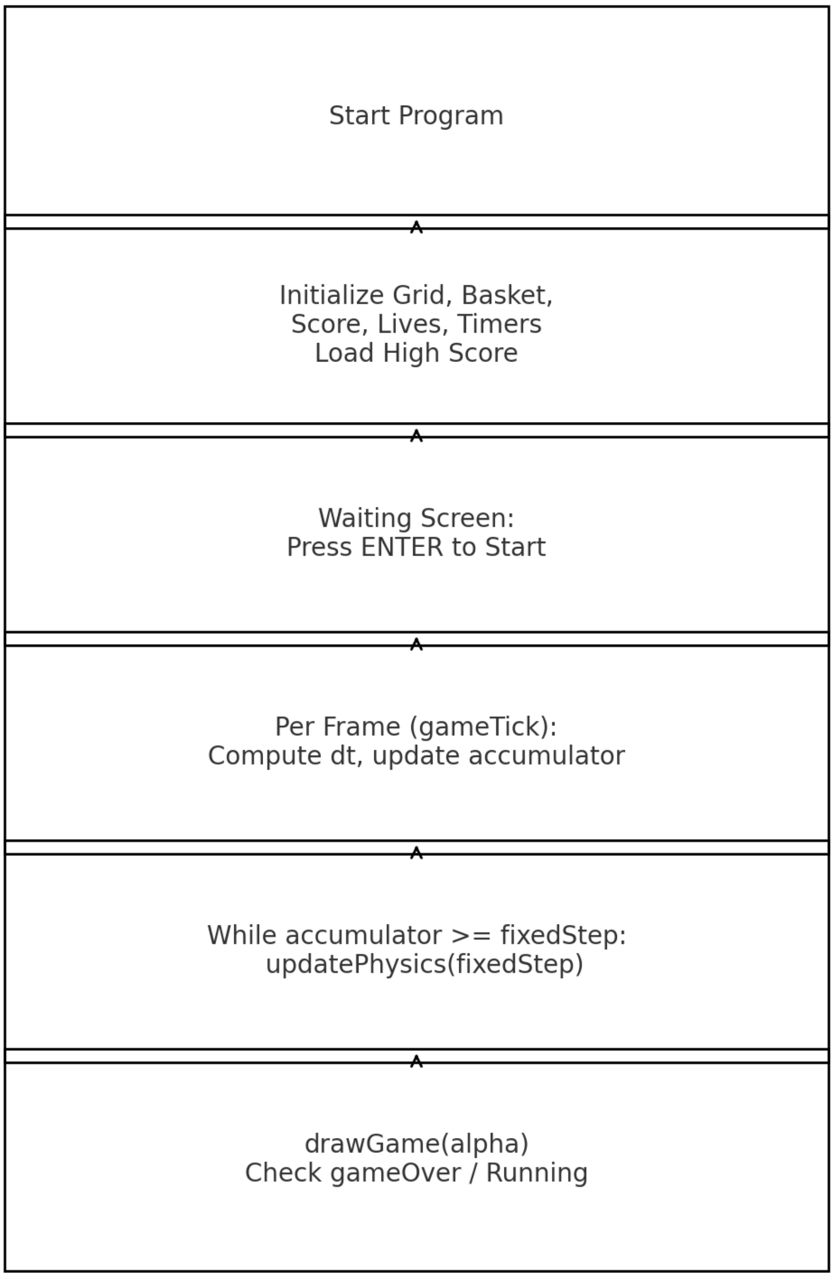

Start Program
Initialize Grid, Basket, Score, Lives, Timers Load High Score
Waiting Screen: Press ENTER to Start
Per Frame (gameTick): Compute dt, update accumulator
While accumulator >= fixedStep: updatePhysics(fixedStep)
drawGame(alpha) Check gameOver / Running

Figure 7: Flowchart of Egg Catcher game loop and state transitions.

13. Results and Discussion

13.1 Observations

- As the score increases, gravity and spawn rate change, leading to faster and denser egg patterns.

- Wind, especially when combined with focus mode, makes predicting egg trajectories more difficult and increases the challenge.
- Life eggs help balance the difficulty by granting extra lives, but they are rarer and fall more slowly.
- Bad eggs introduce risk–reward decisions: moving quickly to catch eggs may result in catching a bad egg and losing lives or score.
- Visual feedback such as flashes, score pulsing, heart pulsing and wind dust/streaks makes the game feel responsive and polished.
- The leaderboard and player name input give the game a competitive feel and motivate players to beat previous scores.

13.2 Performance

The game runs smoothly at approximately 60 frames per second due to:

- The fixed timestep physics loop (1/120 s) which ensures numerical stability.
- Use of simple integer-based grid coordinates for movement and rendering.
- Efficient drawing using a single `QPixmap` as the frame buffer.
- Limiting particle counts and wind effects to maintain performance.

14. Conclusion

In this project, an Egg Catcher game was successfully implemented using Qt and C++. The project demonstrates:

- Application of 2D computer graphics concepts such as rasterization, drawing primitives, coordinate systems and basic animation.
- Use of timer-based animation and event-driven programming for real-time game development.
- Implementation of game mechanics including scoring, lives, multiple egg types, wind-based difficulty, focus mode and particle effects.
- File handling and app data management for storing per-device player names and high scores.
- Integration with a leaderboard manager to maintain and display top global scores with names.

The addition of wind, focus mode and a leaderboard significantly enhances the gameplay experience beyond a simple falling-objects game and shows how multiple systems in computer graphics and game programming can be combined.

Future Enhancements

Some possible enhancements for the Egg Catcher game are:

- Adding background music and different sound effects for each egg type and game state.
- Introducing multiple levels or stages with different lane positions, colours or background themes.
- Implementing a pause/resume menu and difficulty selection at the start.
- Adding power-ups (slow motion, shield, magnet) and additional egg types with more complex behaviours.
- Porting the game to mobile platforms with touch controls.