# **Bubble Burst Game Report**

## **Introduction**

The **Bubble Burst** game is an interactive Java application designed to engage players in a fast-paced environment where they must burst bubbles before time runs out. The game features ten rounds, each increasing in difficulty. Players interact with the game using a graphical user interface (GUI) built with Java Swing.

## **System Overview**

### **I . Graphical User Interface (GUI)**

The game consists of two primary GUIs:

#### **First GUI:**

* **Components**:
  + **Start Button**: Initiates the game.
  + **Restart Button**: Resets the game state.
  + **JSlider**: Allows players to select the game difficulty (Easy, Medium, Hard).

public class MainMenu extends JFrame {

private JButton startButton;

private JButton restartButton;

private JSlider difficultySlider;

public MainMenu() {

setTitle("Bubble Burst - Main Menu");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setSize(400, 300);

setLocationRelativeTo(null);

// Difficulty Slider Setup

difficultySlider = new JSlider(JSlider.HORIZONTAL, 0, 2, 0);

java.util.Hashtable<Integer, JLabel> labelTable = new java.util.Hashtable<>();

labelTable.put(0, new JLabel("Easy"));

labelTable.put(1, new JLabel("Medium"));

labelTable.put(2, new JLabel("Hard"));

difficultySlider.setLabelTable(labelTable);

// Buttons Setup

startButton = new JButton("Start");

restartButton = new JButton("Restart");

startButton.addActionListener(e -> startGame());

restartButton.addActionListener(e -> restartGame());

}

private int getDifficultyBubbleCount() {

switch (difficultySlider.getValue()) {

case 0: return 4; // Easy

case 1: return 5; // Medium

case 2: return 6; // Hard

default: return 4;

}

}

}

#### **Second GUI:**

* **Game Playing Field**: A JPanel where bubbles are displayed and interacted with. This panel handles mouse events for bubble clicking and rendering.

public class GamePanel extends JPanel {

private final MainMenu mainMenu;

private final int numBubbles;

private List<Bubble> bubbles;

private int round = 1;

public GamePanel(MainMenu mainMenu, int numBubbles) {

this.mainMenu = mainMenu;

this.numBubbles = numBubbles;

this.bubbles = new ArrayList<>();

setBackground(Color.WHITE);

}

}

### 

### **II . Playing Field Dimensions**

The playing field is set as the dimensions of the JPanel, providing a clear area for displaying and interacting with the bubbles. The panel is responsible for rendering all visual components of the game.

gamePanel.setPreferredSize(new Dimension(800, 600));

## 

### **1. Game Initialization**

* **Start and Restart Logic**: When the game starts, it initializes the game state, sets the round number to 1, and retrieves the number of bub bles based on the difficulty level selected by the player.

private void startGame() {

roundNumber = 1;

numberOfBubbles = difficultySlider.getValue();

bubbles.clear(); // Clear previous bubbles

repositionGlobal(); // Position bubbles for Round 1

gameTimer.start(); // Start the round timer

}

* **Resetting the Game**: The restart functionality allows players to reset the game without changing the difficulty setting.

private void restartGame() {

roundNumber = 1;

bubbles.clear();

repositionGlobal(); // Reposition bubbles for a new game

gamePanel.repaint(); // Refresh the game panel

}

### **III . Repositioning Bubbles Globally**

### **1. Round 1**

* **User Interaction**:
  + The program prompts the player to define the origin for each bubble by clicking on the JPanel.
  + Error Handling: If the selected origin results in a bubble that extends beyond the panel's dimensions, an error message is displayed, prompting the user to select another position.
* **Start Condition**:
  + Round 1 initiates once the number of defined origins equals the number of bubbles based on the chosen difficulty.

private void handleMouseClick(Point p) {

if (isPlacingBubbles) {

if (isValidBubblePosition(p.x, p.y)) {

bubbles.add(new Bubble(p.x, p.y));

placedBubbles++;

if (placedBubbles == numBubbles) {

isPlacingBubbles = false;

}

}

}

}

### **2. Rounds 2 – 10**

* **Random Positioning**:
  + Bubbles are repositioned using randomly generated coordinates within the bounds of the playing field. Java's random number functions are utilized for this purpose.

private void repositionBubbles() {

bubbles.clear();

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

Bubble newBubble;

int attempts = 0;

do {

int x = (int)(Math.random() \* (getWidth() - 60) + 30);

int y = (int)(Math.random() \* (getHeight() - 60) + 30);

newBubble = new Bubble(x, y);

attempts++;

} while (checkCollisions(newBubble) && attempts < 100);

if (attempts < 100) {

bubbles.add(newBubble);

}

}

}

### **IV . Local Repositioning of Bubbles**

* **Local Neighborhood Movement**: Each bubble can hop within a defined neighborhood area. The neighborhood expands with each round.

public void repositionLocal(int round, int panelWidth, int panelHeight) {

int neighborhood = 50 + (round - 1) \* 18;

double newX, newY;

do {

newX = x + (Math.random() \* 2 - 1) \* neighborhood;

newY = y + (Math.random() \* 2 - 1) \* neighborhood;

} while (!isValidPosition(newX, newY, panelWidth, panelHeight));

x = newX;

y = newY;

}

* **Incremental Neighborhood Expansion**:
  + The local neighborhood increases with each round, starting from 50 for Round 1 and expanding by 18 for each subsequent round.

int neighborhoodSize = 50 + (roundNumber - 1) \* 18; // Calculate neighborhood size for current round

### **V . Collision Avoidance**

* **Overlap Checking**: The game ensures that no two bubbles overlap by checking their boundaries when positioning.

private boolean isValidBubblePosition(int x, int y) {

Bubble tempBubble = new Bubble(x, y);

// Check panel bounds

if (x < tempBubble.getRadius() || x > getWidth() - tempBubble.getRadius() ||

y < tempBubble.getRadius() || y > getHeight() - tempBubble.getRadius()) {

JOptionPane.showMessageDialog(this,

"Invalid position! Bubble must be fully within the playing field.",

"Error", JOptionPane.ERROR\_MESSAGE);

return false;

}

// Check collision with other bubbles

for (Bubble existing : bubbles) {

if (tempBubble.intersects(existing)) {

JOptionPane.showMessageDialog(this,

"Invalid position! Bubbles cannot overlap.",

"Error", JOptionPane.ERROR\_MESSAGE);

return false;

}

}

return true;

}

### **VI . Bubble Burst Mechanism**

* **Burst Logic**: A bubble is burst when the player clicks on it. Upon bursting, the bubble is removed from the game, and the panel is refreshed.

private boolean isValidBubblePosition(int x, int y) {

Bubble tempBubble = new Bubble(x, y);

// Check panel bounds

if (x < tempBubble.getRadius() || x > getWidth() - tempBubble.getRadius() ||

y < tempBubble.getRadius() || y > getHeight() - tempBubble.getRadius()) {

JOptionPane.showMessageDialog(this,

"Invalid position! Bubble must be fully within the playing field.",

"Error", JOptionPane.ERROR\_MESSAGE);

return false;

}

// Check collision with other bubbles

for (Bubble existing : bubbles) {

if (tempBubble.intersects(existing)) {

JOptionPane.showMessageDialog(this,

"Invalid position! Bubbles cannot overlap.",

"Error", JOptionPane.ERROR\_MESSAGE);

return false;

}

}

return true;

}

### 

### **VII. Rounds Implementation**

* **Starting New Rounds**: A new round begins when all bubbles have been burst. The program will reposition the bubbles for the next round and increase the round counter.

private boolean allBubblesBurst() {

return bubbles.stream().allMatch(Bubble::isBurst);

}

protected void paintComponent(Graphics g) {

super.paintComponent(g);

Graphics2D g2d = (Graphics2D) g;

// Draw round information

g2d.drawString("Round: " + round + "/10", 10, 30);

}

### **VIII. Game Over Implementation**

* **Game Over Logic**: The game ends when the player completes all 10 rounds, clicks anywhere in the field, or fails to burst all bubbles within the time limit.

private void endGame() {

gameOver = true;

if (gameTimer != null) {

gameTimer.stop();

}

showFinalScore();

}

private void showFinalScore() {

String message = String.format(

"Game Complete!\n" +

"Final Score: %d\n" +

"Total Bubbles Popped: %d\n" +

"Rounds Completed: %d",

currentScore, totalBubblesPopped, round

);

JOptionPane.showMessageDialog(this, message, "Game Over", JOptionPane.INFORMATION\_MESSAGE);

}

### **IX. Timer Implementation**

* **Round Timer**: Each round starts a timer that counts down from a predefined time limit, decreasing by one second for each subsequent round.

private void startTimer() {

if (gameTimer != null) {

gameTimer.stop();

}

timeLeft = Math.max(5, 15 - (round - 1));

gameTimer = new Timer(1000, e -> {

timeLeft--;

if (timeLeft <= 0) {

endGame();

}

repaint();

});

gameTimer.start();

}

protected void paintComponent(Graphics g) {

Graphics2D g2d = (Graphics2D) g;

if (timeLeft <= 5) {

g2d.setColor(Color.RED);

} else if (timeLeft <= 10) {

g2d.setColor(Color.ORANGE);

}

g2d.drawString("Time: " + timeLeft + "s", getWidth() - 120, 30);

}

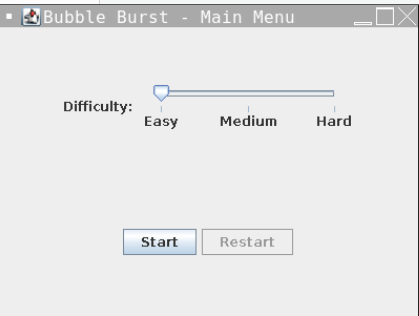
* **Display**: The remaining time is shown on the game interface, prompting players to burst bubbles quickly.

JLabel timerLabel = new JLabel("Time: 15");

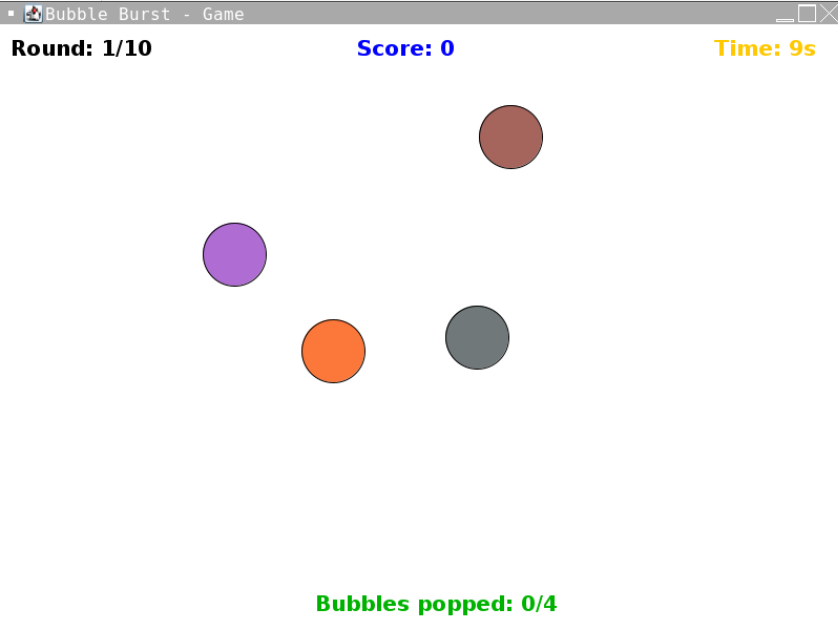
gameFrame.add(timerLabel); // Add timer to the game frame

## 

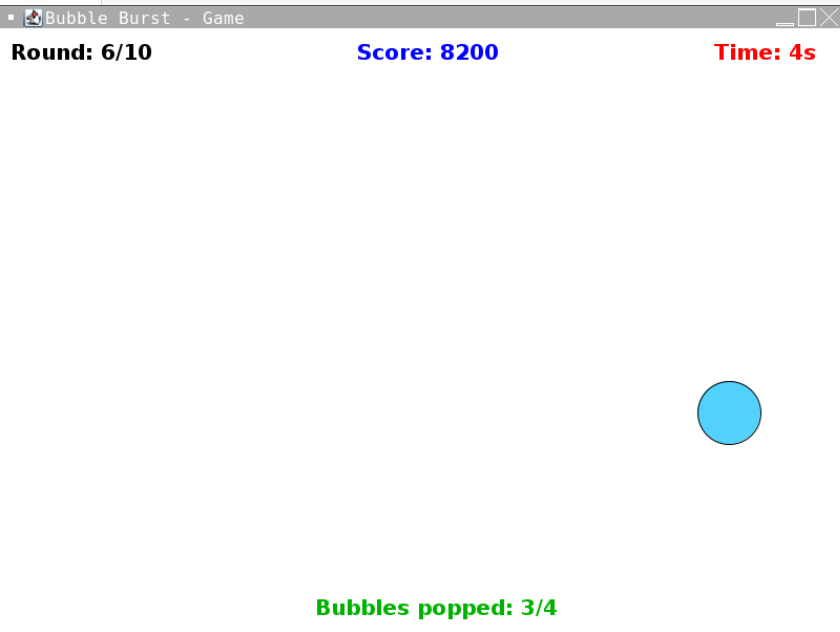
**ScreenShots :**

****

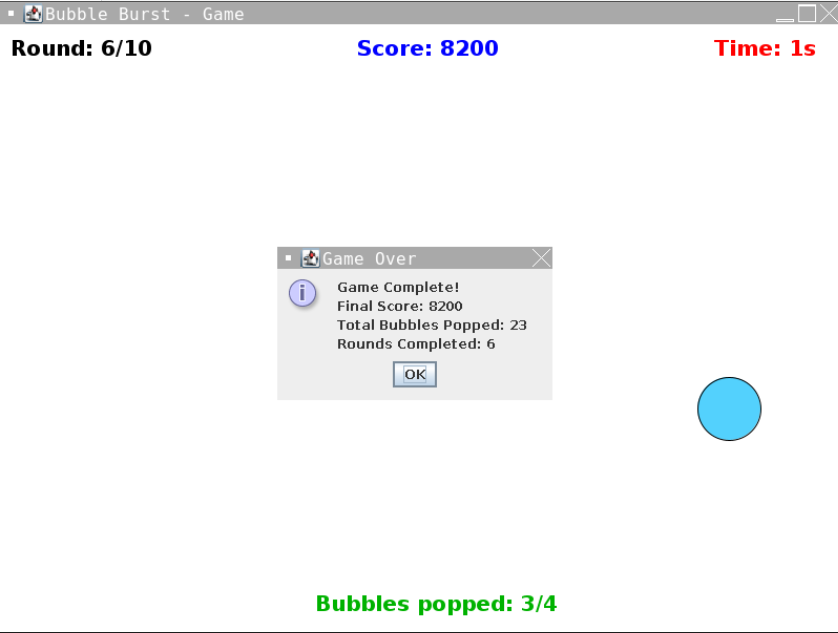
* **Main Menu**

****

* **After starting the game ( this is round 1 )**

****

* **Game is in round 6 now , Time is ticking from 15 seconds to 4 seconds**

****

* **Game Over ( It displays Final Score of the Player )**
* **Time is completed**

**Conclusion**

The Bubble Burst game exemplifies the effective application of Java Swing for creating an interactive gaming experience, leveraging object-oriented programming principles and event-driven design. Through the implementation of various classes, including MainMenu and GamePanel, the game demonstrates a clear separation of concerns, with the GUI effectively managing user input and game state.

Key technical features include:

1. **Graphical User Interface (GUI):** The use of Swing components like JFrame, JButton, and JSlider allows for intuitive user interactions, with the MainMenu class enabling players to select difficulty levels and initiate game sessions.
2. **Game Logic and State Management:** The game maintains a robust state management system, tracking the round number and bubble counts. The startGame() and restartGame() methods illustrate efficient handling of game initialization and state reset.
3. **Bubble Positioning and Collision Detection:** The implementation of bubble repositioning strategies, both global and local, showcases the use of randomization and neighborhood logic for bubble placement. Collision detection is achieved through rigorous validation methods, ensuring that no two bubbles overlap and that all bubbles remain within the defined playing field.
4. **Timer Implementation:** The countdown timer adds a critical time-based challenge, with dynamic adjustment of time limits for each round. The Timer class facilitates effective game pacing, while visual feedback on remaining time enhances player engagement.
5. **Game Over and Scoring System:** The logic for determining game completion, including criteria for both winning and losing, is clearly defined. The showFinalScore() method provides a straightforward mechanism for displaying player performance metrics at the end of the game.

In conclusion, the Bubble Burst game not only serves as a functional application but also as a case study in effective Java programming practices, illustrating key concepts in GUI development, event handling, and game mechanics. Future enhancements could involve the integration of sound effects, advanced graphics using Java 2D or 3D capabilities, and a more complex scoring algorithm, thereby further enriching the player experience and expanding the game's technical scope.