DX-BALL GAME REPORT

Name: Alperen Ulu

Game Description: DX-BALL

DX-BALL is a brick-breaking game where the goal is to clear all bricks by bouncing a ball off a paddle. The game takes place in a zero-gravity environment, with the ball launched at a fixed speed and a chosen angle. It bounces off walls and bricks, changing direction upon collision. The player must keep the ball in play using a paddle—if it touches the bottom, the game is lost. Victory is achieved by removing all bricks.

Gameplay Elements

- **Ball & Paddle:** A blue ball moves across the screen, while a grey paddle prevents it from falling.
- **Bricks:** 44 reddish bricks, each worth 10 points, making the highest possible score 440.
- **Aiming Mechanic:** A red trajectory line helps adjust the launch angle.
- **User Interface:** Displays score, launch angle, and pause status (if enabled).

Introduction to StdDraw

StdDraw is a Java library for creating simple 2D graphics and animations.

Key Features

- **Coordinate System & Scaling:** Customizable x-y axis for precise element placement.
- **Drawing Capabilities:** Supports shapes (rectangles, circles, lines) and text rendering.
- **Animation & Interactivity:** Frame timing control, dynamic screen updates, and key press detection.
- **Customization:** Adjustable fonts and pen colors for better visual representation.

Summary of Given Variables

- **Canvas & Scaling:** The game window is **800x400 pixels**, with the coordinate system scaled accordingly.
- **Brick Colors:** A predefined **array of colors** (red shades) is used for bricks.
- **Ball Properties:** The **blue ball** has a **radius of 8**, an **initial velocity of 5**, and starts at **(400,18)**.
- **Paddle Properties:** The **grey paddle** is centered at **(400,5)** with a **half-width of 60**, a **half-height of 5**, and moves at **speed 20**.
- **Brick Properties:** Each **brick has a half-width of 50 and half-height of 10**.
- **Brick Positions:** A **2D array** stores the coordinates of **44 bricks** arranged in a structured pattern.
- **Brick Colors:** Each brick has an assigned color from the **color array**, creating a varied appearance.

Summary of Created Variables

Doubles

Shooting Direction & Angle:

- shootingAngle: Current shooting angle (0° to 180°).
- shootingTrajectoryLength: The length of the red aiming line.
- trajectoryLengthX, trajectoryLengthY: X and Y components of the aiming trajectory.
- shootingLineStart, shootingLineEnd: Start and end coordinates of the aiming line.

- **Ball Position & Motion:**
  - newBallPos: Stores the most recent position of the ball.
  - velocityOnX, velocityOnY: Ball's velocity components along X and Y axes.
- **Collision Mechanics:**
  - theta, phi: Angles used for collision calculations.
  - theta: Angle between velocity vector and x-axis.
  - phi: Angle between normal vector and x-axis.
  - newVelocityOnX, newVelocityOnY: Ball's velocity after hitting a corner.

Integers

Scoring System:

- scoreOfPlayer: Current score.
- scoreForOneBrick: Points earned per brick (10 points per hit).

- **Game Timing & Brick Count:**
  - frameTime: Frame delay (15 milliseconds per frame).
  - numOfBricks: Total number of bricks in the game.

Booleans

Game State Controls:

- isGameInitialized: Tracks whether the game has started.
- isGameOver: True if the player loses.
- isVictory: True if all bricks are cleared.

- **Pause Mechanism:**
  - isSpacePressed: Tracks spacebar presses for pausing.
  - isPaused: Indicates whether the game is paused.
- **Brick Collision Detection:**
  - isLeft, isRight, isTop, isBottom: Flags to check if a brick has neighboring bricks.
  - brickHitStatus: An array tracking which bricks have been hit.

Summary of Methods

1. contains(double[] coordinate, double[][] arr, boolean[] hit)

- Checks if a **specific coordinate pair** exists in a **2D array** of brick positions.
- Also verifies that the corresponding brick has **not been hit**.
- Returns true if the coordinate is found and the brick is unbroken; otherwise, returns false.

2. indexOf(double[] coordinate, double[][] arr)

- Searches for a **specific coordinate pair** in a **2D array**.
- If found, returns the **index** of that coordinate in the array.
- If not found, returns -1 but it is never used.

3. isAllHit(boolean[] arr)

- Checks if **all bricks have been hit**.
- Iterates through the boolean array arr, which tracks hit status.
- Returns true if **all values are true** (all bricks destroyed); otherwise, returns false.

4. thetaFinder(double velocityOnX, double velocityOnY)

- Computes the **angle (theta)** between the **velocity vector** and the **x-axis**.
- Uses Math.atan(velocityOnY/velocityOnX) to find the angle.
- If the velocity in the x-direction (velocityOnX) is negative, adds **π (180°)** to adjust the angle to the correct quadrant, since the arctan is defined in the interval **[-π/2, π/2]**.
- Returns the calculated **theta**.

5. phiFinder(double[] newBallPos, double[] pos, double halfheight, double halfwidth)

- Computes the **angle (phi)** between the **normal vector** at a corner of a brick and the **x-axis**.
- Determines **which corner of the brick** is being considered:
  - **Top-right, top-left, bottom-right, bottom-left**
- Uses **Math.atan**(distanceOnX/distanceOnY) to compute the normal angle based on the **brick's position and dimensions**.
- Adds **π (180°)** if necessary for correct quadrant adjustments, since the arctan is defined in the interval **[-π/2, π/2].**
- Returns the calculated **phi**.

6. portionFinder(...)

- Determines the **portion of the ball's motion before a collision** with a brick corner.
- Uses the **quadratic formula** * to find the roots of the equation describing the ball's trajectory.
- Selects the **valid root (rootOne or rootTwo)** that falls between 0 and 1, representing the correct portion of the movement.
- Returns the selected **portion value** or -1 if no valid portion is found but it is never used since the conditions are pre-checked.

*** How is the quadratic formula derived (detail)** (as shown in the Figure 1, 2 and 3).**?**

Figure 1:

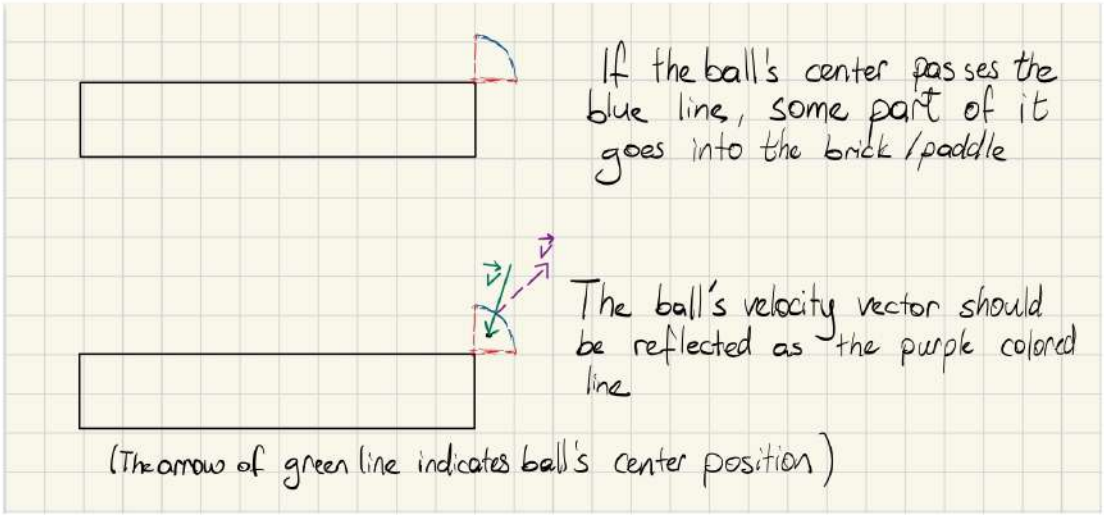


Figure 2:

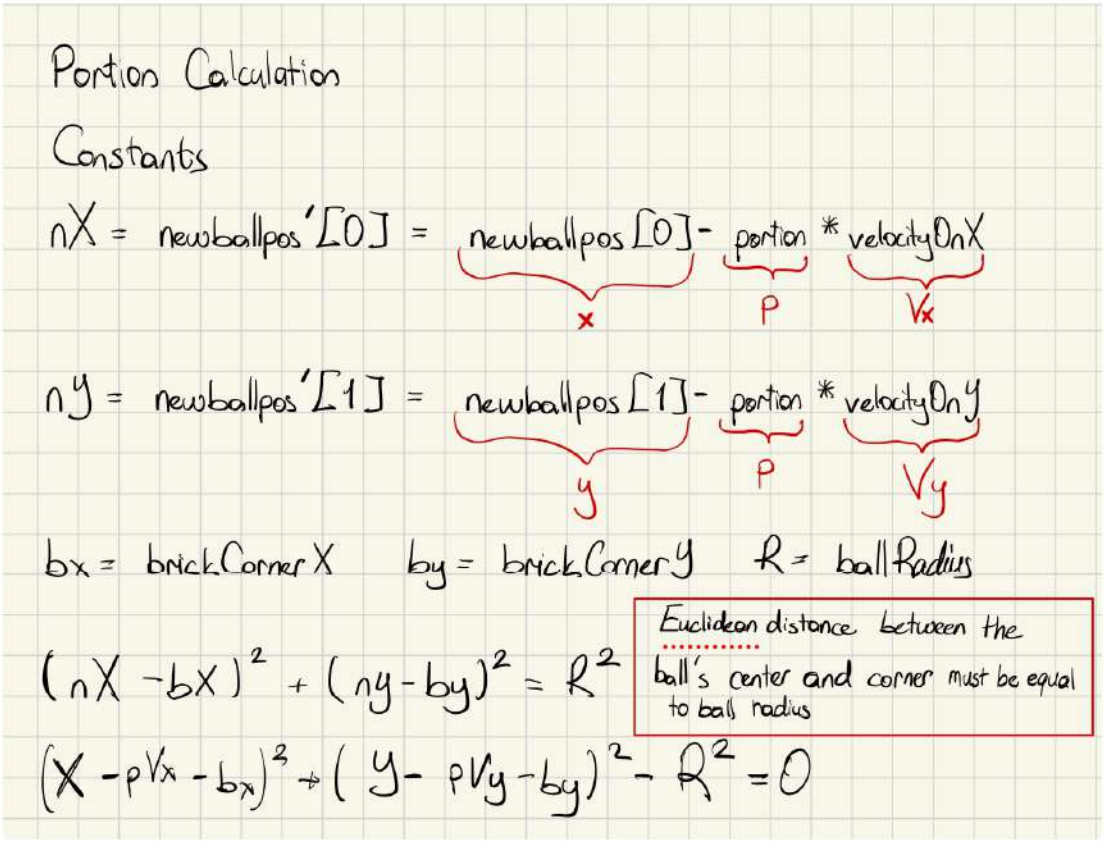
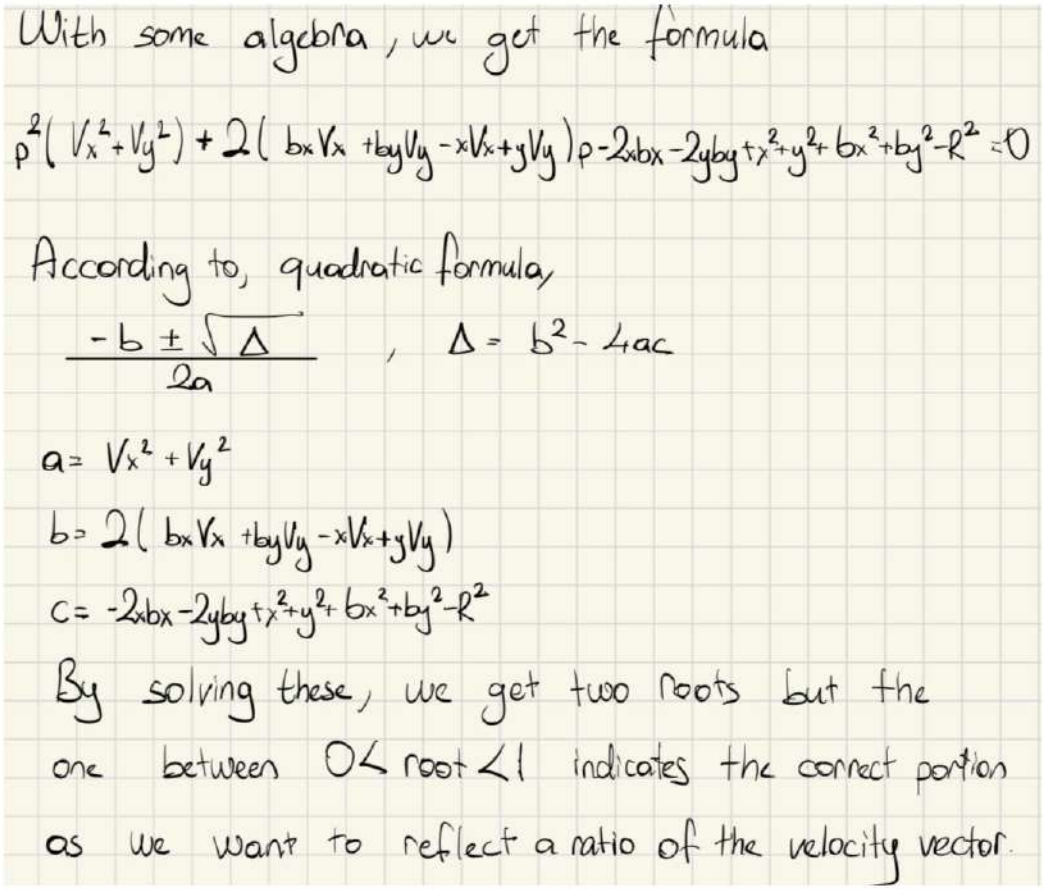


Figure 3:



7. distanceFinder(...)

- Computes the **distance the ball must reflect** after colliding with a **surface**.
- Uses the **ball's position, surface position, and radius** to determine the reflection distance.
- Handles different **collision directions** ("top", "bottom", "left", "right").
- Returns the computed **reflection distance** * or -1 if an invalid direction is given but it is never used.

*** How is the reflection distance derived (detail)** (as shown in the Figure 4 and 5).**?**

Figure 4:

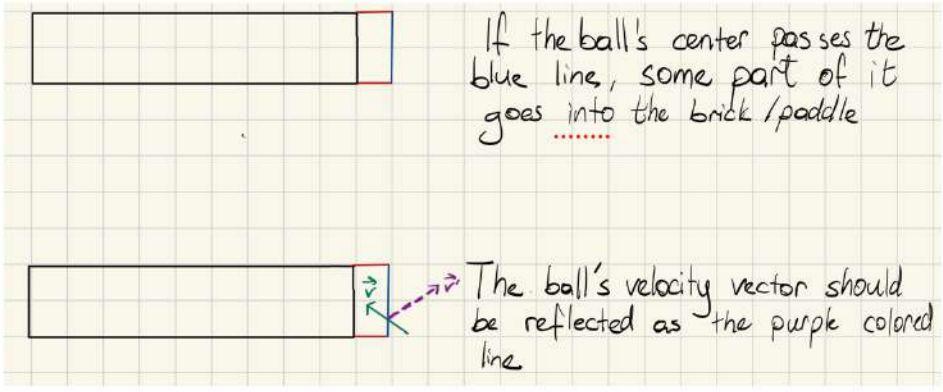
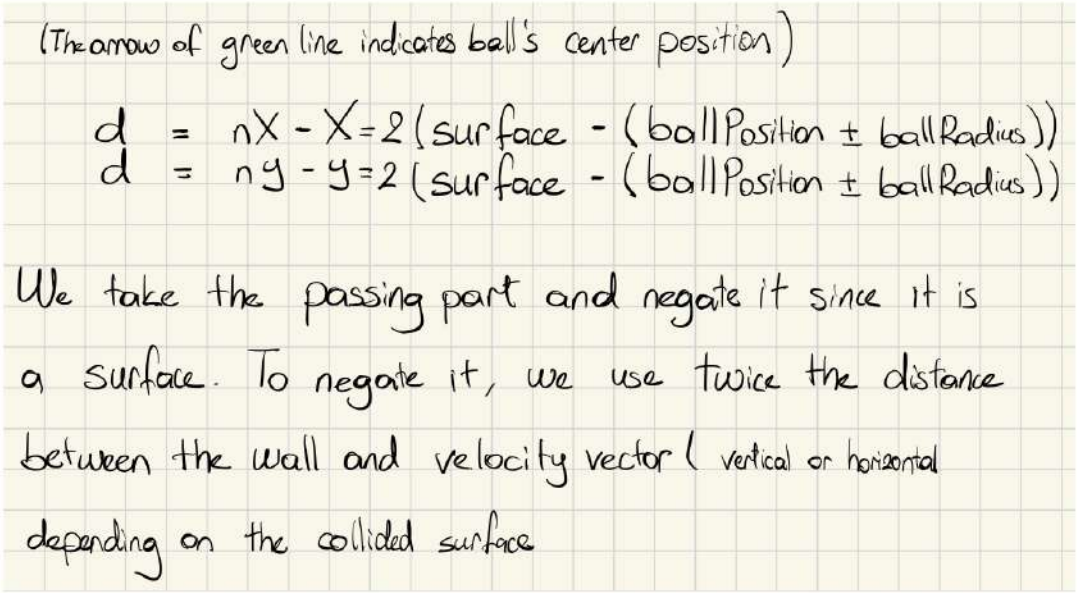


Figure 5:



8. cornerWillCollide(...)

- Determines if the **ball will collide with a brick corner** based on **Euclidean distance**.
- Uses the formula in Euclidean distance formula.
- If the distance is **less than or equal to the ball's radius**, and isDirection (isRight, isLeft etc.) is true, returns true.
- Otherwise, returns false.

Additive Information

This method is used to **fix a bug** in brick collision handling.

Issue:
Bricks are processed **from top-left to bottom-right**, so corner collisions on the **right and bottom** are often missed. The program **mistakenly treats them as surface collisions** with the first detected brick, which is then removed before the actual corner collision can be processed.

Solution:

- **Corner collisions are checked first** before removing bricks.
- Ensures that **both bricks involved in a corner collision** are considered.
- Prevents the issue where **right and bottom brick collisions are skipped**, making the physics more accurate.

Variable Positions on Canvas

Here's where the variables are positioned on the canvas:

1. **Ball**
    - **Position:** newBallPos[0], newBallPos[1]

    - **Radius:** ballRadius
    - **Color:** ballColor
2. **Paddle**
    - **Position:** paddlePos[0], paddlePos[1]
    - **Size:** paddleHalfwidth, paddleHalfheight
    - **Color:** paddleColor
3. **Bricks**
    - **Position:** brickCoordinates[i][0], brickCoordinates[i][1]
    - **Size:** brickHalfwidth, brickHalfheight
    - **Color:** brickColors[i]
4. **Score**
    - **Position:** (680, 380)
    - **Font:** Default
5. **Angle**
    - **Position:** (20, 380)
    - **Font:** Default
6. **Pause Message**
    - **Position:** (20, 380)
    - **Font:** Default
7. **Victory Message**
    - **Position:** (400, 100)
    - **Font:** Helvetica, **Bold**, Size: 40
8. **Game Over Message**
    - **Position:** (400, 100)
    - **Font:** Helvetica, **Bold**, Size: 40
9. **Final Score**
    - **Position:** (400, 70)
    - **Font:** Helvetica, **Plain**, Size: 20

Game Mechanics

The game consists of **two main phases**, each controlled by a separate **while loop**:

1. **Pre-Launch Phase:** (while (!isGameInitialized)) – Adjusts game mechanics before launching the ball.
2. **Gameplay & Animation Phase:** (while (!isGameOver && !isVictory)) – Runs the game animation until the player wins or loses.

1. First While Loop (while (!isGameInitialized))

At this stage, the game prepares for launch:

- **Screen Setup**: The screen is cleared, and the frame time is set.
- **Paddle Movement**:
    - The **left arrow key** moves the paddle **left**, and the **right arrow key** moves it **right**.
    - If the paddle reaches the edge of the frame, movement is restricted using inner if statements.
- **Game Start Command**:
    - Pressing **space** starts the game (isGameInitialized = true).

    - The key press is recorded (isSpacePressed = true).
- **Trajectory Line Calculation**:
    - The trajectory line vectors of the ball on x and y-axis are determined using sine and cosine functions.
    - A shootingLineEnd array stores the end coordinates of the trajectory line.
- **Drawing Game Elements**:
    - The **ball, trajectory line, paddle, bricks, angle, and score** are drawn on the screen using the **StdDraw** library.
    - A **for loop** places the bricks, using their coordinates stored in a **2D array**.
- **Rendering the Frame**: Once all elements are positioned, the **StdDraw.show()** method updates the screen, displaying the visuals.

Velocity values are assigned to **velocityOnX** and **velocityOnY** variables.

Inner Game Variables

- leftDistanceOnX: Distance between the ball and the **left edge** of a brick.
- rightDistanceOnX: Distance between the ball and the **right edge** of a brick.
- bottomDistanceOnY: Distance between the ball and the **bottom edge** of a brick.
- topDistanceOnY: Distance between the ball and the **top edge** of a brick.

If these distances meet certain conditions, it means the ball is colliding with the brick.

- leftDistancePaddleOnX: Distance between the ball and the **left edge** of the paddle.
- rightDistancePaddleOnX: Distance between the ball and the **right edge** of the paddle.
- distancePaddleOnY: Distance between the ball and the **top edge** of the paddle.

If these distances indicate overlap, the ball bounces off the paddle instead of falling.

2. Second While Loop (while (!isGameOver && !isVictory))

The second while loop is divided into two main parts: **Pause** and **Play** states.

Pause State

- When the **spacebar** is pressed and the previous press (isSpacePressed) has been reset, the **pause state toggles**—pausing the game if it was running or vice versa.
- The following if statement ensures that once the spacebar is released, the game is ready to detect the next press.

If the game is **paused**, the following actions take place:

- **Screen Setup**: The screen is cleared, and the frame time is set.
- **Drawing Game Elements**:
    - The **ball, paddle, bricks, pause message ("Paused"), and score** are drawn on the screen using the **StdDraw** library.
    - A **for loop** places the bricks, using their coordinates stored in a **2D array**.
- **Rendering the Frame**: Once all elements are positioned, the **StdDraw.show()** method updates the screen, displaying the visuals.

Play State

- **Screen Setup**: The screen is cleared, and the frame time is set.
- The **new ball position** is determined by adding velocity vectors on the **x** and **y** axes. The **theta** angle is calculated.

Paddle Movement and Collision Control

- The **right and left arrow keys** control the paddle's movement, and when it reaches the edge of the frame, its movement is restricted using **nested if statements**.
- Due to **frame skipping,** the paddle moves relatively faster than the ball, which may cause the ball to pass through it. To fix this, when the ball gets stuck inside the paddle, it is **immediately repositioned just above it** to prevent the issue.

Collision with Walls

- **Left Wall**: The ball's x position (newBallPos[0]) is checked to see if it collides with the left wall (i.e., if it's less than or equal to the ball radius). If it does, the ball's velocity on the X-axis is reversed (velocityOnX = -velocityOnX), and the ball's position is corrected to avoid it going inside the wall.
- **Right Wall**: Similarly, the ball's x position is checked against the right wall (newBallPos[0] >= xScale - ballRadius). If the ball is colliding with the right wall, the X velocity is reversed, and the ball's position is adjusted.
- **Ceiling**: The ball's y position is checked for collision with the top of the screen (newBallPos[1] >= yScale - ballRadius). If this happens, the Y velocity is reversed, and the ball's position is corrected to avoid being inside the ceiling.

Collision with the Paddle

- **Top of the Paddle**: The code checks if the ball's x position is within the width of the paddle and if its y position is close to the top of the paddle (based on its height). If these conditions are met, the ball's Y-velocity is reversed (bouncing off the top of the paddle), and the ball's position is adjusted to ensure it doesn't go inside the paddle.
- **Left and Right Sides of the Paddle**: The ball can collide with the left or right sides of the paddle. If the ball is near the paddle and within the appropriate range, its X-velocity is reversed (bouncing off the side of the paddle), and the ball's position is adjusted to avoid the ball being stuck inside the paddle.

Corner Collisions of the Paddle

- **Top Left Corner**: The distance between the ball and the top-left corner of the paddle is calculated using the Euclidean distance formula. And collision is detected through this formula and some inequalities (as shown in the Figure 6). If the ball is near this corner, the ball's reflection off the paddle is handled differently. Instead of a simple reverse of velocity, first, the portion going inside is calculated by portionFinder method and subtracted from ball's position. Then, the code calculates the bouncing angle (as shown in the Figure 7). The ball's velocity is adjusted to reflect the angle of collision at the corner, ensuring a realistic bounce off the corner. After the adjustments

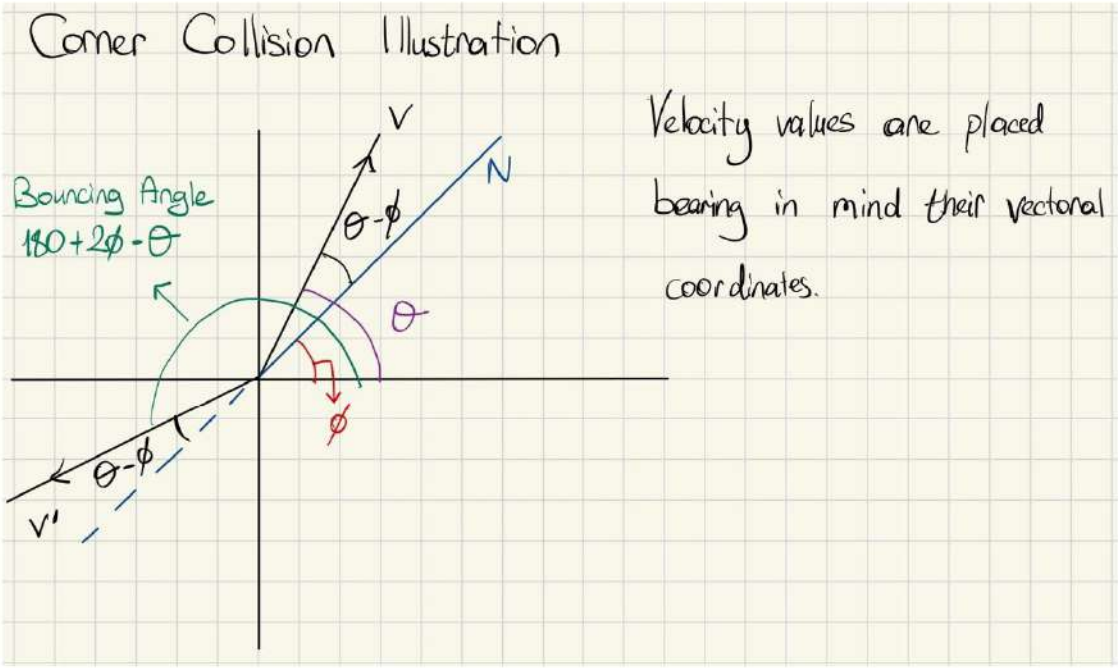
finished the portion of velocity is added to ball position, but being multiplied with the new velocity vector.

- **Top Right Corner**: A similar check and reflection mechanism is applied for the top-right corner of the paddle. The same bouncing angle reflection is calculated, and the ball's position and velocity are adjusted accordingly.

Figure 6:

```
// Top left corner
else if ((Math.sqrt(Math.pow(leftDistanceOnX,2) + Math.pow(topDistanceOnY,2)) <= ballRadius)
        && (newBallPos[0] <= brickCoordinates[i][0] - brickHalfwidth)
        && (newBallPos[1] >= brickCoordinates[i][1] + brickHalfheight)
        && (newBallPos[0] >= brickCoordinates[i][0] - brickHalfwidth - ballRadius)
        && (newBallPos[1] <= brickCoordinates[i][1] + brickHalfheight + ballRadius)){
```

Figure 7:



Brick Collision Mechanics

- A for loop iterates through each brick to check for collisions.
- **Phi and theta angles** are calculated to determine the ball's trajectory upon impact.
- The code checks whether a brick side by side exists in each direction by comparing its coordinates with those stored in the brickCoordinates array.
- If the brick was already hit, the loop moves to the next element.
- If the ball collides with the **walls** or the **brick's surface** (excluding its corners), the corresponding velocity component is inverted as in the case of paddle and the wall:

- If the ball hits the **left or right** side of the brick, velocityOnX is negated while maintaining the same magnitude.
- If the ball hits the **top or bottom**, velocityOnY is negated.

- The distanceFinder method ensures the correct reflection behavior during these interactions as in the paddle corner collision.
- When the ball hits a **brick corner**, the exact corner is identified as in the case of paddle corner collision (similar as in the Figure 6).
- The program then checks if there is an adjacent brick near the corner:
  - If **there is a neighboring brick**, the collision is treated as a **surface collision**, and **both bricks are removed**. The player's score increases by double (10 * 2 = 20 in the standard edition).
  - If **there is no neighboring brick**, the code follows the **else statement** (as shown in the Figure 8).

Figure 8:

```
else {
    // Determining brick's corners.
    double brickCornerX = -brickHalfwidth + brickCoordinates[i][0];
    double brickCornerY = brickHalfheight + brickCoordinates[i][1];
    // Finding the portion which will be reflected.
    double portion = portionFinder(newBallPos,velocityOnX,velocityOnY,
            ballRadius,brickCornerX,brickCornerY);
    // Preventing ball to go in the brick during collision by subtracting the distance having gone inside.
    newBallPos[0] -= (portion * velocityOnX);
    newBallPos[1] -= (portion * velocityOnY);
    // Calculating new velocities after a collision.
    phi = phiFinder(newBallPos,brickCoordinates[i],brickHalfheight,brickHalfwidth);
    newVelocityOnX = ballVelocity * Math.cos(Math.PI + 2 * phi - theta);
    newVelocityOnY = ballVelocity * Math.sin(Math.PI + 2 * phi - theta);
    velocityOnX = newVelocityOnX;
    velocityOnY = newVelocityOnY;
    // Adding subtracted portion to collided ball.
    newBallPos[0] += (portion * velocityOnX);
    newBallPos[1] += (portion * velocityOnY);
    // Marking the brick as hit and adding score for one brick to score of the player.
    brickHitStatus[i] = true;
    scoreOfPlayer += scoreForOneBrick;
}
```

- In the else statement, the portionFinder method calculates how much of the velocity component moved the ball into the brick. This portion is then subtracted from both the **x and y coordinates** of the ball.
- After these adjustments, the **theta and phi angles** are determined.
- The new velocity is calculated based on these angles (as shown in the Figure 7).
- The ball moves with the adjusted velocity after adding the subtracted portion of velocity and multiplying it with the new velocity vector.
- Since the ball **only collides with one brick**, a **single brick is marked as hit**, and the player's score increases by the **value of one brick**.
- During side collisions, the cornerWillCollide method is used to prevent **bugs** caused by the execution order.

Physics Behind Reflections

- **Bouncing Angle**: The key to making the ball's bounce off the paddle look natural is the calculation of the reflection angle. The phiFinder and thetaFinder method calculates the bouncing angle and this angle is used to determine the new velocity of the ball after bouncing off the paddle.
- **Velocity Change**: After a collision with any part of the paddle, the ball's velocity (both velocityOnX and velocityOnY) is updated based on the bouncing angle, ensuring the ball bounces realistically.

Loop Exit Conditions

- Once the ball **hits the floor**, the game **ends** (isGameOver = true).
- If all bricks are **removed**, the player **wins** (isVictory = true).

After all adjustments are made, the following steps occur:

- **Drawing Game Elements**:
  - The **ball, paddle, bricks, and score** are drawn on the screen using the **StdDraw** library.
  - A **for loop** places the bricks, using their coordinates stored in a **2D array**.
- **Rendering the Frame**: Once all elements are positioned, the **StdDraw.show()** method updates the screen, displaying the visuals.

Game Over and Victory Conditions

Game Over

- **Screen Setup**: The screen is cleared.
- **Drawing Game Elements**:
  - The **ball, paddle, remaining bricks, game over text message ("GAME OVER !"), and score** are drawn on the screen using the **StdDraw** library.
  - A **for loop** places the bricks, using their coordinates stored in a **2D array**.
- **Rendering the Frame**: Once all elements are positioned, the **StdDraw.show()** method updates the screen, displaying the visuals.

Victory

- **Screen Setup**: The screen is cleared.
- **Drawing Game Elements**:
  - The **ball, paddle, victory text message ("VICTORY !"), and score** are drawn on the screen using the **StdDraw** library.
  - A **for loop** places the bricks, using their coordinates stored in a **2D array**.
- **Rendering the Frame**: Once all elements are positioned, the **StdDraw.show()** method updates the screen, displaying the visuals.

Modified Version Features

Two arrays of random numbers are created:

- **randomColor** (values between **32 and 256**) is used for the **green color scale** to ensure a greenish color.
- **randomColorLowBounded** (values between **0 and 32**) is used to create a **low effect of blue and red**, keeping the overall tone predominantly green.

Using these random values, **six colors** are generated and stored in the colors array. Since the numbers are randomly assigned, the brick colors change with every gameplay.

Superpowers

Six unique superpowers are implemented in the game:

1. **Eliminate 1-5 Bricks (Randomly eliminates, random numbers of bricks)**
2. **Paddle Width x2**
3. **Paddle Width +5**
4. **Ball Radius x2**
5. **Ball Radius +5**
6. **Score x2**

These powers are placed at predefined coordinates, and **two of them are randomly selected** for each game session. If the same power is selected twice, one instance is replaced with the **next superpower in order**. For example, if Ball Radius +5 is chosen twice, one will be replaced with Score x2.

To track whether a superpower has been used, a **boolean array (usedEffects) with six elements** is maintained. During gameplay:

- The selected superpowers are displayed with a text description.
- Each superpower is represented by a **colored circle**, with the color indicating its effect.
    - **Paddle-related powers** are **grey**,
    - **Ball-related powers** are **blue**.
- If the ball **passes through** a superpower circle, the power is activated, the corresponding circle and text are removed, and usedEffects[i] is set to true.

For the **brick elimination** power, how many bricks will be removed randomly 1 to 5 is determined. The game ensures that **only bricks that haven't been hit yet** are eliminated, maintaining fair gameplay.

Brief Summary of Code Visualization

Figure 9:

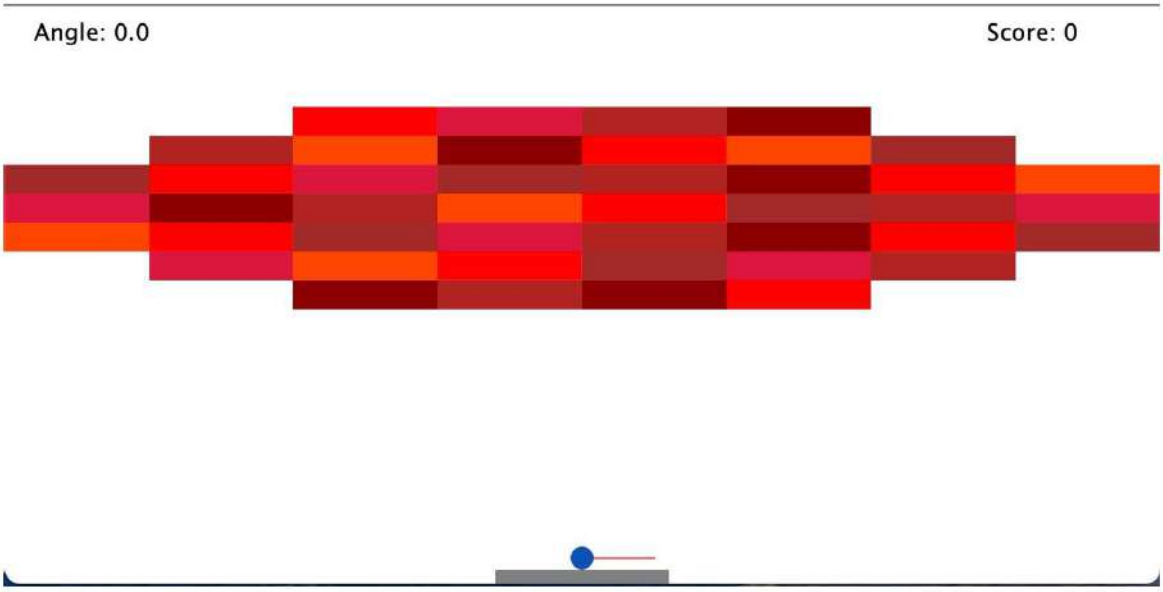


Figure 10:

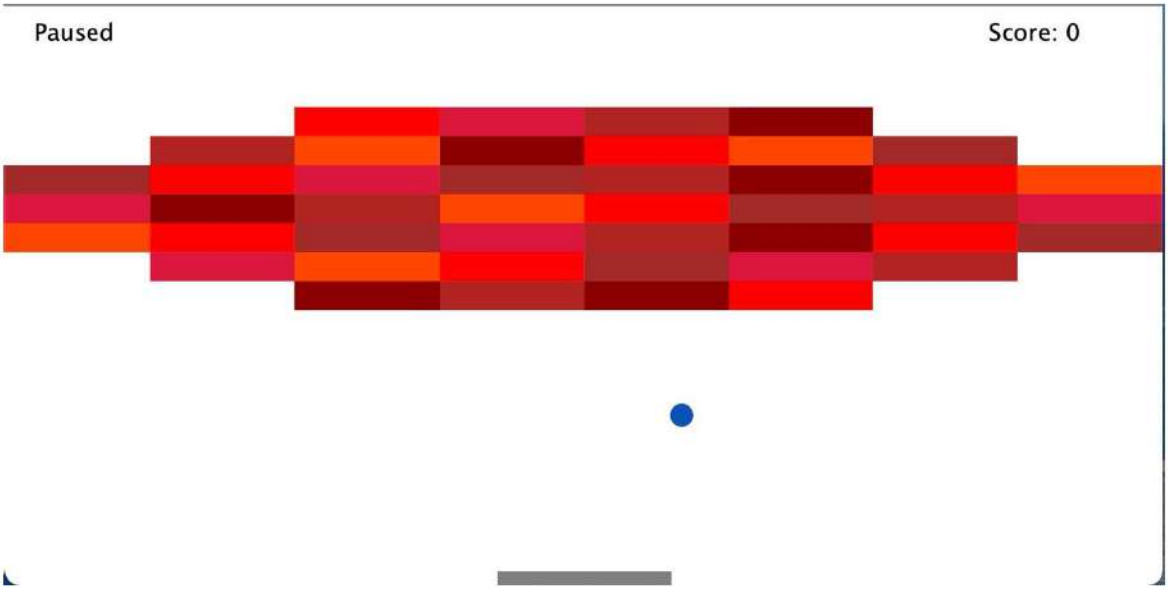


Figure 11:

Figure 12:

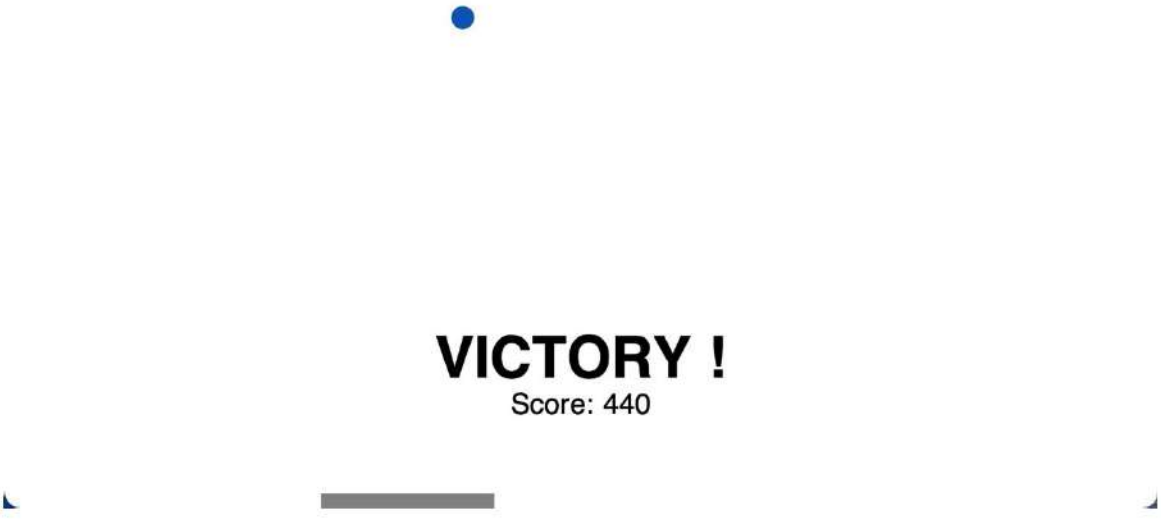


Figure 9 shows the starting screen, Figure 10 shows the paused screen, Figure 11 shows game over screen and Figure 12 shows victory screen.

YouTube Video Links

Standard Version: https://youtu.be/18tqnnZKj6s

Modified Version: https://youtu.be/tkw1vsDKxok