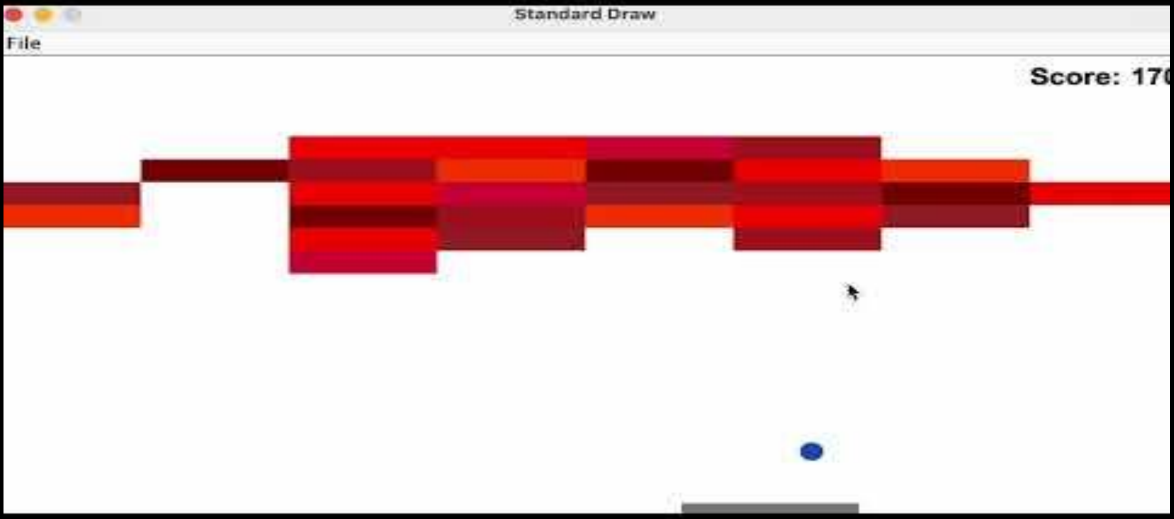
https://youtu.be/ZuluXIBrXo0 **(DX-BALL)**



https://youtu.be/z0Dkg8J5Slc ***(modified version)***

1. Introduction

The Brick Breaker game is a classic arcade game where the player controls a paddle to bounce a ball, aiming to break all the bricks on the screen. The game ends when the player either destroys all the bricks or fails to keep the ball in play, allowing it to fall below the paddle. This project implements a simplified version of the Brick Breaker game using Java and the **StdDraw** library. StdDraw allows for easy rendering of shapes, text, and colors, making it ideal for 2D game development.

The game features a paddle, a ball, and a grid of colorful bricks. The player can move the paddle left and right using the arrow keys and adjust the initial angle of the ball's trajectory. The ball bounces off the paddle, walls, and bricks, with collisions dynamically altering its direction. The game includes a scoring system, a pause feature, and win/lose conditions.

2. Game Mechanism

The core mechanics of the game are centered around the movement of the ball, collision detection, and user interaction. Below is a detailed explanation of the game's mechanics:

2.1 Ball Movement

- The ball starts at a fixed position above the paddle. Its initial velocity is determined by the angle (rad) set by the player before launching the ball.
- The ball's velocity is calculated using trigonometric functions:

$$\text{ballVelocityY}=\text{ballVelocity}\times\sin(\text{rad})$$
$$\text{ballVelocityX}=\text{ballVelocity}\times\cos(\text{rad})$$

Here, ballVelocity is the magnitude of the ball's speed, and rad is the angle in radians. We are decomposing the vector into its components physically.

- The ball's position is updated in each frame by adding the velocity components to its current position:

$$\text{initialBallPos}[0]\mathrel{+}=\text{ballVelocityX}$$
$$\text{initialBallPos}[1]\mathrel{+}=\text{ballVelocityY}$$

2.2 Collision Handling

Collisions are a critical part of the game. The ball interacts with the walls, paddle, and bricks, and each collision type is handled differently:

1. **Wall Collisions**:
    a. When the ball hits the left or right wall, its horizontal velocity (ballVelocityX) is inverted.

-->*(initialBallPos[0] + (ballVelocityX) - ballRadius < 0) || (initialBallPos[0] + (ballVelocityX) + ballRadius > xScale)* is my condition to check in code.

--> *initialBallPos[0] - ballRadius* means that left side of the circle. *+ ballVelocity* means next movement of the ball. Then, my code checks if the left side of the ball's next position is within my canvas. It also checks if the right side of the circle is larger than the size of the canvas.

    b. When the ball hits the top wall, its vertical velocity (ballVelocityY) is inverted.

--> *(initialBallPos[1] + ballVelocityY + ballRadius > yScale)* is my condition to check in Y-axis. It has only top wall condition. Because bottom wall means that the game over.
--> Same algorithm with X-axis

    c. If the ball falls below the paddle, the game ends.

--> *(initialBallPos[1] + (ballVelocityY) - ballRadius < 0)* checks similarly.

--> If this statement true, it means that game over. Then, some design texts and visuals are printed, and the game is finished, meaning the loop is broken.

2. **Paddle Collisions**:
    a. When the ball hits the paddle, its vertical velocity is inverted, causing it to bounce upward.

--> *(initialBallPos[0] + ballVelocityX < paddlePos[0] + paddleHalfwidth && paddlePos[0] - paddleHalfwidth < initialBallPos[0] + ballVelocityX && initialBallPos[1] + ballVelocityY - ballRadius <= paddlePos[1] + paddleHalfheight)*

--> It checks if the ball's position is suitable for collision with the paddle. The ball's y-coordinate should be between the paddle's dimensions, and its bottom should be at the paddle's height.

--> *(ballVelocityX != 0 && ballVelocityY != 0)* To prevent the tab from appearing in the initial state of the ball.

b. The ball can also hit the corners of the paddle. Therefore, simply changing direction along the y-axis will not always be sufficient. Some corner collision rules need to be applied here as well. The corner collision mechanism will be explained in detail in the next section, and the exact same mechanism applies here.

```
--> else if (Math.pow(initialBallPos[0] - paddlePos[0] - paddleHalfwidth,2) + Math.pow(initialBallPos[1]  - paddlePos[1] - paddleHalfheight,2) <= ballRadius*ballRadius){
    double paddleNormalX = initialBallPos[0] - (paddlePos[0] + paddleHalfwidth);
    double paddleNormalY = initialBallPos[1] - (paddlePos[1] + paddleHalfheight);
    double normalLength = Math.sqrt(paddleNormalX * paddleNormalX + paddleNormalY * paddleNormalY);
    if (normalLength != 0) {
        paddleNormalX /= normalLength;
        paddleNormalY /= normalLength;
    }
    double dotProduct = ballVelocityX * paddleNormalX + ballVelocityY * paddleNormalY;
    ballVelocityX = ballVelocityX - 2 * dotProduct * paddleNormalX;
    ballVelocityY = ballVelocityY - 2 * dotProduct * paddleNormalY;
}
```

--> This part of the code handles only the right corner collision. The only difference from the left corner collision is the coordinates of the colliding corner. The full code contains both of them.

--> It starts with else condition because, for a corner collision to occur, the ball's center must be outside the paddle's boundaries. Then, we assume that the ball's x-coordinate is to the right of the paddle and check for a corner collision based on the right corner. This check is done by determining whether the distance between the ball's center and the paddle's corner is smaller than the ball's radius. If this condition is met, we apply the corner collision steps and reflect the ball accordingly.

3. **Brick Collisions**:
    a. A for loop checks all the bricks, and the one most suitable for collision based on the conditions is selected, after which operations are performed for that brick.
    b. Collisions can occur on the top, bottom, left, or right sides of a brick, as well as at its corners.

The collision detection is performed using the following steps:

Calculate Brick Boundaries:
For each brick, the boundaries (left, right, top, and bottom) are calculated based on its center coordinates and half-width/height:

```
--> double left = brickCoordinates[num][0] - brickHalfwidth;

double right = brickCoordinates[num][0] + brickHalfwidth;

double top = brickCoordinates[num][1] + brickHalfheight;

double bottom = brickCoordinates[num][1] - brickHalfheight;
```

Check for Side Collisions:
The game checks if the ball's next position (after applying its velocity) intersects with any of the brick's sides:

```
-->boolean topCollision = initialBallPos[1] + ballVelocityY + ballRadius > top &&
            initialBallPos[1] + ballVelocityY < top &&
            initialBallPos[0] + ballVelocityX > left &&
            initialBallPos[0] + ballVelocityX < right;
```

--> For example, for a top collision to occur, the y-value must be above the top of the block, and the x-value must be within the block's width.

Then, the speed is updated based on the type of collision. If it's a top or bottom collision, the direction of the y-velocity is reversed, while for a right or left collision, the direction of the x-velocity is reversed.

```
--> if (topCollision || bottomCollision) {
  ballVelocityY = -ballVelocityY;
} else  {
  ballVelocityX = -ballVelocityX;
}
```

Check for Corner Collisions:
Corner collisions occur when the ball hits one of the four corners of a brick. There is a method which is called isCornerCollision to detect them. This method calculates the distance between the ball's next position and each corner of the brick. If the distance is less than or equal to the ball's radius, a corner collision is detected. Then, it returns true or false.

```
-->public static boolean isCornerCollision(double cornerX, double cornerY, double[] ballPos, double ballVelocityX, double ballVelocityY, double ballRadius) {

  double nextBallX = ballPos[0] + ballVelocityX;

  double nextBallY = ballPos[1] + ballVelocityY;

  double distanceSquared = Math.pow(nextBallX - cornerX, 2) + Math.pow(nextBallY - cornerY, 2);

  return distanceSquared <= Math.pow(ballRadius, 2);

}
```

-->> distanceSquared≤ballRadius^2

Then, it checked for a corner collision by testing each corner separately. There are 'or' operators in between of conditions because if even one of them is true, it means a corner collision has occurred.

```
-->boolean cornerCollision = isCornerCollision(left, top, initialBallPos, ballVelocityX, ballVelocityY, ballRadius) ||

              isCornerCollision(right, top, initialBallPos, ballVelocityX, ballVelocityY, ballRadius) ||

              isCornerCollision(left, bottom, initialBallPos, ballVelocityX, ballVelocityY, ballRadius) ||

              isCornerCollision(right, bottom, initialBallPos, ballVelocityX, ballVelocityY, ballRadius);
```

If a corner collision is detected, the ball's velocity is reflected using the normal vector of the collision point. The normal vector is calculated as the vector from the corner to the ball's center.

Why Use the Dot Product?
The dot product is used for two main reasons:

1. **Calculate the Reflection Vector**:
   When the ball collides with a corner, we need to determine the new direction of the ball after the collision. The dot product helps us calculate the reflection of the ball's velocity vector relative to the normal vector of the collision surface.
2. **Determine the Angle of Reflection**:
   The dot product allows us to compute the angle between the ball's velocity vector and the normal vector. This angle is used to ensure that the ball bounces off the corner at the correct angle, following the law of reflection (angle of incidence = angle of reflection).

Step 1: Calculate the Normal Vector

The normal vector is a vector perpendicular to the collision surface. For corner collisions, the normal vector is calculated as the vector from the corner of the brick to the center of the ball. cornerX and cornerY are the coordinates of the nearest corner. How they are determined will be explained later.

```
-->double normalX = initialBallPos[0] - cornerX;

double normalY = initialBallPos[1] - cornerY;
```

Step 2: Normalize the Normal Vector
The normal vector must be normalized (scaled to a length of 1) to ensure accurate calculations. This ensures that the normal vector has a magnitude of 1, making it a unit vector.

```
--> double normalLength = Math.sqrt(normalX * normalX + normalY * normalY);

 if (normalLength != 0) {

 normalX /= normalLength;

normalY /= normalLength;

}
```

Step 3: Calculate the Dot Product
The dot product of the ball's velocity vector and the normal vector is calculated.

*--> double dotProduct = ballVelocityX * normalX + ballVelocityY * normalY;*

Step 4: Reflect the Ball's Velocity
Using the dot product, the ball's velocity is reflected relative to the normal vector.

*ballVelocityX = ballVelocityX - 2 * dotProduct * normalX;*

*ballVelocityY = ballVelocityY - 2 * dotProduct * normalY;*

--> V' = v – 2(V.N).N
V: incoming velocity vector
N: surface normal vector
V': reflected velocity vector

This formula works as follows:

1. Subtract twice the projection of the velocity vector onto the normal vector from the original velocity vector.
2. This effectively "flips" the component of the velocity vector that is aligned with the normal vector, simulating a bounce.

Why This Works?
The reflection formula is derived from the law of reflection in physics, which states that the angle of incidence equals the angle of reflection. By using the dot product, we can mathematically compute the reflection of the ball's velocity vector without needing to calculate angles explicitly.

- **Dot Product**: Measures how much of the ball's velocity is aligned with the normal vector.
- **Reflection Formula**: Adjusts the ball's velocity to simulate a realistic bounce off the collision surface.

How cornerX and cornerY are determined?

When a corner collision is detected, we need to determine which corner the ball is colliding with. This is done by calculating the distance between the ball's center and each of the brick's four corners. The corner with the smallest distance is the one involved in the collision.

First, the x and y coordinates of each corner were defined, then the distance to the ball's current position was calculated using the Pythagorean theorem.

-->double topLeftDist = Math.pow(initialBallPos[0] - topLeftX, 2) + Math.pow(initialBallPos[1] - topLeftY, 2);

```
double topRightDist = Math.pow(initialBallPos[0] - topRightX, 2) + Math.pow(initialBallPos[1] - topRightY, 2);

double bottomLeftDist = Math.pow(initialBallPos[0] - bottomLeftX, 2) + Math.pow(initialBallPos[1] - bottomLeftY, 2);

double bottomRightDist = Math.pow(initialBallPos[0] - bottomRightX, 2) + Math.pow(initialBallPos[1] - bottomRightY, 2);
```

--> distanceSquared=(BallX−cornerX)^2+(BallY−cornerY)^2

Then, the shortest distance is found using the chained min functions, and **cornerX** and **cornerY** are defined.

```
--> double minDist = Math.min(Math.min(topLeftDist, topRightDist), Math.min(bottomLeftDist, bottomRightDist));
```

After Collision?

-After a collision, the coordinates of the collided block cannot be deleted because an array is used (not arrayList). Instead, a coordinate like {-100, -100} (which is outside the canvas) is assigned to ensure that the brick is not shown among the bricks in the next while loop iteration. The brick's previous position is also painted white to create the appearance of empty space.

```
--> StdDraw.setPenColor(0, 0, 0);
StdDraw.filledRectangle(brickCoordinates[num][0], brickCoordinates[num][1], brickHalfwidth, brickHalfheight);
brickCoordinates[num] = new double[]{-100, -100};
```

-Thanks to the following lines inside the if statements, the ball's position is updated in each while loop iteration.

```
--> initialBallPos[0] += ballVelocityX;
initialBallPos[1] += ballVelocityY;
```

2.3 User Interaction

- The player can move the paddle left and right using the arrow keys (KeyEvent.VK_LEFT and KeyEvent.VK_RIGHT).

```
-->if (StdDraw.isKeyPressed(KeyEvent.VK_LEFT)) {
   paddlePos[0] -= paddleSpeed;
   if (paddlePos[0] - paddleHalfwidth <= 0) {
      paddlePos[0] = 0 + paddleHalfwidth;
   }
   StdDraw.setPenColor(paddleColor);
```

```
    StdDraw.filledRectangle(paddlePos[0], paddlePos[1], paddleHalfwidth, paddleHalfheight);
}
```

--> We update the paddle position based on the paddle velocity. We do this by subtracting the velocity from the paddle position each time the left arrow key is pressed. The purpose of the if statement is to prevent the paddle from going outside the canvas area

--> Update position:

paddlePos[0]−=paddleSpeed(for left movement)

paddlePos[0]+=paddleSpeed(for right movement)

- The player can adjust the initial angle of the ball's trajectory before launching it by pressing the left and right arrow keys.

```
--> double xLineEnd = initialBallPos[0] + 100 * Math.cos(rad);
double yLineEnd = initialBallPos[1] + 100 * Math.sin(rad);
```

--> When drawing a line, xLineEnd and yLineEnd are defining the endpoint. As the user presses the left and right arrow keys, the rad (in radians) changes, and the line's endpoint moves left and right with trigonometric functions. 100 is the length of the line. This is the movement algorithm:

```
--> if (StdDraw.isKeyPressed(KeyEvent.VK_LEFT)) {
    if (rad <= Math.PI - 0.017) {  //set left limit
        rad = rad + 0.017;
    } else {
        rad = Math.PI;
    }
}
```

--> The if statement prevents it from being more than 0 or 180 degrees. 0.017 is also a randomly small number.

-->Then, the initial speed of the ball is determined by the following code block. If the ball has no speed, it is given velocity values in the direction of the selected line.

```
--> if ((ballVelocityX == 0) && (ballVelocityY == 0)) {
    ballVelocityX = ballVelocity * Math.cos(rad);
    ballVelocityY = ballVelocity * Math.sin(rad);
}
```

- The game can be paused and resumed by pressing the spacebar (KeyEvent.VK_SPACE).

```
--> if (StdDraw.isKeyPressed(32)) {
    if (!spacePressed) {
        toggle = !toggle;
        spacePressed = true;
    }
} else {
    spacePressed = false;
}

// If the game is paused, display "PAUSED" and skip the rest of the loop
if (toggle) {
    StdDraw.setPenColor(StdDraw.BLACK);
    StdDraw.setFont(new java.awt.Font("Montserrat", java.awt.Font.BOLD, 24));
    StdDraw.textLeft(100, 350, "PAUSED");
    StdDraw.show();
    continue;
}
```

--> When paused, the while loop continues running. When it enters the pause block, the continue statement prevents it from moving to the next line, The same part at the beginning of the loop keeps repeating.The game looks like frozen. When pressed again, this situation is resolved, and the entire while loop starts running again.

3. Game Loop

The game consists of two main loops:

1. **Initialization Loop**:
   a. Waits for the player to press the spacebar to start the game.
   b. Displays the bricks, paddle, and ball.
   c. Allows the player to adjust the ball's initial angle.
2. **Main Game Loop**:
   a. Handles ball movement, collisions, and user input.
   b. Updates the game state and renders the frame.

```
-->StdDraw.show();
StdDraw.pause(10);
StdDraw.clear()
```

--> Lines like these ensure that the adjustments within the while loop are displayed and waited for, creating a smooth visual effect.

4. Winning and Losing Mechanism

In the Brick Breaker game, the player wins or loses based on specific conditions. These conditions define the core objectives and boundaries of the game. Below is a detailed explanation of the winning and losing mechanisms:

Score Mechanism

- **Brick Destruction**: Each time the player destroys a brick by hitting it with the ball, the score increases by **10 points**.
- **Score Update**: The score is updated immediately after a brick is destroyed.

```
--> score += 10;
```

- The current score is displayed in the top-right corner of the screen during the game.
- The score is updated in real-time as bricks are destroyed.

```
--> StdDraw.textLeft(700, 380, "Score: " + score)
```

Winning Condition

The player wins the game by breaking all the bricks on the screen. This condition is checked by comparing the player's score to the total possible score. Each brick destroyed adds 10 points to the score. When all bricks are destroyed, the score equals the number of bricks multiplied by 10.

```
--> if (score == brickCoordinates.length * 10) {
    StdDraw.setFont(new java.awt.Font("Montserrat", java.awt.Font.BOLD, 28));
    StdDraw.textLeft(350, 220, "VICTORY");
    StdDraw.show();
    break;
}
```

Losing Condition

The player loses the game if they fail to keep the ball in play, allowing it to fall below the paddle. This condition is checked by monitoring the ball's y-coordinate. If the ball's bottom edge goes below the screen's lower boundary, the game is over. Game Over” is displayed on the screen, and the current score is shown below it.

```
--> if (initialBallPos[1] + (ballVelocityY) - ballRadius < 0) {
    StdDraw.setFont(new java.awt.Font("Montserrat", java.awt.Font.BOLD, 24));
    StdDraw.textLeft(350, 150, "GAME OVER");
    StdDraw.setPenColor(StdDraw.BLACK);
    StdDraw.setFont(new java.awt.Font("Arial", java.awt.Font.BOLD, 20));
    StdDraw.textLeft(360, 130, "Score: " + score);
    StdDraw.filledCircle(initialBallPos[0], initialBallPos[1], ballRadius);
    StdDraw.show();
    break;
}
```

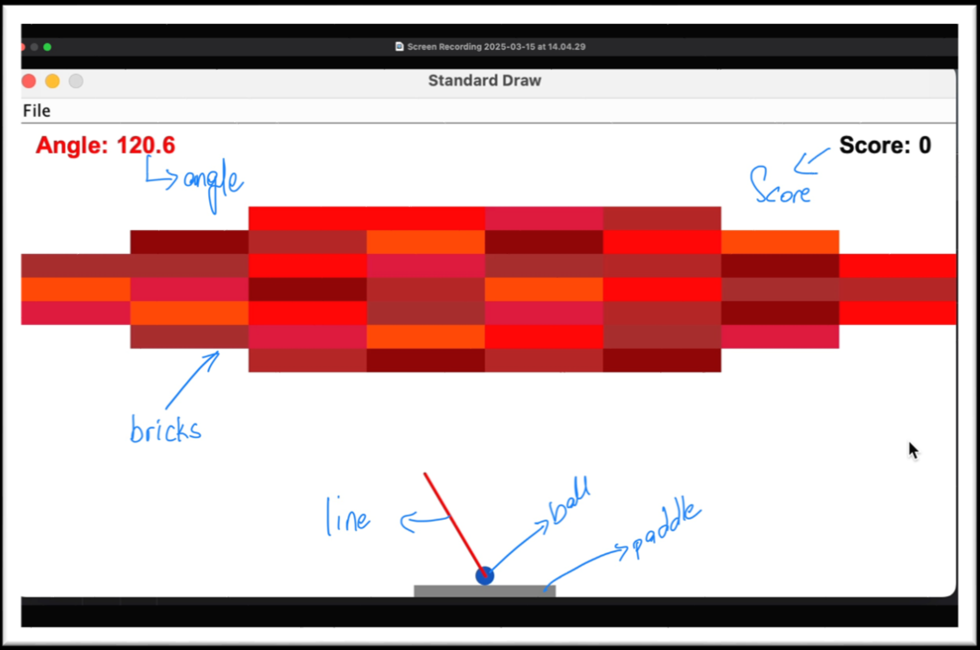
Game Termination
In both winning and losing scenarios, the game loop (while (true)) is terminated, and the game screen freezes. The player must restart the program to play again.

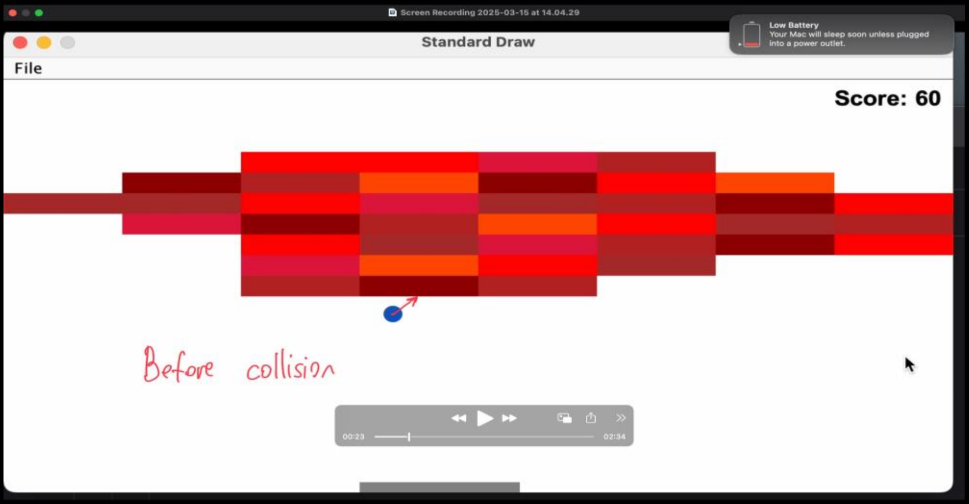
5. Conclusion

This project successfully implements a simplified version of the Brick Breaker game using Java and the StdDraw library. The game features core mechanics such as ball movement, collision detection, and user interaction, along with a scoring system and win/lose conditions. The use of geometric calculations for collision detection and the StdDraw library for rendering ensures a smooth and enjoyable gaming experience. Future improvements could include adding levels, power-ups, and more complex brick patterns.

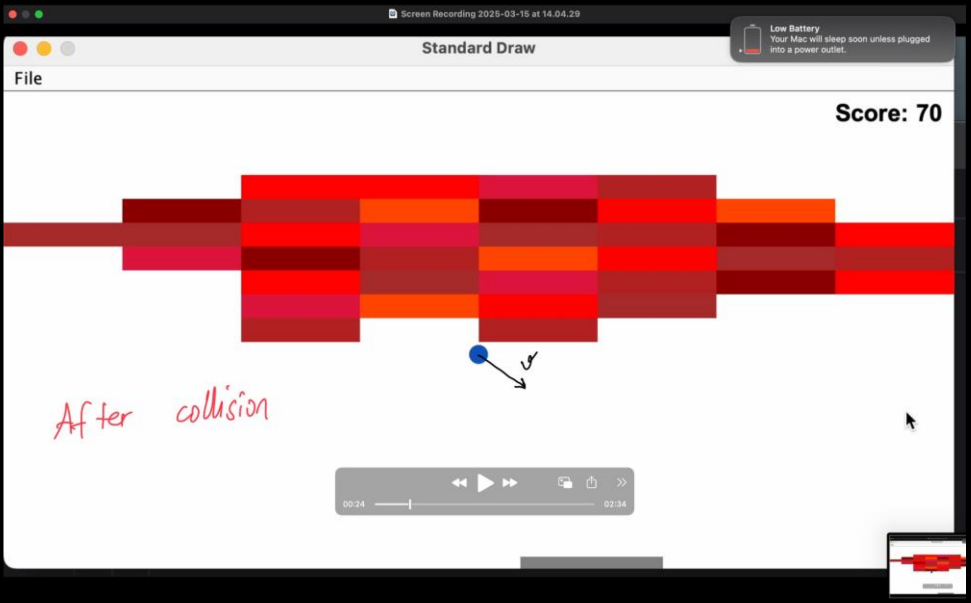
Screenshoots from Game

These screenshots belong to the standard DX-Ball game. The description and images of the modified version (rafa silva) will be provided below.

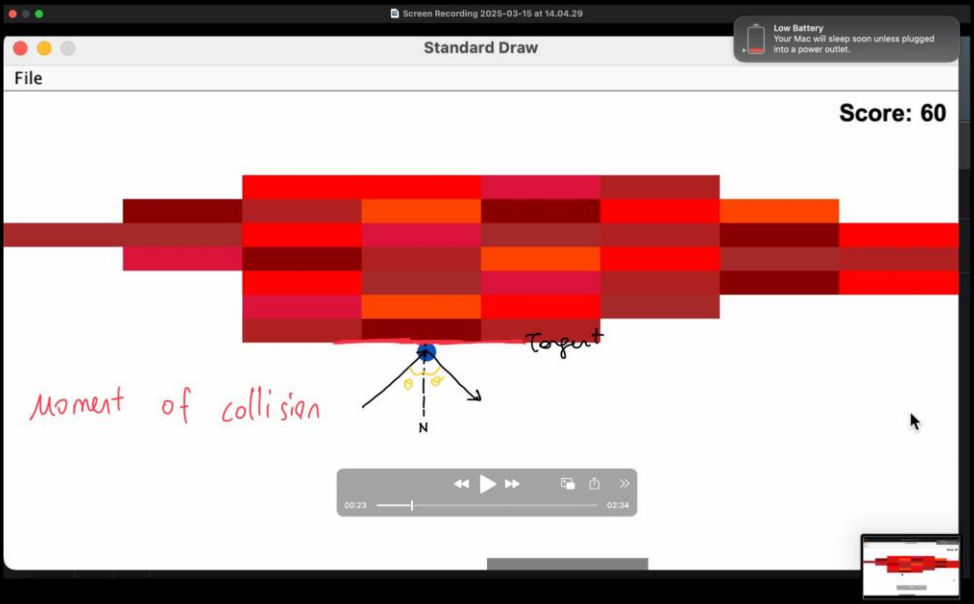




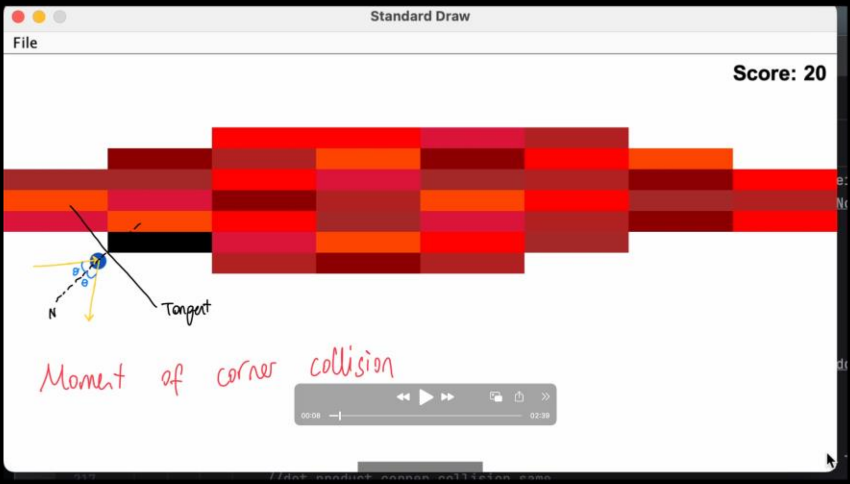
Low Battery
Your Mac will sleep soon unless plugged into a power outlet.

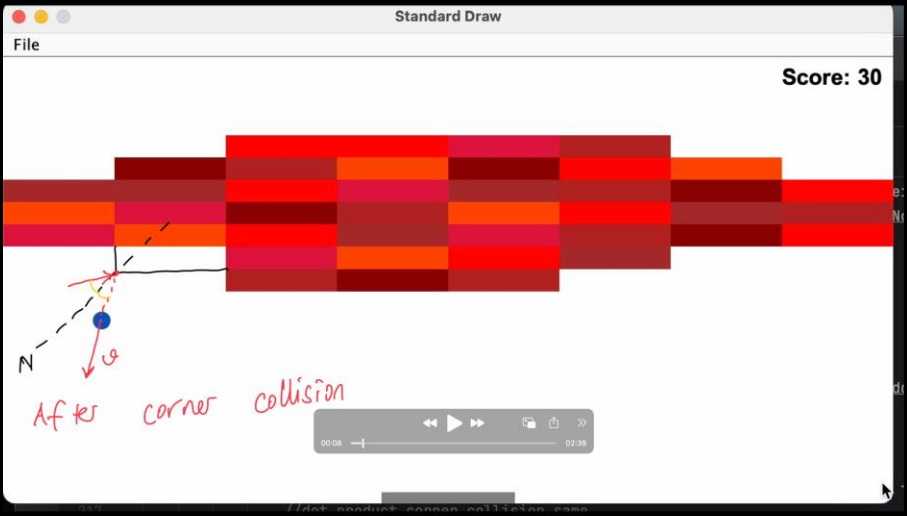


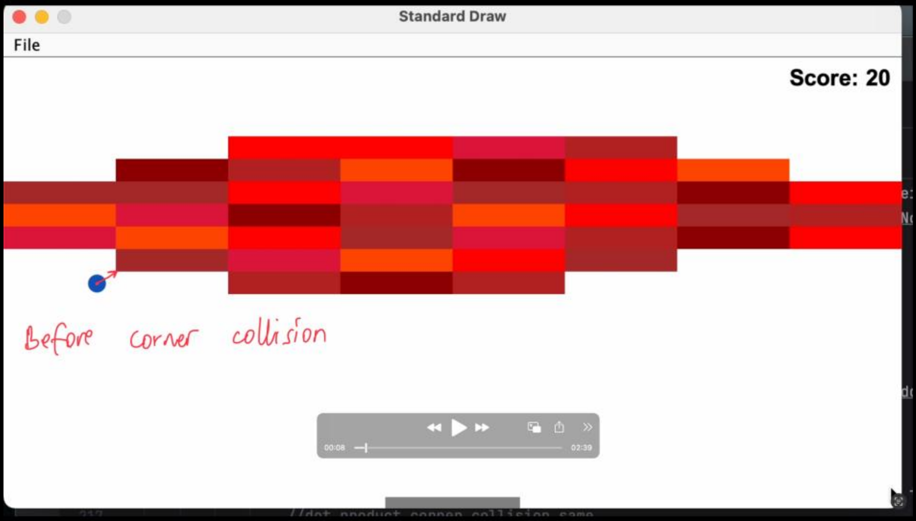
Low Battery
Your Mac will sleep soon unless plugged into a power outlet.

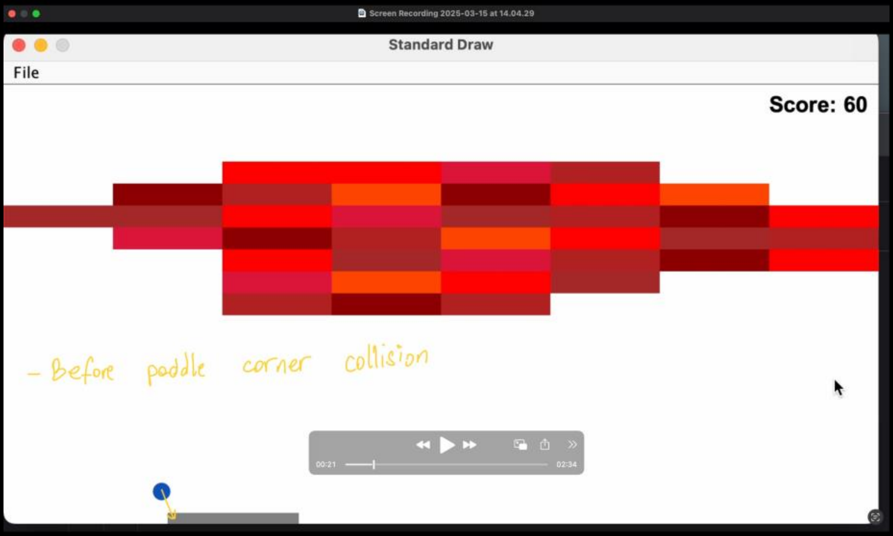


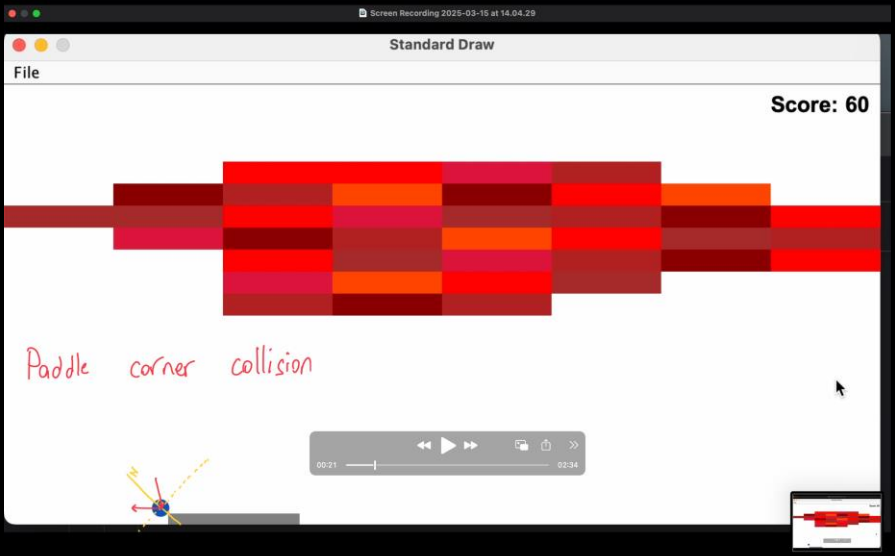
Low Battery
Your Mac will sleep soon unless plugged into a power outlet.

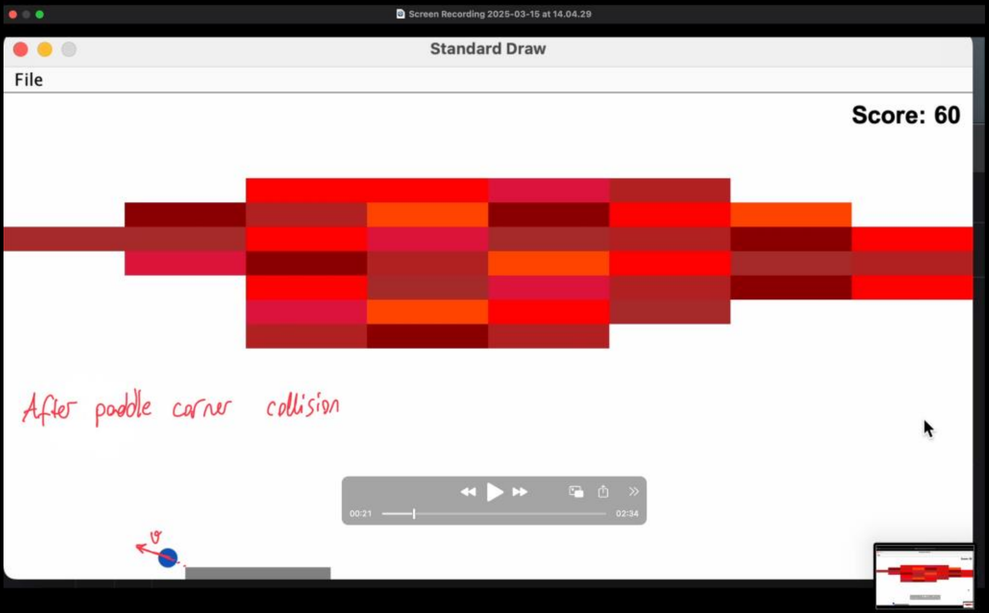

Standard Draw
File
Score: 20
θ
θ
θ
N
Tangent
Moment of corner collision
00:08
02:39

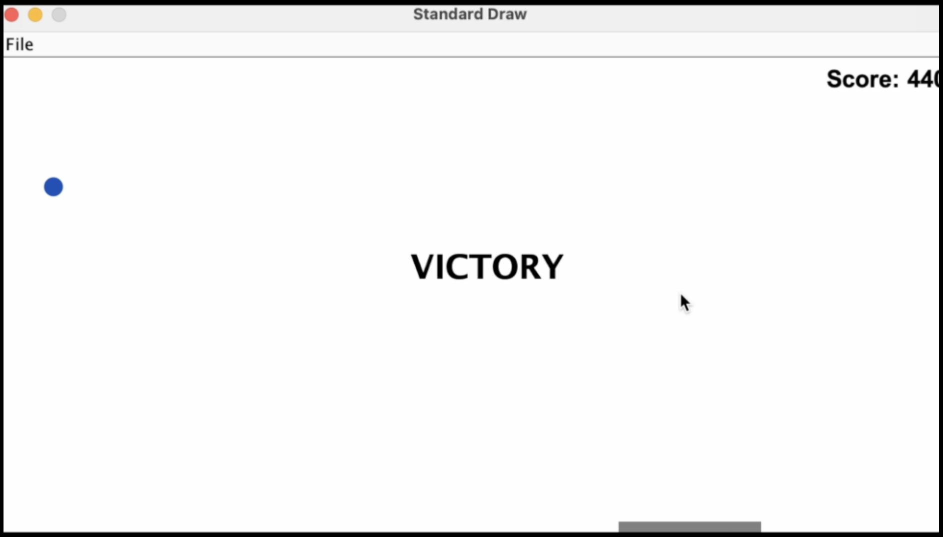

Standard Draw
File
Score: 30
N
v
After corner collision
00:08
02:39

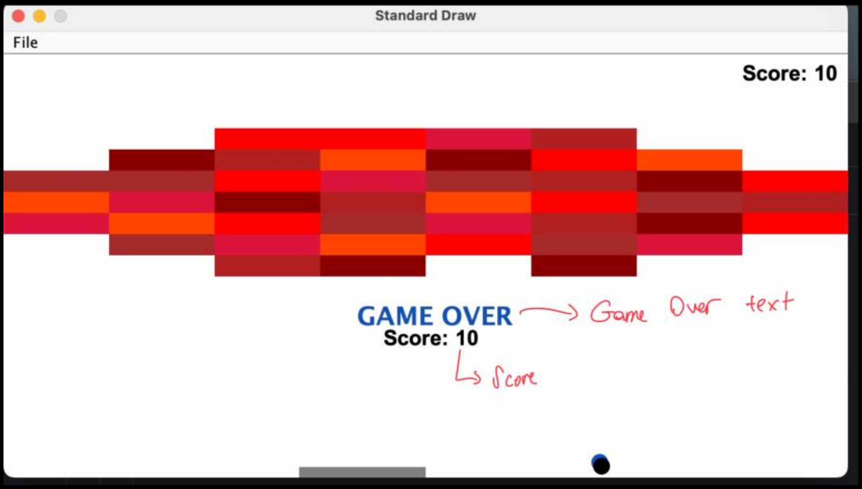

Standard Draw
File
Score: 20
Before corner collision
00:08
02:39

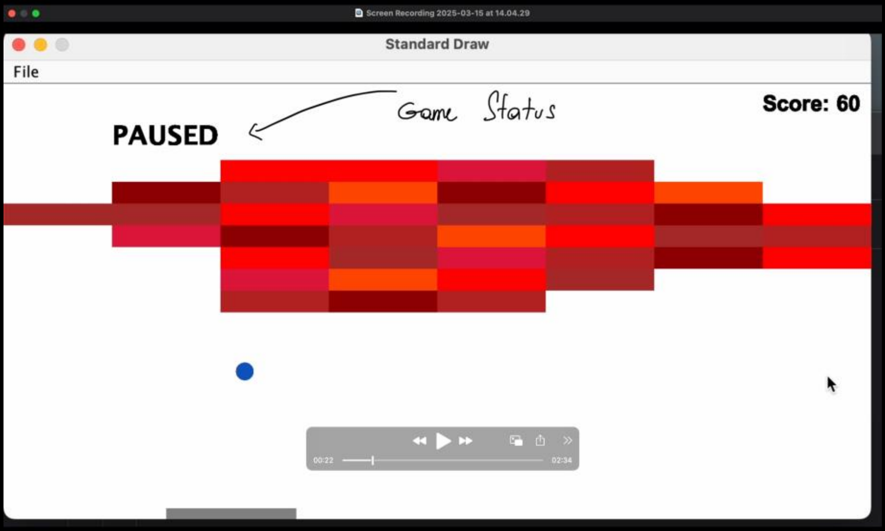

Screen Recording 2025-03-15 at 14.04.29
Standard Draw
File
Score: 60
- Before paddle corner collision
00:21
02:34


Screen Recording 2025-03-15 at 14.04.29
Standard Draw
File
Score: 60
Paddle corner collision
N
00:21
02:34


Screen Recording 2025-03-15 at 14.04.29
Standard Draw
File
Score: 60
After paddle corner collision
v
00:21
02:34



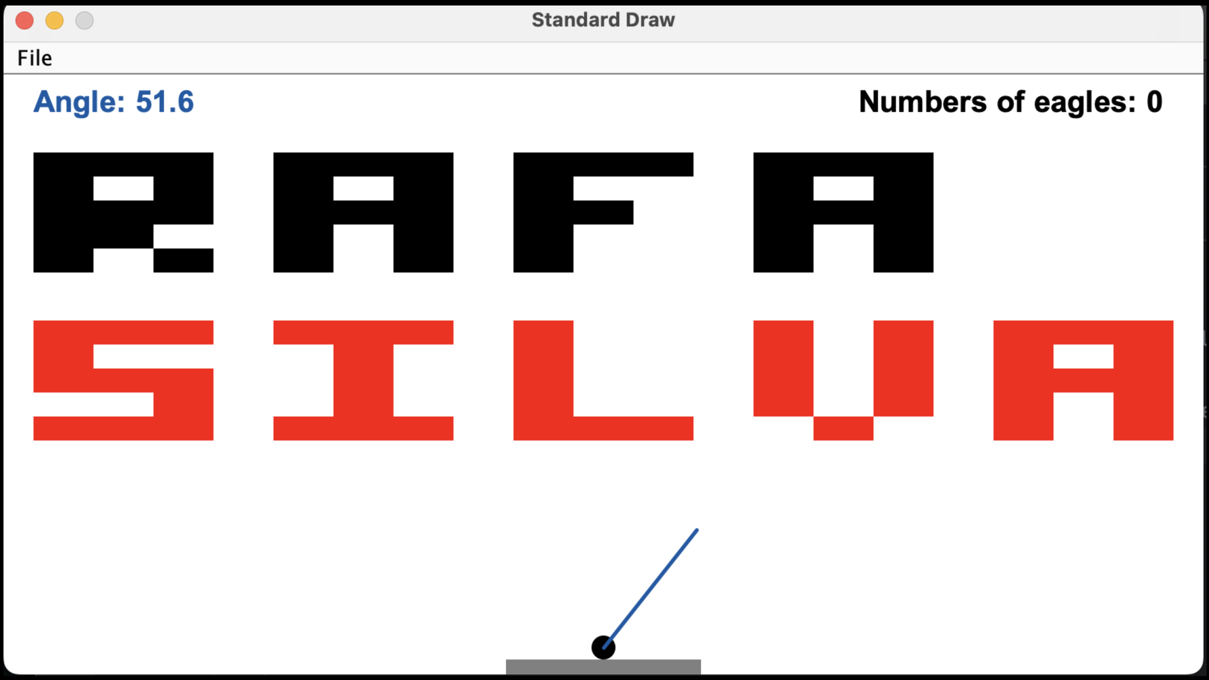




MODIFIED GAME (RAFA SILVA)

I designed a game based on Beşiktaş and my favorite player, Rafa Silva. The mechanics of the game are generally the same as DX-Ball. However, it includes some design changes, such as the placement and color of the bricks, the ball color, the score text, and the victory text. Additionally, there are a few small differences in the gameplay. Let's go over them:

The game's interface photo is attached below. Design changes such as brick positions, brick colors, ball color, and the score text have been made solely by modifying the values of variables.



There are two major changes in terms of gameplay:

1.Increasing Ball Velocity Over Time

In each while loop iteration, the ball's velocity vectors are multiplied by a small constant factor, gradually increasing its speed. This way, the ball accelerates proportionally over time.

```
--> if (ballVelocityX != 0 && ballVelocityY != 0){
    ballVelocityX = ballVelocityX * 1.0001;
    ballVelocityY = ballVelocityY * 1.0001;
}
```

--> The reason for checking that velocity vectors are not zero is to ensure that the acceleration works properly at the start of the game.

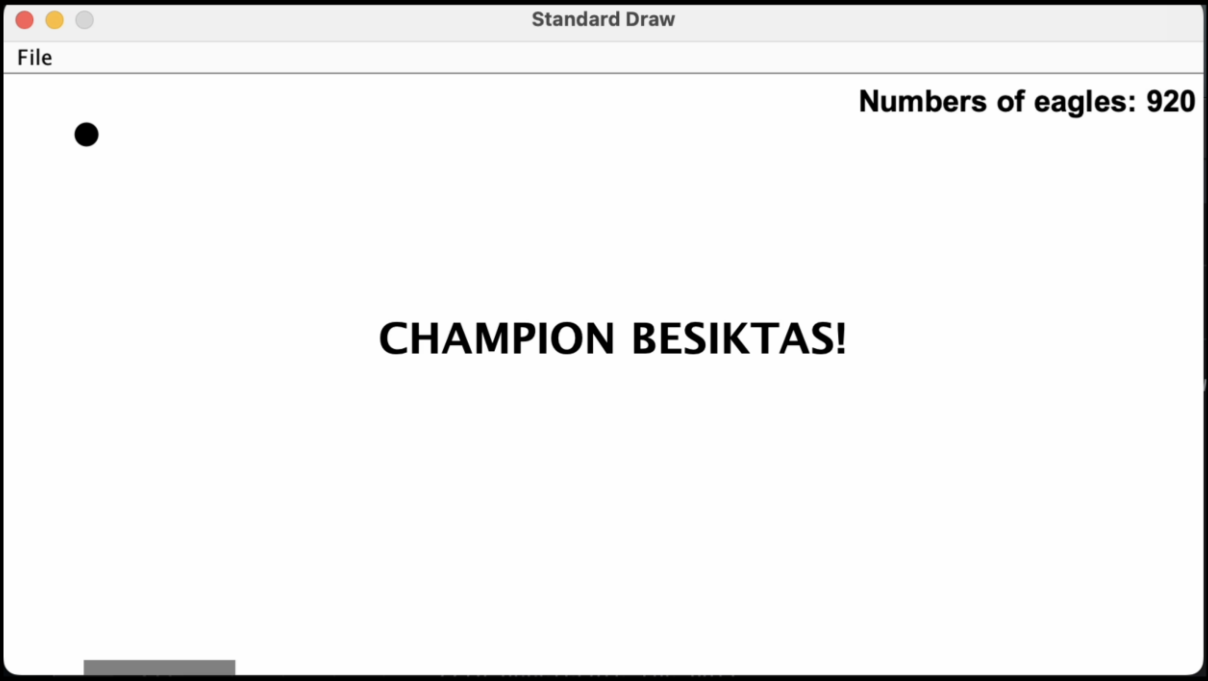
2.Decreasing Paddle Size Over Time

It follows the same logic as increasing the ball velocity. Here, the paddle size is multiplied by a certain factor over time, gradually decreasing its size at a fixed rate. These changes make the game progressively more challenging towards the end.

--> paddleHalfwidth -= 0.0015;

Note: I made these two changes (velocity and paddle size) after the pause block of the while loop to prevent them from continuously occurring while the game is paused.

--> Victory Scene



Numbers of eagles: 920

CHAMPION BESIKTAS!