Mehmet Arda Kutlu

DX-BALL Game Report

https://youtu.be/DxoPygk_JwY

https://youtu.be/PGfl_xhDdUw

Introduction

About Dx-Ball

DX-Ball is a simple 2D computer game based on basic physics principles. It consists of three main components: a ball, a paddle, and a set of bricks. The objective is to clear all the bricks from the game area. Initially, the ball rests above the paddle, and the game begins when the player launches it. The ball moves in a straight path until it collides with a wall, a brick, or the paddle, at which point it bounces off, following the rules of elastic collision. When the ball strikes a brick, the brick disappears and the player earns points. As the ball continues to move, the player must prevent it from leaving the game area by controlling the paddle. The game ends in victory if all the bricks are eliminated or in defeat if the player fails to intercept the ball, allowing it to hit the bottom surface.

About StdDraw

StdDraw is a graphics library developed by Princeton University that allows users to create drawings using basic geometric shapes, text, and simple animations. It also supports user interaction by handling keyboard and mouse inputs, making it an ideal tool for creating interactive graphics.

Game Mechanics

Setting the Properties of the Fundamental Game Components

Canvas: Since all gameplay takes place within the game area, defining it is the first step. In **StdDraw**, this game area is referred to as the **canvas**. The canvas size is set using the setCanvasSize method, ensuring a well-defined space for rendering game objects.

X and Y Scale: StdDraw positions all objects using a coordinate system. By default, both the x-axis and y-axis range from 0.0 to 1.0, which can make precise positioning difficult. To simplify this, the **xScale** and **yScale** values are adjusted to match the actual dimensions of the canvas, which are 800 and 400, respectively. This allows for more intuitive placement of game elements.

Ball Properties: The ball's properties are then defined. The radius, speed, and initial position are set, as the position and radius are essential for forming a circle. Additionally, brick colors are selected from different shades of red, ensuring clear visual distinction between bricks.

Paddle Properties: The paddle, which is rectangular, has predefined height, width, and initial position. These three values are crucial for rendering the paddle correctly. The paddle's velocity is also specified, restricting its movement to the x-axis only. To maintain visual clarity, the paddle color is set to gray.

Brick Properties: Like the paddle, bricks are rectangular with predefined width and height. Their positions are stored in the **brickCoordinates** array, while their colors are managed using the **brickColors** array. Additionally, an **isBricksBroken** array tracks whether each brick has been cleared. This array is initialized with false values, indicating that all bricks are intact at the start of the game.

- **enableDoubleBuffering** command is executed to eliminate screen flickering and provide smoother animations.
- **pauseDuration** determines the time limit between the two frames in milliseconds. At the values below 25, I encountered a screen flickering problem, so its value is set to 25, which corresponds to 40 frames per second.

Some Significant Variables

- **isGameStarted:** Determines whether the game has started. It becomes true when the player presses the space key to launch the ball.
- **isGameLost:** Tracks whether the game is lost. It turns true when the ball hits the bottom surface.
- **isGameWon:** Indicates if the game is won. This happens when all bricks are broken.
- **isGamePaused:** Checks if the player pressed the space key in order to pause the game.
- **isSpaceCooledDown:** Ensures that at least 0.2 seconds have passed since the last space key press, preventing unintended rapid inputs.
- **spaceChecker:** Monitors whether the player is holding down the space key.
- **bricksBroken:** Keeps track of the number of broken bricks. When it becomes equal with the number of total bricks, **isGameWon** turns true.
- **score:** Stores the player's score, which increases as bricks are broken.
- **spaceCooldownTime:** A timer controlling the **isSpaceCooledDown** variable. It is initialized to 8 and decreases by 1 per frame. Since each frame lasts 25 ms, this ensures a 0.2-second cooldown between consecutive space key presses.

Initial Game

Components

Trajectory Line: A trajectory line is displayed to help the player visualize the ball's initial path. This line originates from the center of the ball and extends outward, providing a directional clue before launch.

Angle Information: The trajectory angle represents the angle between the trajectory line and the positive x-axis, measured in the counterclockwise direction. Its value (in degrees) is displayed at the top-left corner of the screen. The variable **thetaDegrees** is used to store this angle information.

finalLinePosx and finalLinePosy: These two variables store the end coordinates of the trajectory line. They are initially set such that the angle will be 0 degrees.

- Initial game mainly consists of a while loop whose parameter is **!isGameStarted**. Since **isGameStarted** will turn true when the player hits the enter key, the main game will not start until this moment.

Generating Frames

clear(color): Wipes out all the drawings from the screen and makes screen ready for a new frame.

Drawing the Paddle: **setPenColor(color)** picks the desired color and **filledRectangle(x, y , width/2, height/2)** draws the rectangle(paddle).

Drawing the Bricks: It is the same process as the paddle. The difference is, a for loop is used to obtain all the color and position information from the predefined arrays **brickCoordinates** and **brickColors**.

Drawing the Ball: **filledCircle(x, y, radius)** draws the ball. Ball's position values are stored in **initialBallPos** array and its color is given by **ballColor**.

Drawing the Trajectory Line: **setPenRadius(radius)** is used to determine the thickness of the line. The lines is drawn by **line(x initial, y initial, x final, y final)**.

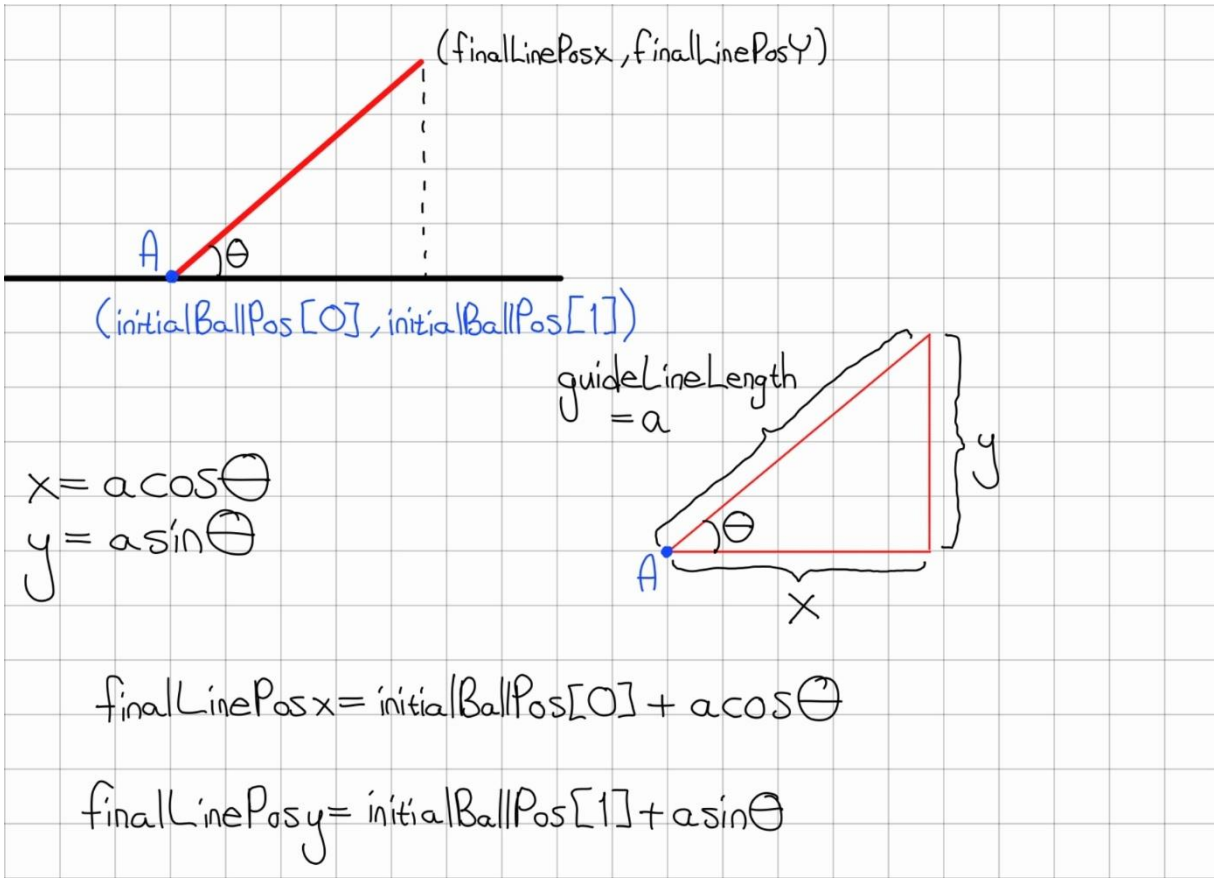
Writing the Angle Information: **setFont(font, font, font size)** specifies the size and format of the text and **textLeft(x, y, text)** writes a left-aligned text to the specified position.

Changing the Angle by Taking User Input

isKeyPressed(key) is used to obtain keyboard input from the user. This method returns true if the specified key is pressed. By using this method with an if statement, variable **thetaDegrees** can be incremented or decremented by one. **KeyEvent.VK_LEFT** stands for the left arrow and **KeyEvent.VK_RIGHT** stands for the right arrow. While changing the value, borders **thetaDegrees < 180** and **thetaDegrees > 0** must be defined to prevent user from exceeding the angle limits.

The Math Between the Drawing of The Line

Basic trigonometry is needed to steer the trajectory line properly. When the angle is changed, the updated endpoint coordinates must be found.




```
finalLinePos_x = guideLineLength * Math.cos(Math.toRadians(thetaDegrees)) + initialBallPos[0];
finalLinePos_y = guideLineLength * Math.sin(Math.toRadians(thetaDegrees)) + initialBallPos[1];
```

Breaking the While Loop: When the user hits the space key, **isGameStarted** will turn into true, allowing the player to exit the while loop and continue to main game. Otherwise, the code will constantly display the drawings that have been made by using **show()** method and pauses before displaying the new frame for 25ms by using **pause()**. After quitting the loop, x and y components of the ball's velocity is calculated using the final **thetaDegrees** value.

```
double ballVelocity_x = ballVelocity * Math.cos(Math.toRadians(thetaDegrees));
double ballVelocity_y = ballVelocity * Math.sin(Math.toRadians(thetaDegrees));
```

Main Body of the Game

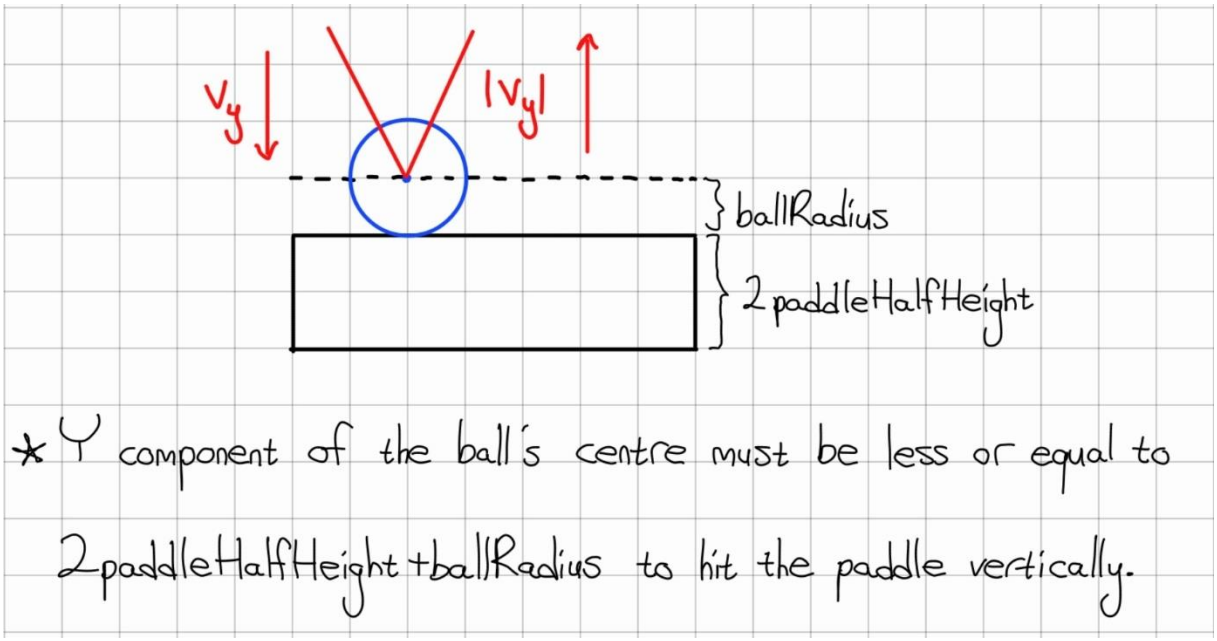
Moving the Ball: At the beginning of each frame, coordinates of the ball are updated by adding the velocity values. Velocity values **ballVelocity_x** and **ballVelocity_y** are subject to change during the collisions.

Paddle Collisions

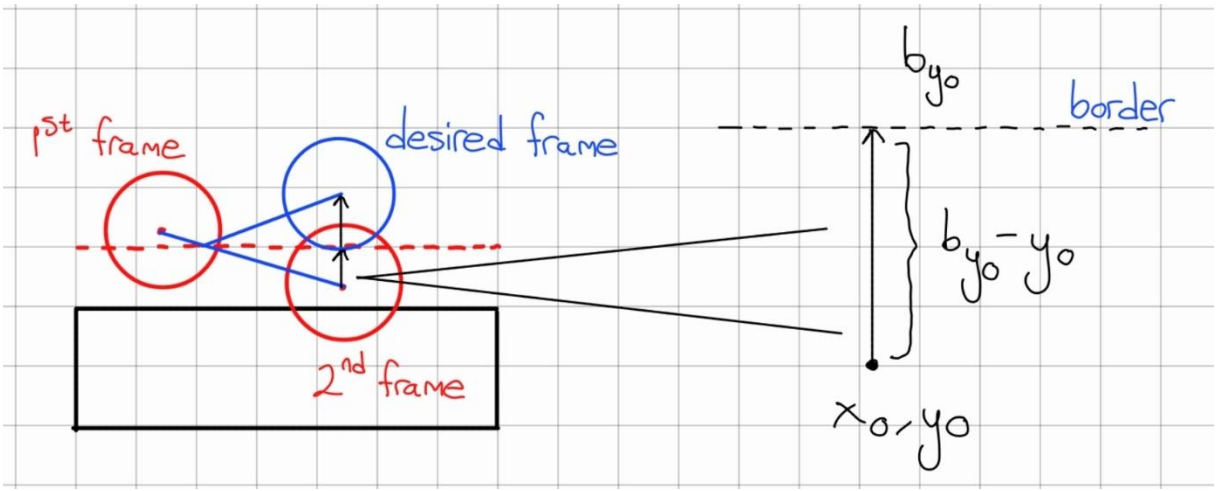
1. Vertical Collision

If the ball approaches to the horizontal surface of the paddle enough, it will bounce back to upwards. By elastic collision laws, **ballVelocity_y** must point upwards after the collision, which implies:

ballVelocity_y = Math.abs(ballVelocity_y)

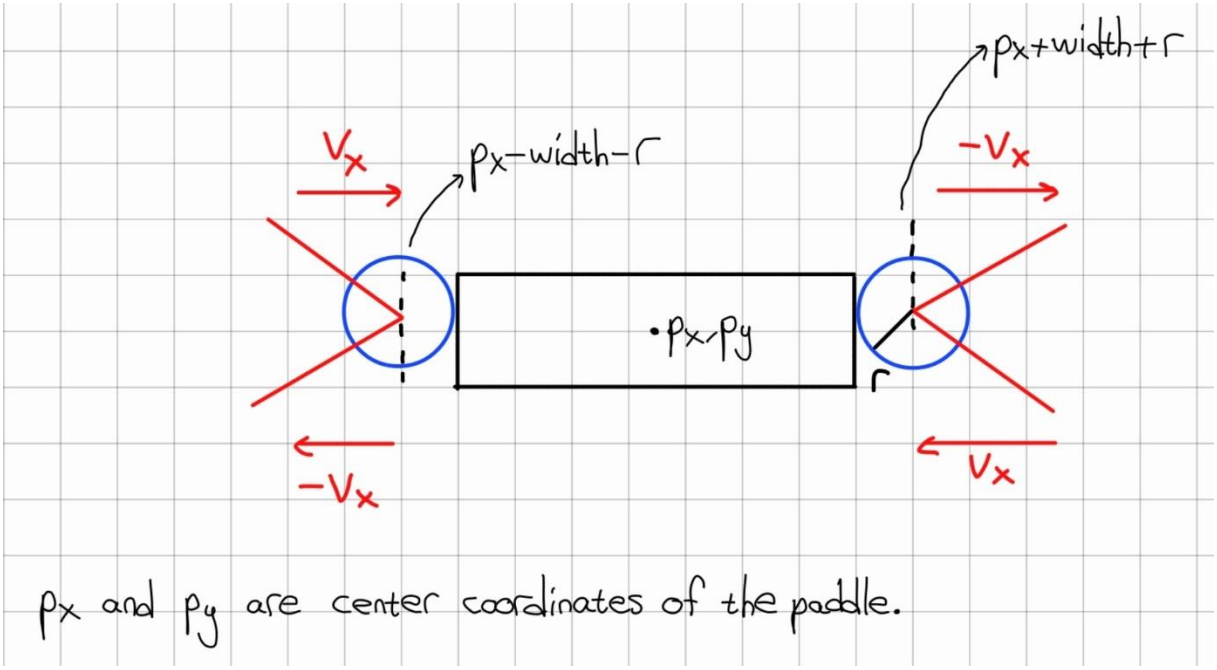


Visual Improvements: Because the ball moves frame by frame, some parts of the ball may enter the paddle, which is an undesired situation. To hinder this issue, ball should be reflected from the collision border.



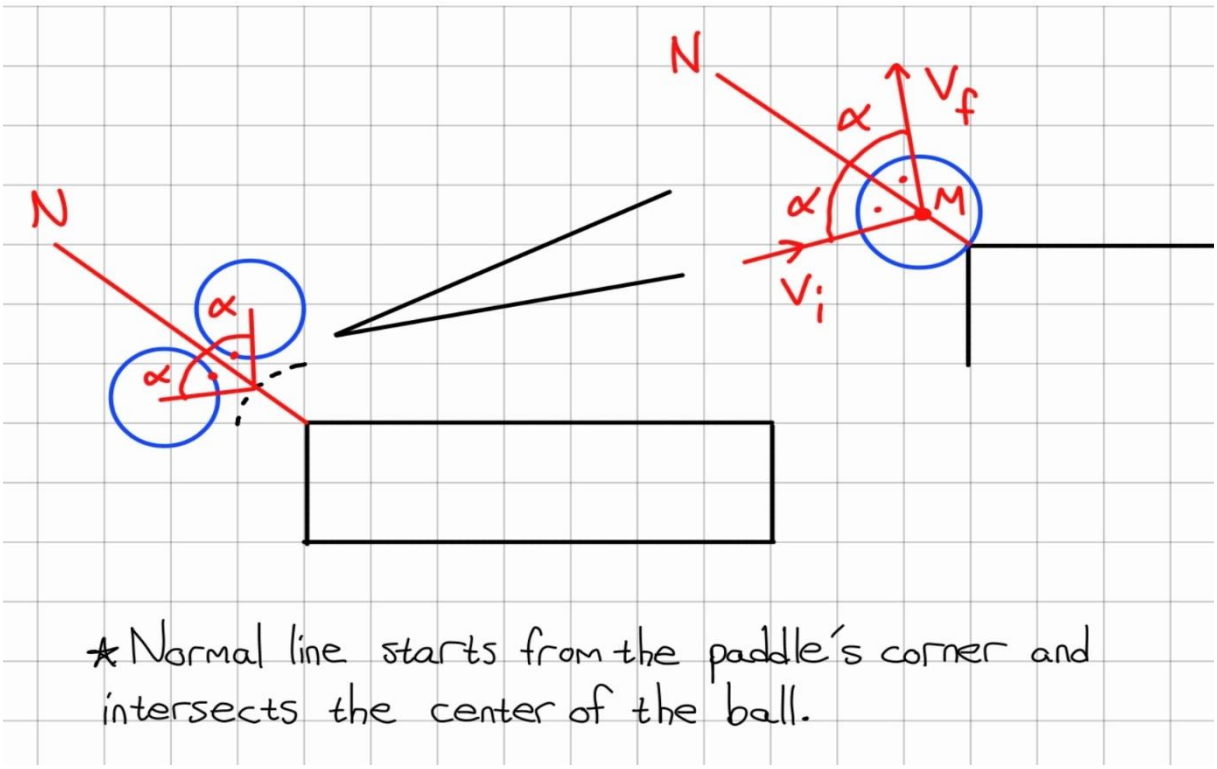
2. Horizontal Collision

When the ball hits the sides of the paddle, according to elastic collision principles, after the collision, **ballVelocity_x** must change sign. Visual improvements that are made in vertical collision are also done at this collisions and the subsequent collision types.

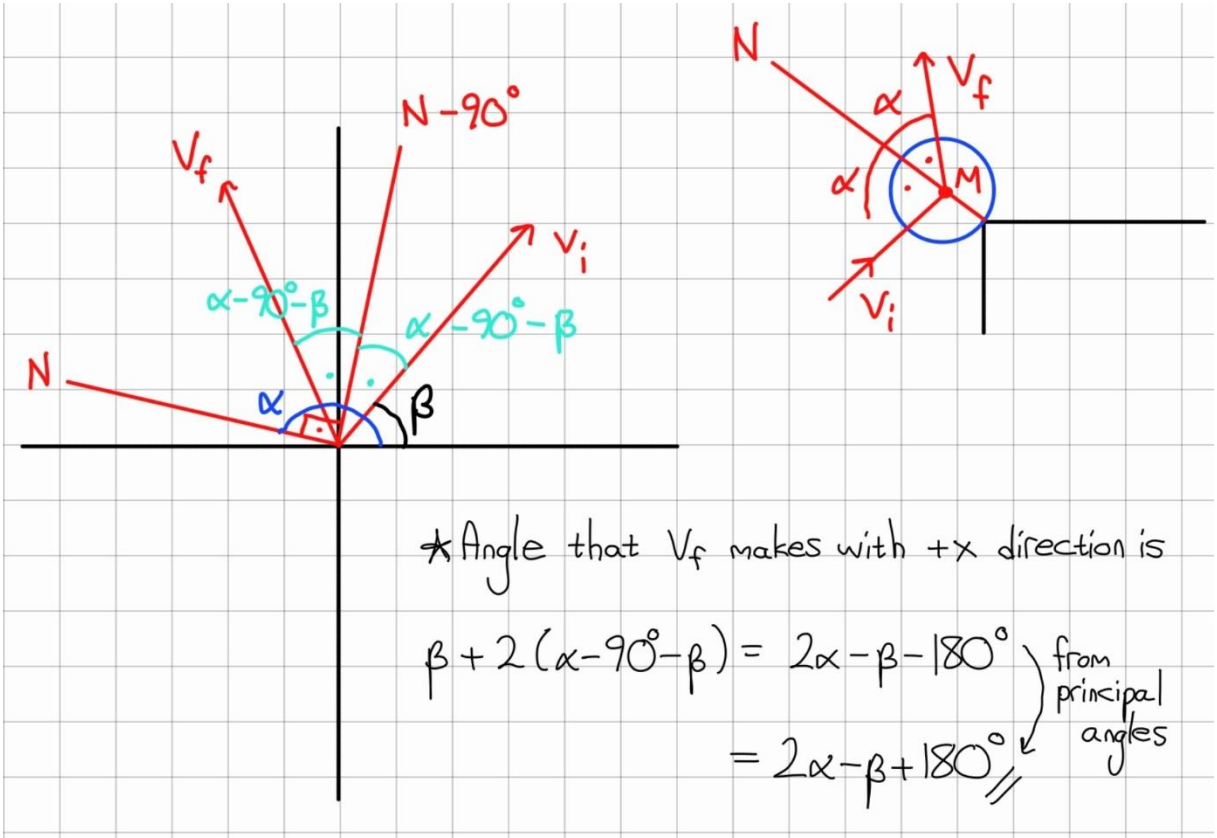


3. Corner Collision

Corner collision may be the most challenging collision type among these three. This is because the normal lines at these collisions are not the same, they depend on the ball's position. One of the possible approaches to cope with this complex circumstance is plotting the normal line and the initial and the final velocity vectors to a coordinate system and observing the principal angles.



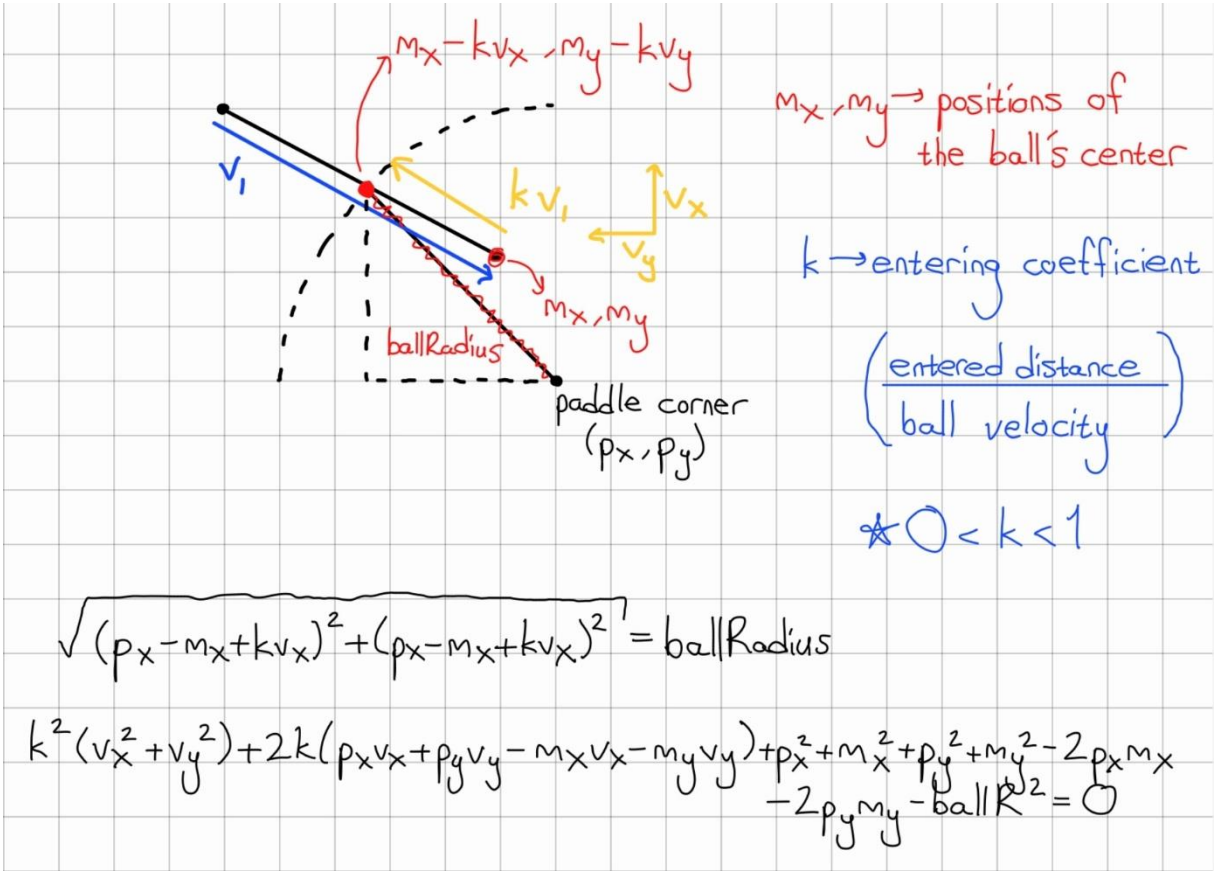
Corner collision case



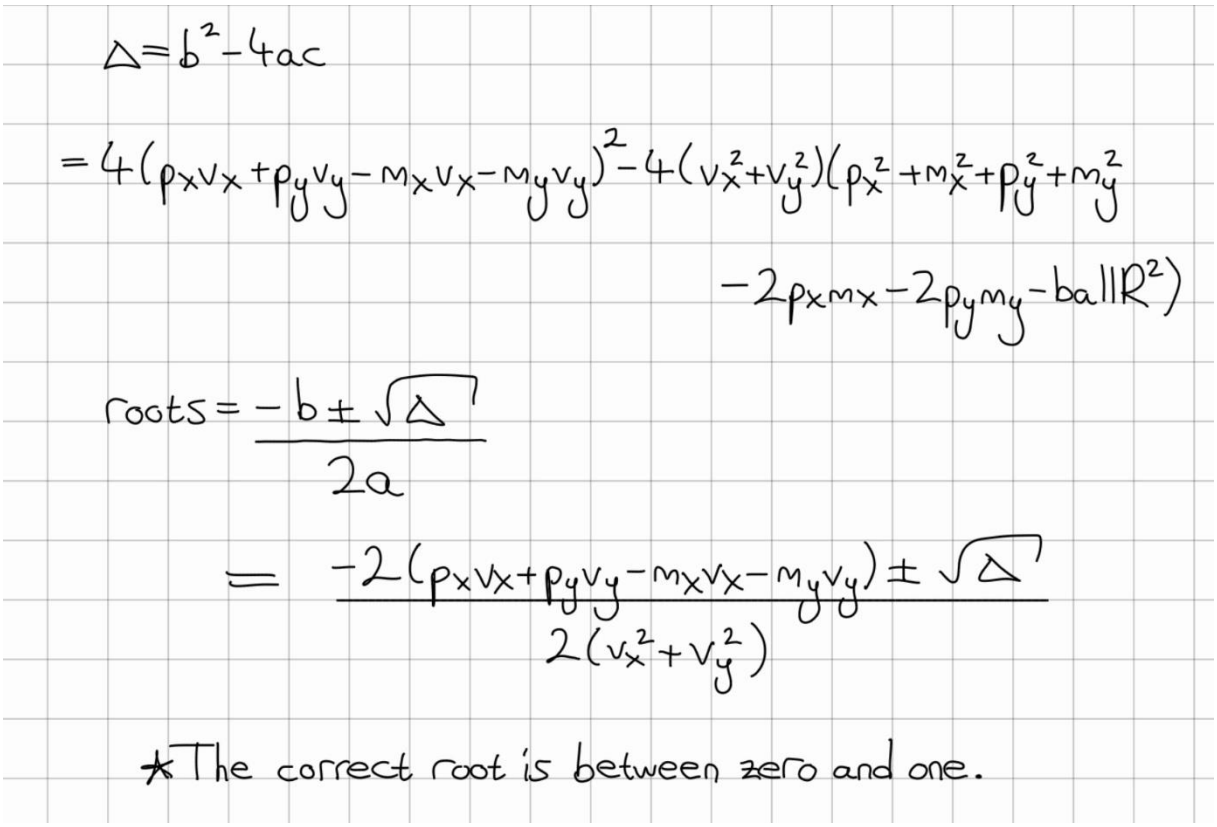
The approach used in this code to find the final velocity by determining its angle.

Preventing the Ball from Entering the Paddle

The common feature of the three types of collision is that the ball may enter the paddle slightly. However, while at the first two types, ball entering only causes an unpleasant looking frame, at the corner collision, it results in an incorrect direction change since the normal line changes with the ball's position. To prevent this, the ball must be brought back to the state in which it touches the ball. Then the velocities are updated and finally the ball reflects from the corner. Finding the touching point requires complex mathematical operations.



Deriving the equation for finding the distance travelled after the entering.



Solving the second-degree equation.

Brick Collisions

Handling the bricks are easier than the paddle since the bricks are stationary. However, some cases that are neglectable in paddle collisions should be considered, which are defining the bottom border of the bricks in vertical collisions and bottom left and bottom right corner borders in corner collisions.

- **Vertical collisions:** The y component of the ball is redirected depending on whether the ball hits the bottom or the top surface. The collision mechanics and the visual improvements are done in the same way as the paddle collision.
- **Horizontal collisions:** The x component of the ball is reversed depending on the collision side. This collision is virtually the same as the paddle horizontal collision.
- **Corner Collisions:** All of the multifaceted operations that are performed in the paddle version are also done in the bricks. The main difference is that instead of top corners, all of the corners are considered.

After the collision, the brick(s) are marked as cleared in **isBricksBroken** array. Elimination of each brick earns the player 10 points and makes him one step closer to the victory.

Wall Collisions

Wall collisions are much easier than paddle and brick coordinates since a corner collision is impossible. There are three types of wall collisions.

Left Wall: x component of the ball's velocity redirects to right, using the coordinate system logic:

ballVelocity_x = Math.abs(ball velocity_x)

Right Wall: It's just the reverse of the left wall:

ballVelocity_x = - Math.abs(ball velocity_x)

Top Wall: Instead of the x component, the y component changes and points out downwards:

ballVelocity_y = - Math.abs(ball velocity_y)

Bottom Wall: Hitting the bottom wall is not a desired case and there is no need to consider a calculation since it means that the game is lost. When the ball touches the bottom wall, which can be shown as:

initialballPosition[0] <= ballRadius

the main while loop is broken, **isGameLost** returns true and the player is redirected to the game over screen.

Paddle Movement: When the right or left arrow keys are pressed, paddle moves along the x-axis by **paddleSpeed** units. When the right side of the paddle hits the right wall, paddle stops moving to +x direction, and vice versa in the -x direction.

Handling Paddle Bug

There is a troublesome fact that **paddleSpeed** is 20, which is four times greater than the **ballVelocity**. Considering that ball's x component of velocity is even less than 5, there is a huge gap between the velocities, which gives rise to a bug that occurs when the paddle is moved towards the ball. In this case, the ball may be swallowed by the paddle, and this case is not handled in any collision cases. To prevent this bug, a code segment is added to teleport the ball above the paddle and set the y component of velocity to point out upwards, which brings back the ball "back to the life".

```
if(Math.abs(initialBallPos[0] - paddlePos[0]) < paddleHalfwidth &&
Math.abs(initialBallPos[1] - paddlePos[1]) < paddleHalfheight){
    initialBallPos[1] = 2 * paddleHalfheight + ballRadius;
    ballVelocity_y = Math.abs(ballVelocity_y);
}
```

Drawing and Displaying the Frames in the Main Game

The logic behind the frame construction in the initial game and the main game is very similar. The visible differences are that the trajectory line and the angle information at the top left is not present in the main game. Instead, there is a score indicator at the top right and the game status part at the top left, which is empty while the game is continuing.

Pause Mechanism

The code handles the pause mechanism by using two fundamental components. An **isGamePaused** boolean variable and a while loop whose parameter is **isGamePaused.** When the player hits the space key, **isGamePaused** returns true, and the code enters the pause loop. In the pause loop, all of the game components are stationary and a **"PAUSED"** text at the top left indicates that the game is paused. When the player hits the space key again, **isGamePaused** becomes false again, resulting in the break of the pause loop and the code turns back to the main loop. Two parameters are defined to prevent repeated space hits to constantly change game status.

spaceCooldownTime: While this integer parameter is not equal to zero, **isSpaceCooledDown** remains false. In that case, the game will not respond to space hits until the **spaceCooldownTime** equals 0 and **isSpaceCooledDown** returns true. Pressing the space key resets the timer to 8, which corresponds to 200ms.

spaceChecker: When the player hits the enter key, **SpaceChecker** returns true, and it remains as true until the **isKeyPressed(KeyEvent.VK_SPACE)** method returns false, which means that if the player presses the space key and holds it down, the game will not change status. This variable helps code perceive the key hold downs as single presses, making the pause mechanism more natural.

Final Messages

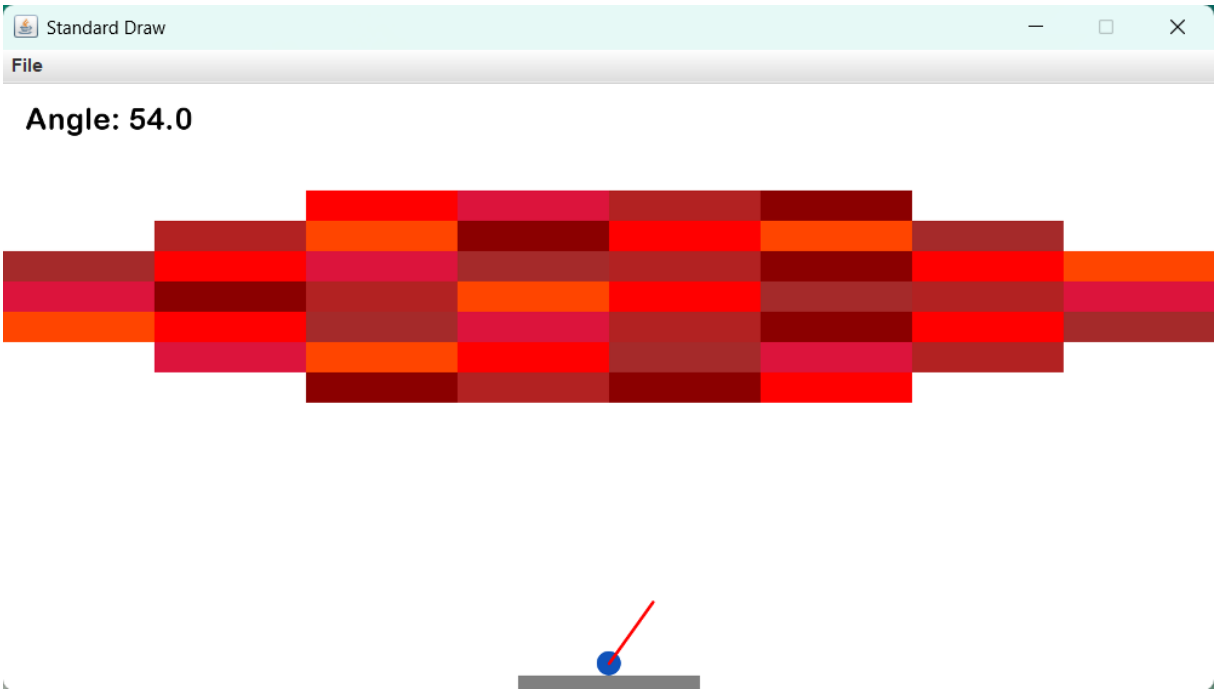
Victory

As mentioned before, clearing all the bricks turns **isGameWon** to true and the victory message appears on the screen. A huge "VICTORY!" message is displayed, and the score of the player is shown below the victory message. The ball remains at the screen at its last position.

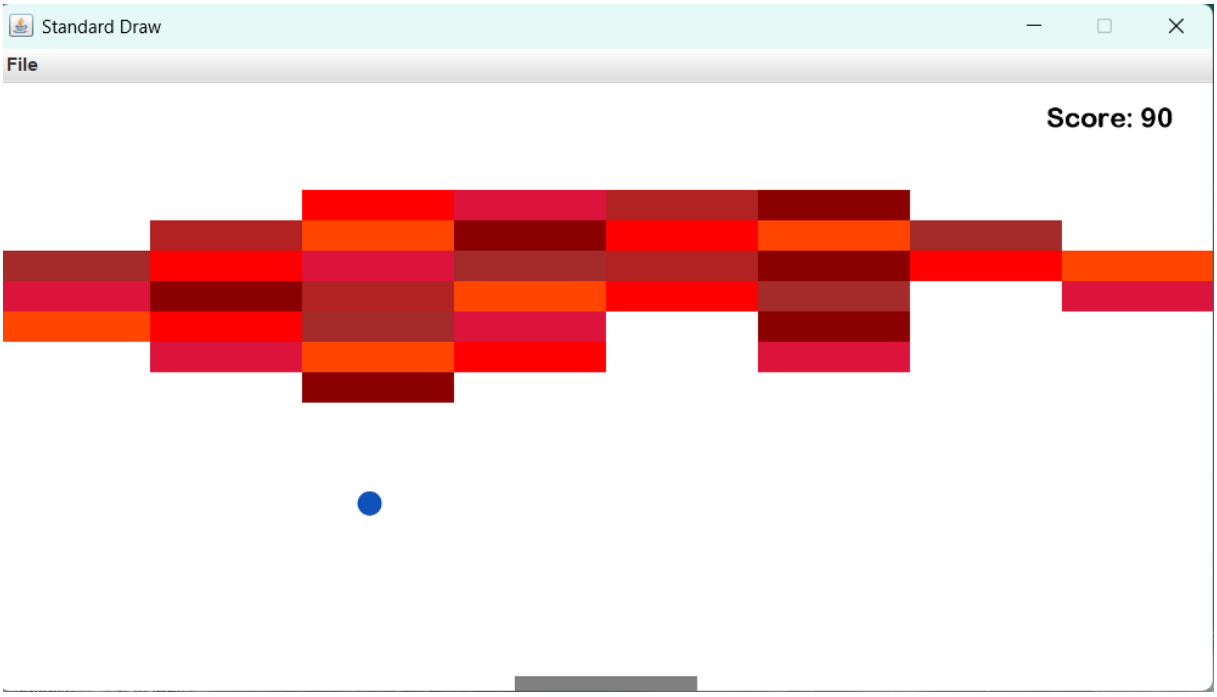
Loss

When the ball touches the bottom surface, **isGameLost** becomes true and the game over message appears on the screen. A huge "GAME OVER!" message is displayed, and the score of the player is shown below the game over message. The intact bricks and the ball is displayed on the background.

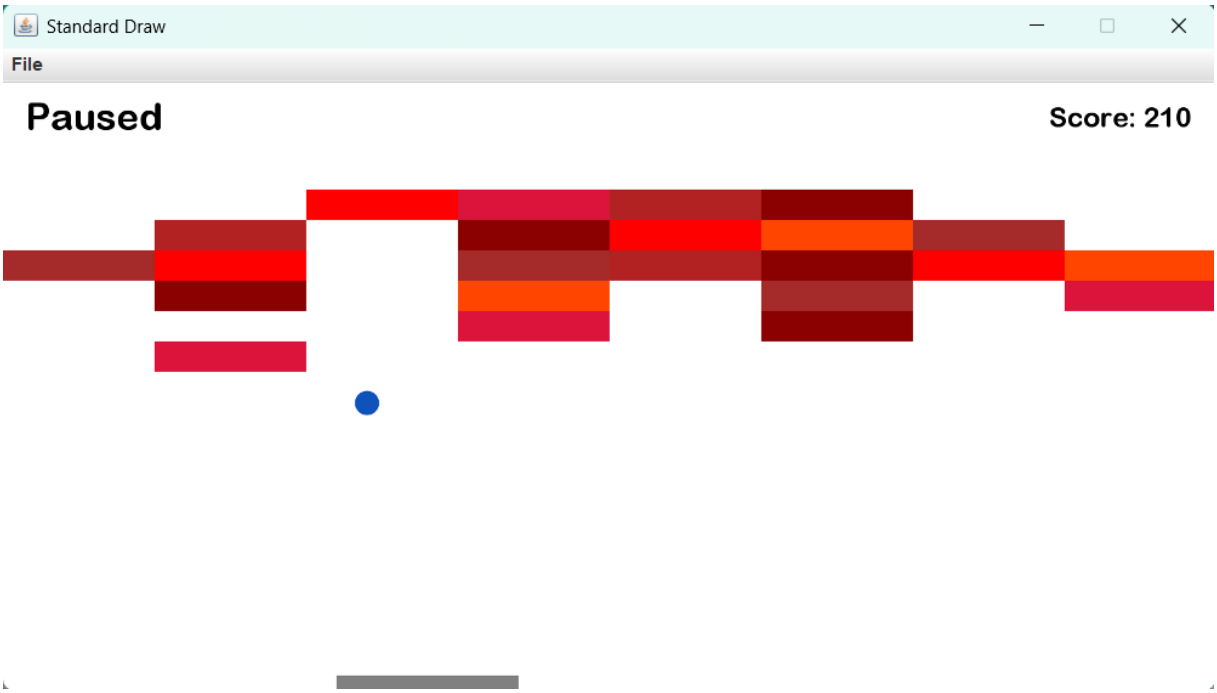
Screenshots



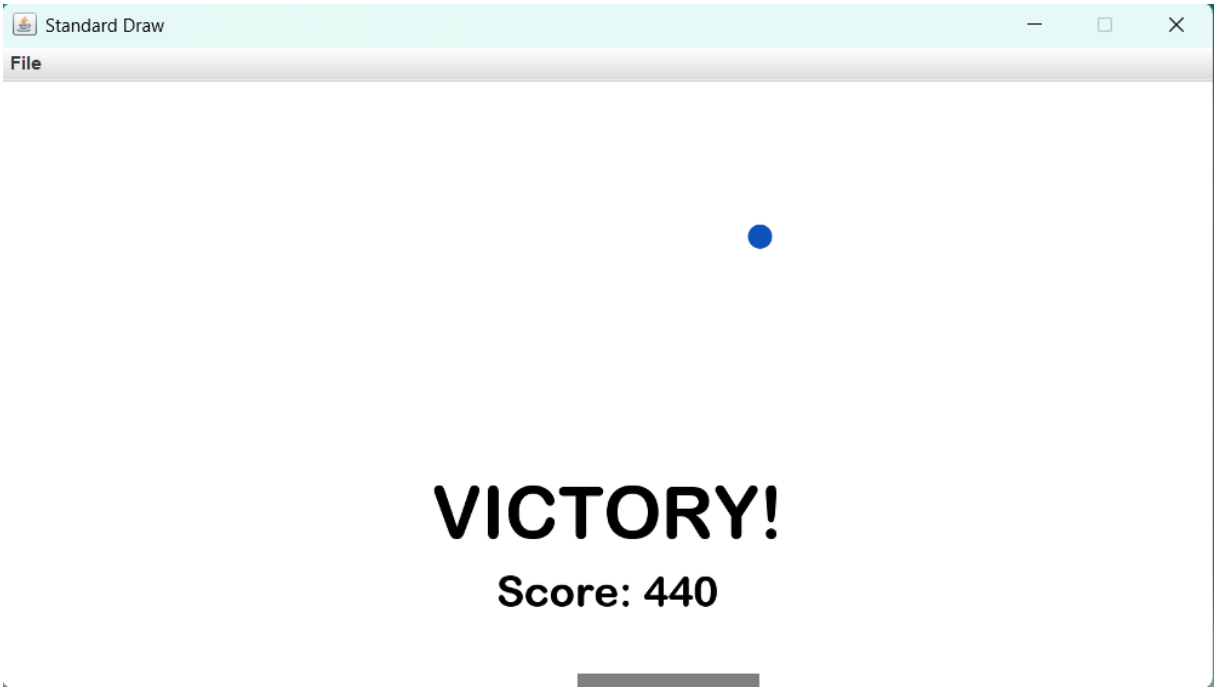
Initial game



Continuing Game



Paused Game



Victory screen

Game over screen

My Version of the Game

In my version, I introduced several key gameplay and brick placement changes to make the experience more dynamic and engaging. These modifications enhance both the game's visual appeal and the player's willingness to play the game multiple times.

Colors

The standard version of the game uses monotone colors, which may feel boring. To create a more vivid and immersive experience, I adjusted the color scheme:

- The paddle is now dark blue.
- The ball is orange for better visibility.
- Standard bricks feature nine different shades of green, making them more visually distinct.

Maps

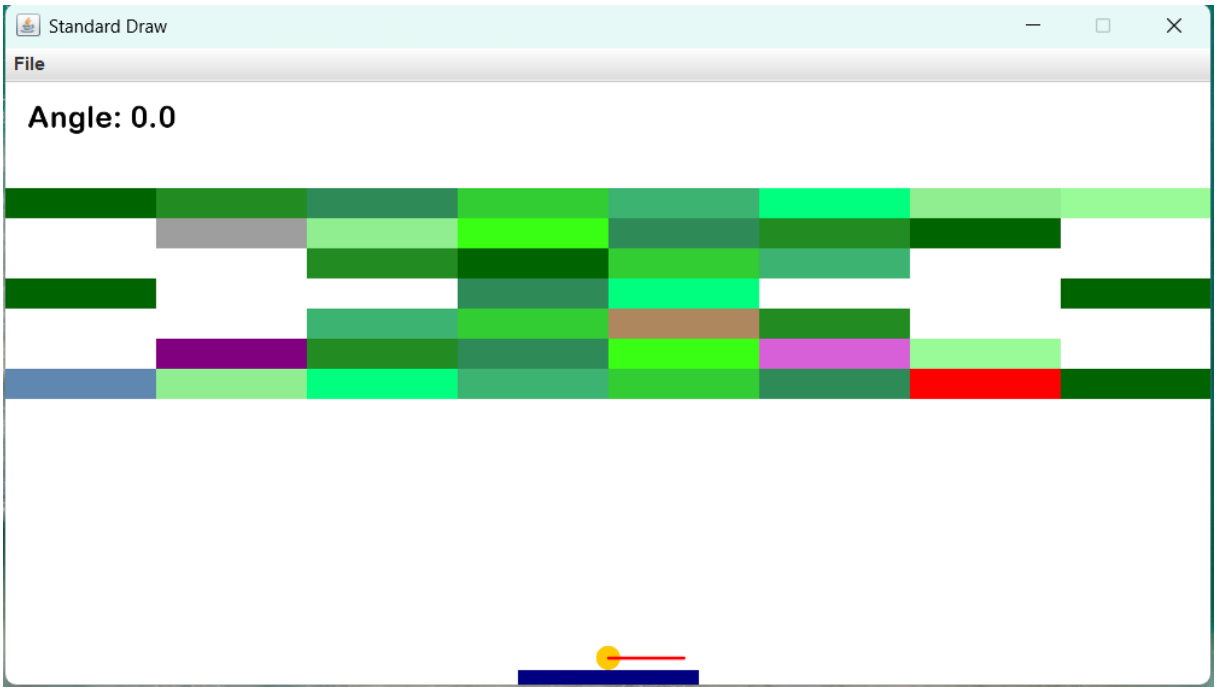
The original game features only one static map, which can become repetitive over time. To enhance variety, I designed four unique maps, each offering a new challenge. The game randomly selects a map at the start, ensuring a different scenario in each playthrough.

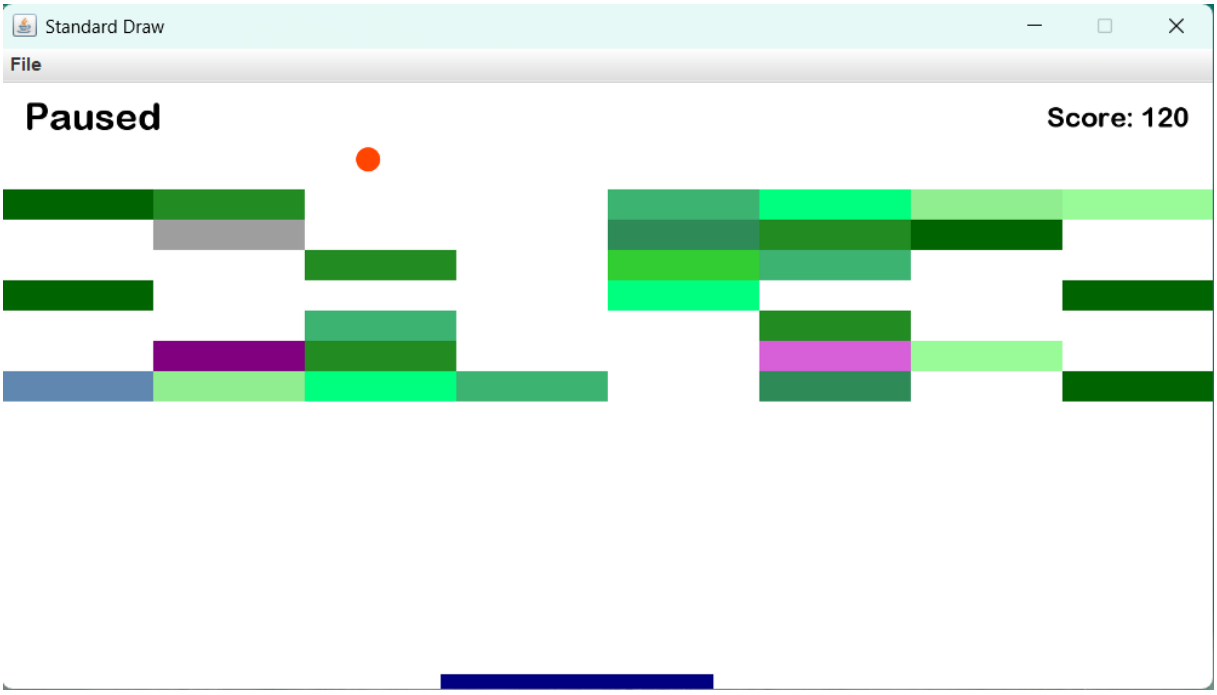
Power-Ups

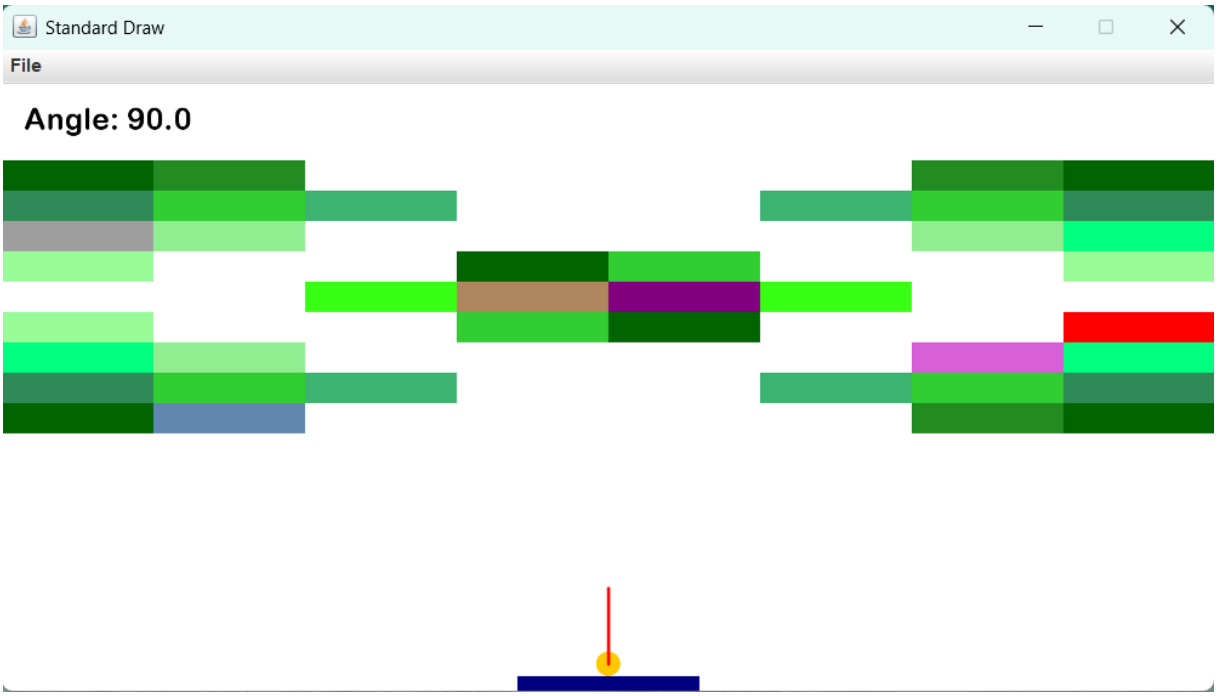
Perhaps the most exciting addition to my version is the introduction of six distinct power-ups, obtained by breaking specific bricks. These special bricks are chosen randomly and have unique colors to set them apart from standard green bricks.

- **Gray Brick (Decrease ball radius):** Reduces the ball's radius from 8 to 6. Activates only if the faster ball brick is not broken to prevent unexpected bugs.
- **Brown Brick (Increase paddle width):** Expands the paddle's half-width from 60 to 90, making it easier to catch the ball.
- **Pink Brick (Faster ball):** Increases the ball's velocity from 5 to 7.5. Activates only if the decrease ball radius brick is not broken to hinder unexpected behavior.
- **Light Blue Brick (Increase ball radius):** Expands the ball's radius from 8 to 12, making it easier to hit bricks.
- **Purple Brick (Decrease paddle width):** Shrinks the paddle's half-width from 60 to 40, making the game more challenging.
  Red Brick (Fire ball): The most thrilling power-up in the game! When activated:
- The ball turns a vivid shade of red.
- Instead of bouncing off bricks, it passes through them, allowing the player to destroy multiple bricks in a short time.

Screenshots


Standard Draw
File
Angle: 0.0


Standard Draw
File
Paused
Score: 120


Standard Draw
File
Angle: 90.0