

<https://www.youtube.com/watch?v=u-QDGQ74oVc>

<https://www.youtube.com/watch?v=4G9pDVt2YVc>

Introduction:

This is a DX-Ball-like game made by using StdDraw library in Java. In the game, the player first determines the direction of the ball by using the arrow keys on the keyboard.

Then the player controls the paddle at the bottom of the screen to prevent the ball from hitting the bottom boundary of the window (which is the game-over condition), while the ball destroys the bricks it touches and bounces back. The player gains 10 scores for each brick the ball destroys and wins the game if all bricks on the screen are destroyed.

Game Mechanism:

Initially, the bricks, the paddle, and the ball are standing on the screen. The red line indicates the trajectory of the ball and takes direction according to the keys pressed by the player. Shooting angle text on the top left displays the angle of ball's trajectory. The player can start the game by pressing space on this screen.

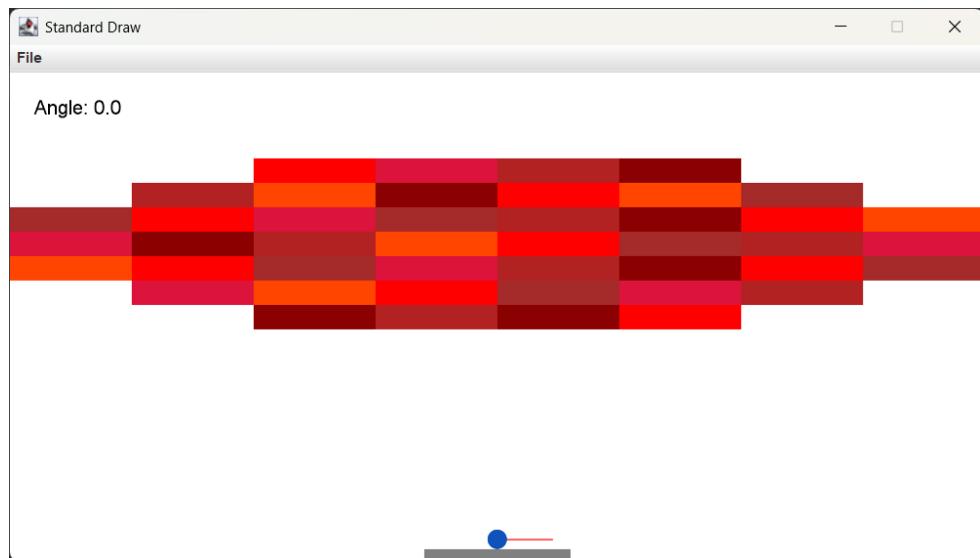


Figure 1: Initial Game

The game starts after the player presses the space key. After the game has started, the player can pause and continue the game by pressing the space key. The ball starts moving at the determined angle when the game starts. There is a text on the top left that displays the current score. If the game is paused, there will be a game paused text on the top left.

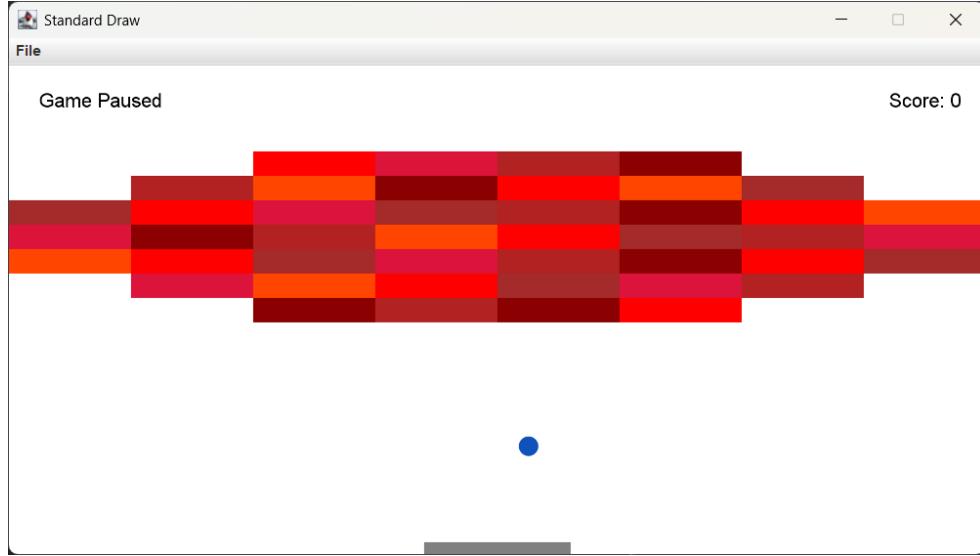


Figure 2: Paused Game

a) Ball Movement

In the code, I make the ball move with the following code:

```
ballPos[0] += ballVelocity * Math.cos(radang);  
ballPos[1] += ballVelocity * Math.sin(radang);
```

Where `ballPos[0]` is the x component of the ball's position, and `ballPos[1]` is the y component. The variable `radang` is the radian version of the variable `angle`, which keeps track of the ball's direction similar to the angle in a unit circle as in Figure 3. The variable `angle` will be changing throughout the game to keep track of direction.

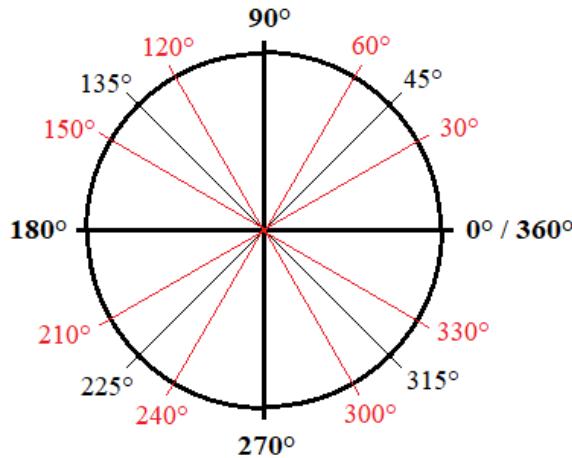


Figure 3: Circle with angles

b) Paddle Movement

```

if (StdDraw.isKeyPressed(KeyEvent.VK_RIGHT) && paddlePos[0] < 800 - paddleHalfwidth) {
    paddlePos[0] += paddleSpeed;
}
if (StdDraw.isKeyPressed(KeyEvent.VK_LEFT) && paddlePos[0] > paddleHalfwidth) {
    paddlePos[0] -= paddleSpeed;
}

```

The paddle moves right when the right arrow key is pressed, and left when the left arrow key is pressed. It moves by `paddleSpeed` pixels in each frame. The paddle doesn't have a y component of velocity. The paddle can't move further than any edge of the screen, this is checked by the `paddlePos` comparisons in the if statements.

c) Surface Collision

```

if (ballPos[0] > xScale - ballRadius || ballPos[0] < ballRadius)
{
    angle = 180 - angle;
}
if (ballPos[1] > yScale - ballRadius) {
    angle = -angle;
}

```

When colliding with the borders of the window, I simply changed the angle. If the ball collides with a horizontal line, the angle becomes the negative version of itself, which doesn't change the x component of velocity and reverses the y component. It similarly sets the new `angle` to `180-angle` when the ball collides with a horizontal line.

```
if (edgeDistanceY < edgeDistanceX) {  
    angle = -angle;  
    ballPos[0] += (distanceY - ballRadius) * Math.cos(radang);  
    ballPos[1] += (distanceY - ballRadius) * Math.sin(radang);  
    radang = Math.toRadians(angle);  
    collisionFix[0] = -(distanceY - ballRadius) *  
Math.cos(radang);  
    collisionFix[1] = -(distanceY - ballRadius) *  
Math.sin(radang);  
}  
else if (edgeDistanceX < edgeDistanceY) {  
    angle = 180 - angle;  
    ballPos[0] += (distanceX - ballRadius) * Math.cos(radang);  
    ballPos[1] += (distanceX - ballRadius) * Math.sin(radang);  
    radang = Math.toRadians(angle);  
    collisionFix[0] = -(distanceX - ballRadius) *  
Math.cos(radang);  
    collisionFix[1] = -(distanceX - ballRadius) *  
Math.sin(radang);
```

The variables `edgeDistanceY` and `edgeDistanceX` are the closest distance in X and Y axis between the ball and the brick (or paddle). If the ball is closer to a horizontal edge of the brick, the collision happens as if the ball hit a horizontal line, and a vertical line if it's closer to a vertical edge.

Also, if the ball collides with a brick or the paddle it steps back by the amount `distancex/y`(which is the perpendicular distance to the closer edge). This makes the

ball just touch the edge of the brick, as in Figure 4.

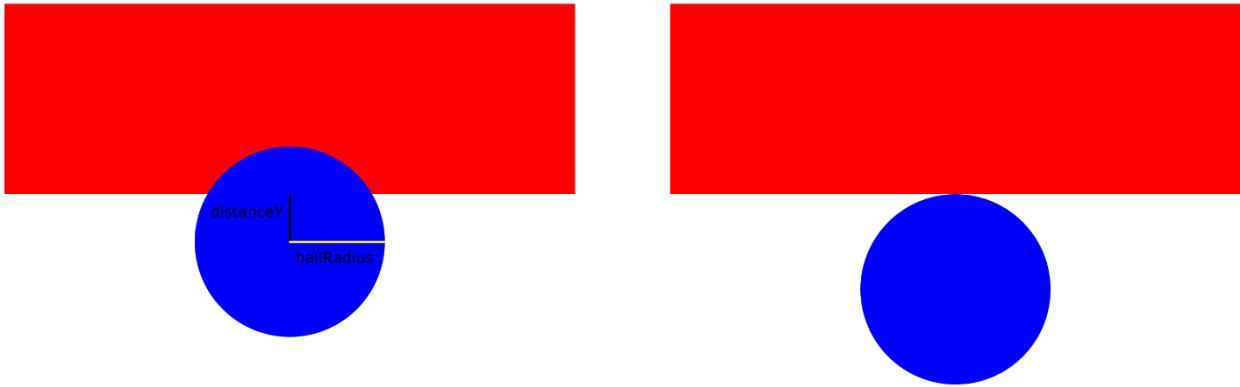


Figure 4:Ball stepback

After taking the ball back to the edge of the brick, the amount by which the ball is taken back is stored in the array **collisionFix**

```
ballPos[0] += collisionFix[0];
ballPos[1] += collisionFix[1];
collisionFix[0]=0;
collisionFix[1]=0;
```

After all possible collisions are calculated, the ball moves in a positive direction by **collisionFix** units. This makes the ball move by **ballVelocity** units every frame, making the animation more consistent.

d)Corner Collision

```
else {
    ballPos[0] += (distance - ballRadius) * Math.cos(radang);
    ballPos[1] += (distance - ballRadius) * Math.sin(radang);
    double[][] brickCorners = {
        {brick_coordinates[i][0] + brickHalfwidth,
        brick_coordinates[i][1] + brickHalfheight},
        {brick_coordinates[i][0] + brickHalfwidth,
        brick_coordinates[i][1] - brickHalfheight},
        {brick_coordinates[i][0] - brickHalfwidth,
        brick_coordinates[i][1] + brickHalfheight},
        {brick_coordinates[i][0] - brickHalfwidth,
        brick_coordinates[i][1] - brickHalfheight}
    };
    double min = Double.MAX_VALUE;
    int index = 0;
```

```

        for (int j = 0; j < 4; j++) {
            dist = Math.sqrt(Math.pow(ballPos[0] -
brickCorners[j][0], 2) + Math.pow(ballPos[1] -
brickCorners[j][1], 2));
            if (dist < min) {
                min = dist;
                index = j;
            }
        }
        double[] corner = brickCorners[index];
        tangent = (ballPos[1] - corner[1]) / (ballPos[0] -
corner[0]);
        angle = 180 + 2 * Math.toDegrees(Math.atan(tangent)) - angle
radang = Math.toRadians(angle);
        collisionFix[0] = -(distance - ballRadius) *
Math.cos(radang);
        collisionFix[1] = -(distance - ballRadius) *
Math.sin(radang);
    }
}

```

If the ball's distance to the closest vertical and horizontal edge are equal, a corner collision happens. A corner collision is the same as a surface collision other than the change in angle. However, collisions have to happen according to the tangent line which is perpendicular to the normal line as in Figure 5.

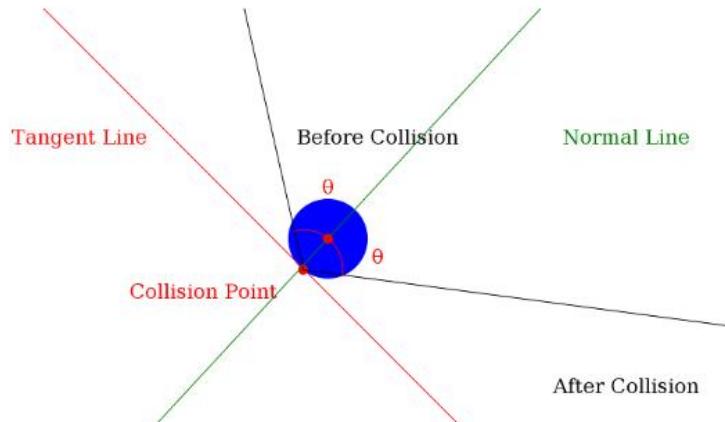


Figure 5:Corner collision

To find the direction of the tangent line, we first determine the tangent of the angle between the ball and the corner using the

```
tangent = (ballPos[1] - corner[1]) / (ballPos[0] - corner[0]);
```

equation. We can calculate the angle between the normal line and the x-axis by calculating the arctangent of this tangent value, I will call this angle α . Then we can tell

the angle between the x-axis and the tangent line, which is $\alpha+90$ or $\alpha-90$. Which one you choose doesn't matter since they are the same line.

The collision between the sloped tangent line and the ball is similar to reflection of light in Figure 6. In Figure 6 it can be seen that if the line is rotated positively by θ degrees, the reflection rotates by 2θ degrees.

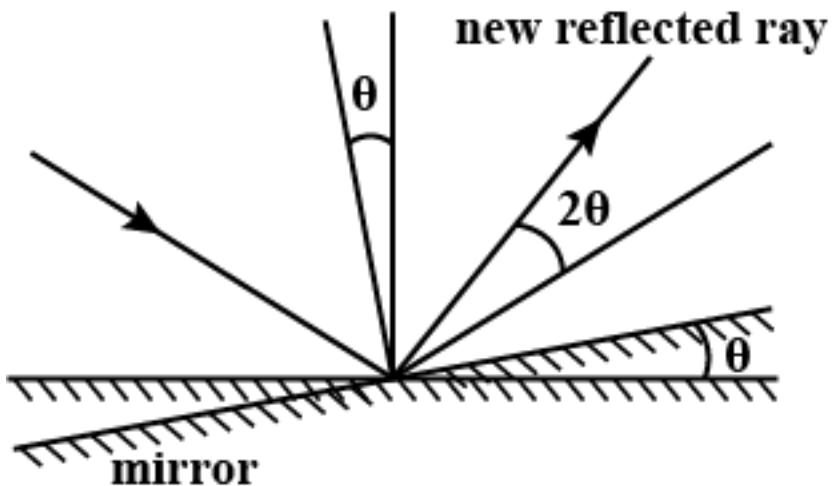


Figure 6: Reflection from an inclined surface

When the base angle is zero, the bounce angle is **-angle**. Therefore, the bounce angle will be **2θ-angle** when the line is rotated by θ degrees.

We can calculate the angle of the normal line by using the method **Math.atan** which return arctangent in radians. Then we have to use **Math.toDegrees** in order to convert the radian value into degree. These methods give us

Math.toDegrees(Math.atan(tangent)) which is equal to α (the angle between the normal line and x-axis). From there, we can tell the angle of the tangent line is **90+Math.toDegrees(Math.atan(tangent))**. We can use the **2θ-angle** formula that is explained before, to get the equation

```
angle = 180 + 2 * Math.toDegrees(Math.atan(tangent)) - angle;
```

which gives us the final value of angle after collision.

Modified Game:

In the modified version, I changed the positions, colors, and size of the bricks as seen in Figure 7.

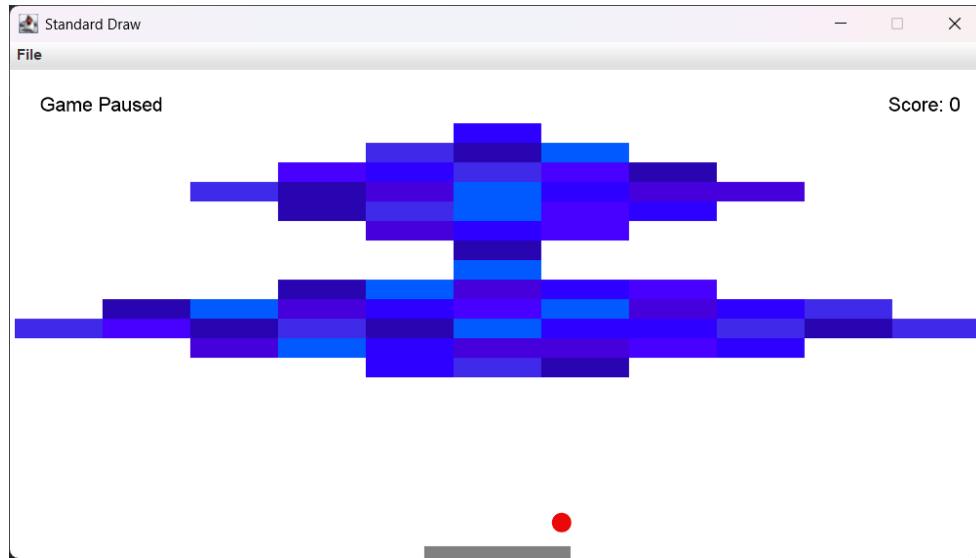


Figure 6:Modified Game

The ball also gets faster with each collision in this version, which is done by incrementing the variable `ballVelocity` by 0.1 after each collision.

I also added a restart mechanic which allows the player to restart the game by pressing R after winning or losing a game.

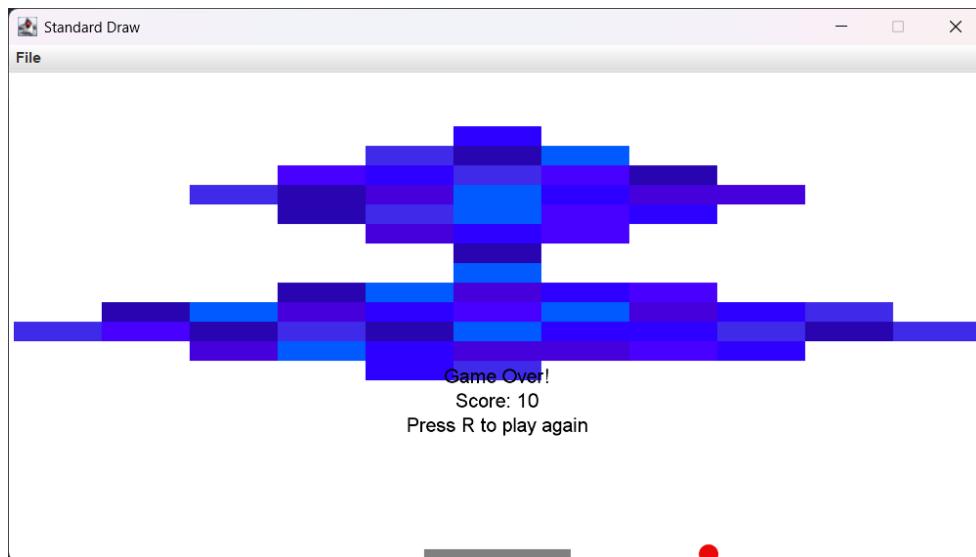


Figure 6:Restart Screen