CMPE160 ASSIGNMENT-1 REPORT

•Youtube Links:

→Original Game: https://www.youtube.com/watch?v=WJfd-04D0GA

→Modified Game: https://www.youtube.com/watch?v=URNc6RGjmZU

• **Introduction:** In this game, player tries to break all the bricks by colliding them properly. If he can hit the brick with the ball, he is awarded by 10 points. Player can adjust actively the position of the paddle by using the left and the right arrow keys. The game just starts if player presses " space" character and pauses if player presses "space" again. If the player is not able to hold the ball above the paddle ,which results with touching the ball with the bottom surface, the game is over. However, if player clear all the bricks, he wins the game and his final score is shown. The game takes place a while loop which is connected to whether game is started, paused or over. Inside the while loop, the positions of the ball are being examined to determine whether it bounces from the side walls, the paddle, the bricks or the top side. Bouncing situations are determined by looking at the distance relative to the radius of the ball. To accomplish all of these, stdDraw library is used.

•StdDraw.set(): To set the canvas size,xScale and yScale.

•StdDraw.setPenColor(): To arrange the color

•StdDraw.isKeyPressed(): To find which key is pressed

•StdDraw.pause(): To adjust speed of the game.

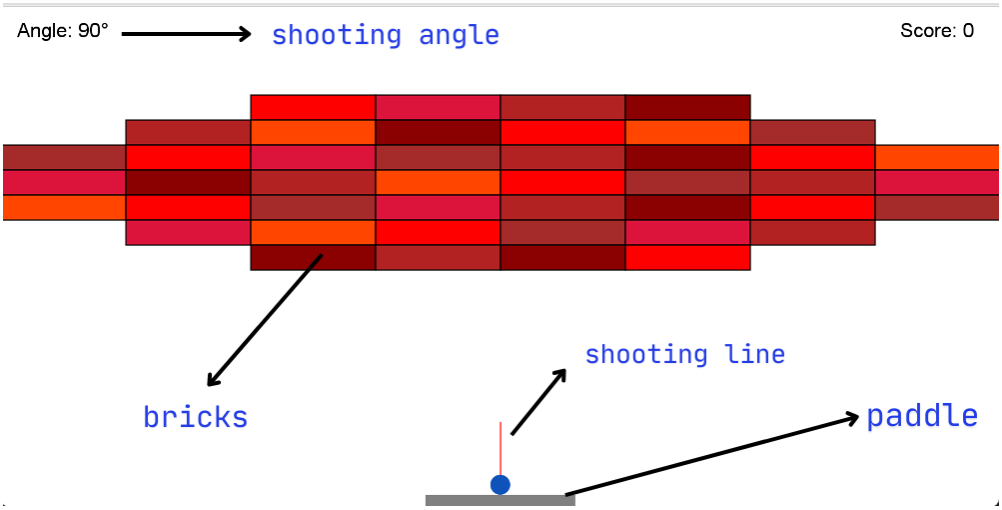
•StdDraw.text(): To show what you write at the game screen.

•StdDraw.clear(): To clean the screen.

•StdDraw.filledRectangle/Circle(): To draw shapes you want.

•StdDraw.enableDoubleBuffering(): To prevent flickering by swapping.

•StdDraw.show(): It is used with enableDoubleBuffering, it reflects drawing operations to the screen.



• **Game Mechanism:**

1) **Ball Movement:** There are two main variable to determine the ball movement.

•Ball Position : It shows the exact position of center of the ball, and it has two components ballPos[0] is x-component, ballPos[1] is y-component.

•Ball Velocity: It shows speed and direction of the ball , and it has two components too. ballVelocityComponents[0] is x-component, ballVelocityComponents[1] is y-component. Components are {0,0} and the speed is 5 before game starts.

→*At point 3.)* :
Before shooting is occured, Velocity components are updated according to shooting angle. cos() means x-component and sin() means y-component mathematically.

```
/*
3.) components of the ball's velocity , basic physics rules.
ballVelocityComponents and pallPos will be actively used.
 */

ballVelocityComponents[0] = ballVelocity * Math.cos(shootingAngle);
ballVelocityComponents[1] = ballVelocity * Math.sin(shootingAngle);
```

→ *At point 6.)* :

If game is started, ball position is updated according to this equation given in description. The ball moves in a straight line following this equation until it collides with a surface. The ball's position is updated each frame based on its velocity components.

$$(x_1, y_1) = (x_0 + v_x dt,\ y_0 + v_y dt)$$

```
// 6.) Updating game state if game is started and not paused or over.
if (gameHasStarted && !gameOver && !gameHasPaused) {
    // Updating ball position by given formula in description.
    ballPos[0] += ballVelocityComponents[0];
    ballPos[1] += ballVelocityComponents[1];
```

2) **Paddle Movement:** The paddle has the same movement style with the ball if the ball does not collide anywhere. Only and the most important part about the paddle movement is to remain the paddle within the predetermined canvas boundaries, preventing it from moving beyond the leftmost and rightmost edges.

→*At point 5.)* :

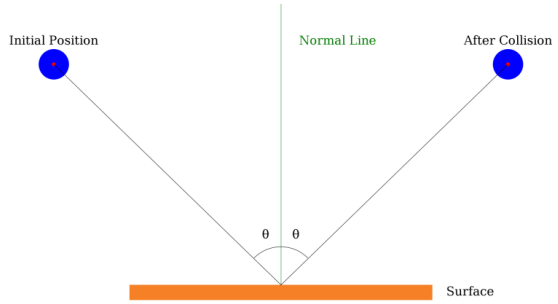
If right arrow is pressed and the game is active, paddle position update will be occured. paddlePos[0] + paddleSpeed is the desired new position, and xScale - paddleHalfWidth is the rightmost allowed position. So Math.min() is used to prevent the paddle from going outside of the canvas.

If left arrow is pressed and the game is active, paddle position update will be occured. paddlePos[0]- paddleSpeed is the desired new position, and paddleHalfWidth is the leftmost allowed position. So Math.max() is used, the function returns whichever value is larger, ensuring the paddle stays on screen.

```
//5.) Paddle movement and checking that the paddle inside the canvas.
if (isRightDown && gameHasStarted && !gameOver && !gameHasPaused) {
    paddlePos[0] = Math.min(paddlePos[0] + paddleSpeed, xScale - paddleHalfWidth);
}

if (isLeftDown && gameHasStarted && !gameOver && !gameHasPaused) {
    paddlePos[0] = Math.max(paddlePos[0] - paddleSpeed, paddleHalfWidth);
}
```

3) **Collisions:** If the ball is within a certain distance to the paddle, sides or bricks, it is assumed that the ball hits there. This distance must be equal to or less than the radius of the ball. However, the final velocity and direction change of the ball depending on whether the collision is surface or corner collision. There are different equations for surface and corner collisions.

   a) **Surface Collision:** If the ball collides with a surface, it is assumed that the ball is reflected at the same angle as coming angle to the surface.



a.1) Side Walls Collision:

→ *At point 7.) :*

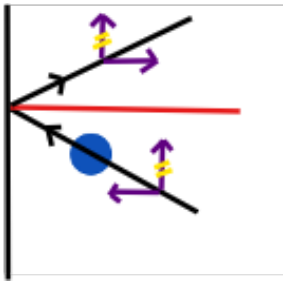
First surface collision type is side wall collisions. If the ball ball is closer the left wall than its radius (ballPos[0] - ballRadius <= 0), it is accepted as it hits the left wall.

If the ball is closer the right wall than its radius (ballPos[0] + ballRadius>=xScale), it is accepted as it hits the right wall.

Since radius is accepted as measure to collision, new position of the ball is arranged according to the ball radius. To understanding velocity components, figure.1 shows that y-component of the velocity does not change, and x-component of the velocity reverses its direction.

```
//7.) side walls (x-component)
if (ballPos[0] <= ballRadius || ballPos[0] + ballRadius >= xScale) {

    //if the ball hits the left side
    if (ballPos[0] - ballRadius <= 0) {

        //adjusting the new position of the ball
        ballPos[0] = ballRadius;
        //adjusting the new speed of the ball according to basic physics law.
        ballVelocityComponents[0] = -ballVelocityComponents[0];

    } else if (ballPos[0] + ballRadius >= xScale) {
        //adjusting the new position of the ball
        ballPos[0] = xScale - ballRadius;
        //adjusting the new speed of the ball according to basic physics law.
        ballVelocityComponents[0] = -ballVelocityComponents[0];
    }
}
```

figure.1)



a.2) Top and Bottom Wall Collisions:

→*At point 8.) and 9.) :*

Similar logic with side walls. However, this time y-component of the velocity changes, and if the ball hits the bottom wall, that means game is over. (gameOver=true) will be used at point 36.)

```
//8.) if position of the ball y component is closer than its radius to the top edge, it is accepted as it hits to the top edge.
if (ballPos[1] + ballRadius >= yScale) {

    ballPos[1] = yScale - ballRadius;
    ballVelocityComponents[1] = -ballVelocityComponents[1];
}

//9.) Detect ball falling below bottom edge
if (ballPos[1] - ballRadius <= 0) {
    gameOver = true;
}
```

a.3) Paddle Surface Collision:

➔*At point 10.) , 11.) and 12.) :*

By ballHitsThePaddle boolean, potential collision is examined.

If the rightmost point of the ball is on the right of the leftmost point of the paddle, and

If the leftmost point of the ball is on the left of the rightmost point of the paddle, and

If the lowest point of the ball is below the highest point of the paddle, and

If the highest point of the ball is above the lowest point of the paddle

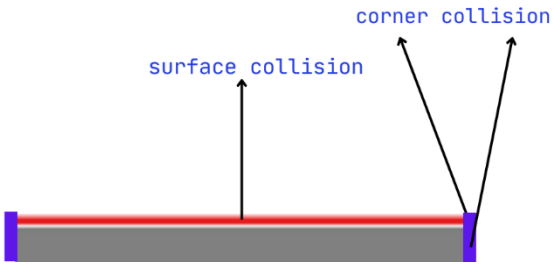
It means position of the ball is restricted to position of the paddle, and the ball has to collide with the paddle. However, it is unknown that whether this collision is surface or corner collision.

```
// 10.)Paddle collision
//Checks if the ball overlaps with the paddle (potential collision).
//if this boolean returns true, that means ball hits the paddle.
boolean ballHitsThePaddle = (ballPos[0] + ballRadius >= paddlePos[0] - paddleHalfWidth &&
        ballPos[0] - ballRadius <= paddlePos[0] + paddleHalfWidth &&
        ballPos[1] - ballRadius <= paddlePos[1] + paddleHalfHeight &&
        ballPos[1] + ballRadius >= paddlePos[1] - paddleHalfHeight);

//11.)
if (ballHitsThePaddle) {
```

To detect this, point 12.) and 12.1) are used. At point 12.1) it is decleared that if the ball is coming to surface of the paddle, it is accepted as surface collision, else [13.)] (it hits to sides or corner of the paddle) it is accepted as corner collision.After detecting it is a surface collision, this time y-component reverses (figure.1) and position is arranged according to radius of the ball.

```
// 12.) Check if the ball is approaching the paddle from above
//If this if statement is not written,the ball will fall if it goes parallel to the paddle
//for example: in the case if initial angle is equal to 0.
if (ballVelocityComponents[1] < 0) {

    //12.1) Check if it's a flat collision or corner collision
    if (ballPos[0] > paddlePos[0] - paddleHalfWidth &&
            ballPos[0] < paddlePos[0] + paddleHalfWidth) {
        // Top collision - simple reflection
        ballPos[1] = paddlePos[1] + paddleHalfHeight + ballRadius;
        ballVelocityComponents[1] = -ballVelocityComponents[1];

        //13.) Corner collision handling
    } else {
```

•If this part of my code is visualized, It appears that except for surface of the paddle, everywhere of it is accepted as corner. However due to the fact that even if the ball came to the side of the paddle, it could not bouncess of through the game, it definitely went to the bottom of the game and game is over. So, it is not important which collision type occurs at the side of the paddle in my solution.



a.4) Brick Surface Collision:

→ *At point 19.) and 20.)*:

It is not about collision yet, but I want to show how it is defined whether the ball hits the brick. All bricksCoordinates are checked, and if there is a brick then its coordinates are taken (brickX,brickY). Then the same logic is applied as in the paddle, and the unique position of collidingBrick is held. Example is given below. (It is for overlapping bricks.)

```
// 19.) Find all colliding bricks
for (int i = 0; i < brickCoordinates.length; i++) {
    if (!brickVisible[i]) {
        double brickX = brickCoordinates[i][0];
        double brickY = brickCoordinates[i][1];

        // 20.) Check for collision with this brick
        boolean collisionWithTheBrick = (ballPos[0] + ballRadius >= brickX - brickHalfWidth &&
                ballPos[0] - ballRadius <= brickX + brickHalfWidth &&
                ballPos[1] + ballRadius >= brickY - brickHalfHeight &&
                ballPos[1] - ballRadius <= brickY + brickHalfHeight);

        if (collisionWithTheBrick) {
            // Store this colliding brick
            collidingBricks[collidingCount] = i;
            collidingCount++;
        }
    }
}
```

```
[41, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[41, 42, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[36, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[36, 37, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[29, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[29, 30, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

→*At point 22.) and 23.)* :

By checking every collidingBricks we will determine whether it was broken with surface collision or corner collision. Additionally, By doing point 23.) it is detected where the ball comes from. [It will be used at point 29.)] If conditions of the corner collision are not satisfied by position of the ball, it is said that collision is surface collision. (conditions of the corner collision part is not here ; if this part was not understood, you can go to page…)

```
//22.) Process all colliding bricks first to determine overall collision type
for (int k = 0; k < collidingCount; k++) {
    int i = collidingBricks[k];
    double brickX = brickCoordinates[i][0];
    double brickY = brickCoordinates[i][1];

    // Mark brick as broken
    brickVisible[i] = true;
    score += 10;

    // 23.)to determine collision side
    //when the ball coming from the left
    double leftEdge = Math.abs((brickX - brickHalfWidth) - (ballPos[0] + ballRadius));
    //when the ball coming from the right
    double rightEdge = Math.abs((ballPos[0] - ballRadius) - (brickX + brickHalfWidth));
    //when the ball coming from the top
    double topEdge = Math.abs((ballPos[1] - ballRadius) - (brickY + brickHalfHeight));
    //when the ball coming from the bottom
    double bottomEdge = Math.abs((brickY - brickHalfHeight) - (ballPos[1] + ballRadius));
```

→ *At point 29.)* :

minOverlap provides us collision side by looking minimum of left-right and top-bottom. It uses this to determine whether collision is horizontal (xCollision) or vertical (yCollision).

```
        // 29.) if it is not corner collision.
        } else {
            // Find smallest overlap to determine collision side
            double minOverlap = Math.min(Math.min(leftEdge, rightEdge), Math.min(topEdge, bottomEdge));

            // Set collision flags based on direction
            if (minOverlap == leftEdge || minOverlap == rightEdge) {
                hasXCollision = true;
            }
            if (minOverlap == topEdge || minOverlap == bottomEdge) {
                hasYCollision = true;
            }
        }
    }
```

→ *At point 30.)* :

By using data above, it changes direction of velocity. If it is horizontal collision , x-component of the velocity changes. If it is vertical collision, y-component of the velocity changes (I have already explained logic behind it at figure1.)

```
// 30.) Only apply velocity changes if we didn't have a corner collision
if (!hasCornerCollision) {
    // Apply velocity changes for edge collisions
    if (hasXCollision) {
        ballVelocityComponents[0] = -ballVelocityComponents[0];
    }
    if (hasYCollision) {
        ballVelocityComponents[1] = -ballVelocityComponents[1];
    }
}
```

b) Corner Collision: Corner collision is different from surface collision, implementing direct reflection principle does not provide the correct reflection. The exact point of collision should be determined first, then tangent line should be used which is perpendicular to normal line. After these conditions are satisfied reflection principle can be applied. I have used dot product method to cope with corner collision. The dot product can be illustrated in below as figure2.)

***I use this page :** https://math.stackexchange.com/questions/13261/how-to-get-a-reflection-vector*

figure2.)

So, the *first step* is using the *dot product* to get a vertical vector that will be used in *step 2*.

With *step 1* my partial formula is: $2 \times (a + (-\vec{a}) \cdot \vec{n} \times n)$

mind the change of sign of $\vec{a}$ above, we "flipped" it

Then in *step 2*, I can write: $-\vec{a} + 2 \times (a + (-\vec{a}) \cdot \vec{n} \times n)$

Now, I can distribute: $-\vec{a} + 2 \times \vec{a} + 2 \times (-\vec{a}) \cdot \vec{n} \times n$

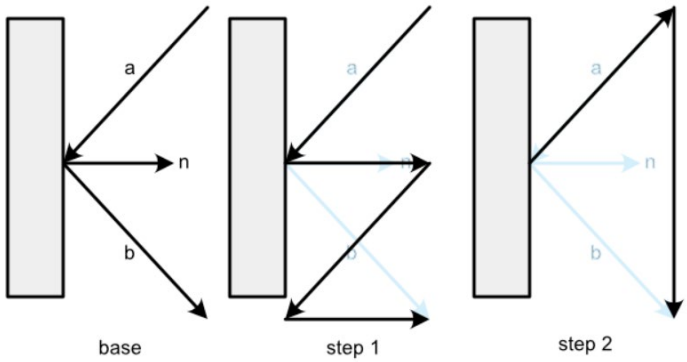
Then simplifying, I end up with: $\vec{a} + 2 \times (-\vec{a}) \cdot \vec{n} \times n$

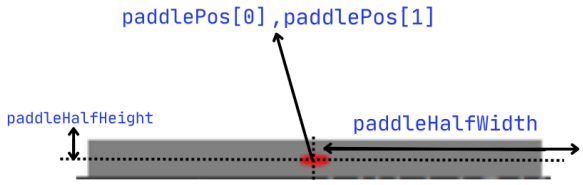
If you negate a vector in the dot product, you negate the result of the dot product.

$\vec{a} \cdot \vec{b} = -(-\vec{a}) \cdot \vec{b}$

That means that I can rewrite the formula like this:

$\vec{a} - 2 \times (\vec{a}) \cdot \vec{n} \times n$





b.1) Paddle Corner Collision:

→ *At point 13.)* :

At point 13.1) it determines if the ball coming from left, collision point will be upper left corner of the paddle. Else it will be upper right. At point 13.2) distance are defined, they are used at 13.3) to get nX and nY which can be considered as normal vectors for each components.

```
    //13.) Corner collision handling
} else {
    // Determine exact collision point on the paddle's corner
    double collisionPointX;
    double collisionPointY;

    // 13.1)Determine which corner was hit
    if (ballPos[0] < paddlePos[0]) {
        // Left corner collision
        collisionPointX = paddlePos[0] - paddleHalfWidth;
        collisionPointY = paddlePos[1] + paddleHalfHeight;
    } else {
        // Right corner collision
        collisionPointX = paddlePos[0] + paddleHalfWidth;
        collisionPointY = paddlePos[1] + paddleHalfHeight;
    }

    //13.2) Calculate the distance from collision point to ball center
    double distX = ballPos[0] - collisionPointX;
    double distY = ballPos[1] - collisionPointY;
```

→ *At point 13.4) and 13.5)* :

From dot product formula , dot product is determined, it is **inital velocity . normal vector**, then again from formula (below) reflected velocities are found.

ReflectedVelocity=InitialVelocitiy- 2 *(InitialVelocity . NormalVector) * Normal Vector

```
// 13.3)nX and nY they are normal vectors for each component.
double length = Math.sqrt(distX * distX + distY * distY);
double nX;
double nY;
nX = distX / length;
nY = distY / length;

// 13.4) Calculate the dot product
double dotProduct = ballVelocityComponents[0] * nX + ballVelocityComponents[1] * nY;

// 13.5) Formula for reflection: b = a - 2 * (a·n) * n
// Where a is the incident vector, n is the normal, and b is the reflection vector
ballVelocityComponents[0] = ballVelocityComponents[0] - 2 * dotProduct * nX;
ballVelocityComponents[1] = ballVelocityComponents[1] - 2 * dotProduct * nY;
```

→*At point 14.) :*

If this was not writte , then some bugs would occured. By Updating ballPos[1] as surface of the paddle + radius of the ball, bugs are prevented.

```
// 14.) Adjust ball position to prevent sticking to the paddle
ballPos[1] = paddlePos[1] + paddleHalfHeight + ballRadius;
```

I added two pictures to explain dot product part and point 14.) , they illustrates my works.

→https://hizliresim.com/i85jw98 →https://hizliresim.com/nbjviik

b.2) Brick Corner Collision:

Because there are a lot of bricks , I did not use the same method with the paddle to determine whether it is corner collision or not. However, after defining it is corner collision, dot product method is the same. To determine whether it is corner collision I divided corners into the parcels as shown figure 3.) If the ball comes from these parcels I both determine it is corner collision and which corner is the collision corner. In all corner detections , situation of brickCornerCollision changes and becomes true.

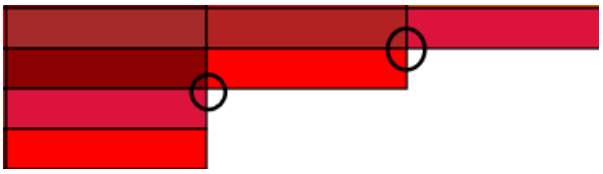
Figure3.): https://hizliresim.com/ilahi1a

```
// 24.1) Check top-left corner
if (ballPos[0] < brickX - brickHalfWidth && ballPos[1] > brickY + brickHalfHeight) {
    cornerX = brickX - brickHalfWidth;
    cornerY = brickY + brickHalfHeight;
    brickCornerCollision = true;
}
// 24.2) Check top-right corner
else if (ballPos[0] > brickX + brickHalfWidth && ballPos[1] > brickY + brickHalfHeight) {
    cornerX = brickX + brickHalfWidth;
    cornerY = brickY + brickHalfHeight;
    brickCornerCollision = true;
}
// 24.3) Check bottom-left corner
else if (ballPos[0] < brickX - brickHalfWidth && ballPos[1] < brickY - brickHalfHeight) {
    cornerX = brickX - brickHalfWidth;
    cornerY = brickY - brickHalfHeight;
    brickCornerCollision = true;
}
// 24.4) Check bottom-right corner
else if (ballPos[0] > brickX + brickHalfWidth && ballPos[1] < brickY - brickHalfHeight) {
    cornerX = brickX + brickHalfWidth;
    cornerY = brickY - brickHalfHeight;
    brickCornerCollision = true;
}
```

→*At point 25.), 26.) ,27.) and 28.)* :

distToBrickCorner is to be sure that is corner collision of the correct brick, because Three bricks may intersect in some corners [Figure4.)] and if it was not written, the ball might percieve the hidden one as corner collision too.(if the ball is actually touching the corner or just in the corner region.)

Figure4.)



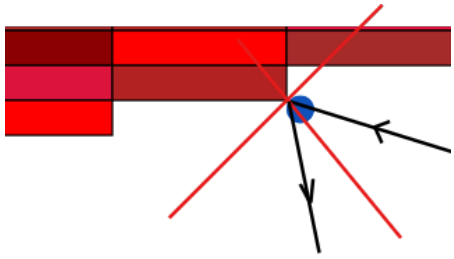
At point 26.) if the distance is closer than the ball radius, it means corner collision occurs.

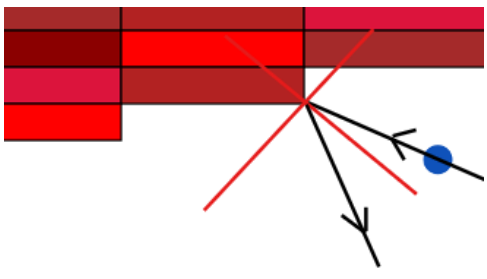
27.) dot product is the same.

28.) break condition prevents the ball from keep going after breaking a brick by corner collision.

```
// 25.) if it is corner collision.
if (brickCornerCollision) {
    // Check if ball is actually touching the corner
    double distToBrickCorner = Math.sqrt(Math.pow(ballPos[0] - cornerX, 2) + Math.pow(ballPos[1] - cornerY, 2));
    //26.) It prevents unexpected corner collisions.
    if (distToBrickCorner <= ballRadius) {
        hasCornerCollision = true;

        // 27.)Calculate the distance from collision point to ball center.
        double distX = ballPos[0] - cornerX;
        double distY = ballPos[1] - cornerY;

        // nX and nY they are normal vectors for each component.
        double length = Math.sqrt(distX * distX + distY * distY);
        double nX = distX / length;
        double nY = distY / length;

        // Calculate the dot product of velocity and normal.
        double dotProduct = ballVelocityComponents[0] * nX + ballVelocityComponents[1] * nY;

        // Formula for reflection:  b = a - 2 * (a·n) * n
        // Where a is the incident vector, n is the normal, and b is the reflection vector
        ballVelocityComponents[0] = ballVelocityComponents[0] - 2 * dotProduct * nX;
        ballVelocityComponents[1] = ballVelocityComponents[1] - 2 * dotProduct * nY;

        // 28.) Prevent further collision processing for this ball movement
        break;
```

Examples from the game are given below:





•Implementation Details:

Main Components

1. **Canvas Setup and Game Parameters**

    •Canvas size: 800x400 pixels

    •Ball: Blue circle with radius 8

    •Paddle: Gray rectangle (120x10)

    •Bricks: 44 colorful rectangles (100x20) arranged in 6 rows

```
// Canvas settings
double xScale = 800.0, yScale = 400.0;
StdDraw.setCanvasSize( canvasWidth: 800,  canvasHeight: 400);
StdDraw.setXscale(0.0, xScale);
StdDraw.setYscale(0.0, yScale);
StdDraw.enableDoubleBuffering();

// Game parameters
double ballRadius = 8;
double ballVelocity = 5;
double[] ballPos = {400, 18};
Color ballColor = new Color( r: 15,  g: 82,  b: 186);
Color[] colors = { new Color( r: 255,  g: 0,  b: 0), new Color( r: 220,  g: 20,  b: 60),
        new Color( r: 178,  g: 34,  b: 34), new Color( r: 139,  g: 0,  b: 0),
        new Color( r: 255,  g: 69,  b: 0), new Color( r: 165,  g: 42,  b: 42)
};

double[] paddlePos = {400, 5};
double paddleHalfWidth = 60;
double paddleHalfHeight = 5;
double paddleSpeed = 20;
Color paddleColor = new Color( r: 128,  g: 128,  b: 128);
```

2. **Game States**

- Not started (waiting for space bar)
- Active gameplay
- Paused
- Game over (win or loss)

```
//1.1) game status
int score = 0;
boolean gameHasStarted = false;
boolean gameOver = false;
boolean gameHasPaused = false;
boolean victory = false;
boolean spaceWasPressed = false;
```

→ At point 1.):

This point is to arrange status before while loop starts, firstly brickVisible[i] is equalized to false in for loop to handle collision situtations (If collision happens, brickVisible[i] will change and become true).

```
// 1.) Initially all bricks are visible (not broken)

boolean[] brickVisible = new boolean[brickCoordinates.length];
for (int i = 0; i < brickVisible.length; i++) {
    brickVisible[i] = false;
}
```

→*At point 2.) , 2.1) and 2.3 :*

At point 2.), Inputs are taken to arrange the paddle position and initial shooting angle.

At point 2.1), Space is currently pressed and it wasn't pressed in the previous frame means game is started. After entering this if statement, ballVelocityComponents are calculated according to shooting angle [before gameHasStarted, it has been calculated at point 2.3)]

```
//2.)continuity of the game is provided by while (true) loop.
while (true) {
    StdDraw.clear();

    boolean isSpaceDown = StdDraw.isKeyPressed( keycode: 32); // Space key
    boolean isLeftDown = StdDraw.isKeyPressed( keycode: 37); // Left arrow
    boolean isRightDown = StdDraw.isKeyPressed( keycode: 39); // Right arrow

    //2.1)If space is pressed and it is for  first time (spaceWasPressed = false) that means game is starting.
    if (isSpaceDown && !spaceWasPressed) {
        if (!gameHasStarted) {
            //by changing gameStarted status,  declared that the game started.
            gameHasStarted = true;
```

At point 2.3) after checking the game has not started yet, shooting angle is adjusted and this angle is used when the game starts.

```
//2.3) Arranging shooting angle:
if (!gameHasStarted) {
    if (isLeftDown) {
        shootingAngle = Math.min(shootingAngle + 0.025, Math.PI);
    }
    if (isRightDown) {
        shootingAngle = Math.max(0,shootingAngle - 0.025 );
    }
```

- → *At point 2.2):*

  If the game is already started but not over, pressing space will make the pause state (gameHasPaused) true.

  (spaceWasPressed = true) is set to prevent multiple triggers while space is held down. And prevent code from entering this if statement if user does not press to space again

  (!isSpaceDown) resets the spaceWasPressed flag when the space key is released. This allows the space key to be used again after releasing.

```
        //2.2)game status has already changed as started , so if space is pressed again then the code will arrive this else if statement.
    } else if (!gameOver) {
        gameHasPaused = !gameHasPaused;
    }
    spaceWasPressed = true;

} else if (!isSpaceDown) {
    spaceWasPressed = false;
}
```

→ *At point 15.), 16.), 17.) and 18.):*

This parts checks whether all bricks were destroyed if it is game is over and user wins the game , otherwise if there is still visible bricks, these bricks are checked one by one whether they have potential to hit by the ball. At point 19.) (page 6)

```
// 15.) Find colliding bricks
boolean allBricksDestroyed = true;

// 16.) Create arrays to store colliding brick indices
int[] collidingBricks = new int[brickCoordinates.length];
int collidingCount = 0;

// 17.) First check if any bricks are still visible
for (int i = 0; i < brickCoordinates.length; i++) {
    if (!brickVisible[i]) {
        allBricksDestroyed = false;
        break;
    }
}

//18.) If all bricks are destroyed, set victory condition
if (allBricksDestroyed) {
    gameOver = true;
    victory = true;
}
```

3.) Collision Detection, Game Mechanism and Physics Rules.

In report this part was briefly explained, so basically it can be said that the paddle and the ball movements are calculated, where the ball will hit is estimated accurately according to physics rules and ballRadius and ballPos are taken reference to this. After determining where the ball hits, it is checked whether the collision will be corner or surface collision. Appropriate collision steps are handled (Reflection Principle and Dot Product), and finally positions and velocities are updated according to collisions.

4.) Draw The Game: stdDraw library is actively used to draw the game.

→*at point 4.)* :Drawing the aim tool and texting angle. (.line, .setPenColor, .text)

```
//4.) Drawing the aim tool.
StdDraw.setPenColor(Color.RED);
double lineLength = 50;
double lineEndX = ballPos[0] + lineLength * Math.cos(shootingAngle);
double lineEndY = ballPos[1] + lineLength * Math.sin(shootingAngle);
StdDraw.line(ballPos[0], ballPos[1], lineEndX, lineEndY);
StdDraw.setPenColor(Color.BLACK);
StdDraw.text( x: 50,  y: 380,  text: "Angle: " + Math.round(Math.toDegrees(shootingAngle)) + "°");
```

→*at point 31.) and 32.)* : Drawing the ball and the paddle.

```
// 31.) Draw paddle
StdDraw.setPenColor(paddleColor);
StdDraw.filledRectangle(paddlePos[0], paddlePos[1], paddleHalfWidth, paddleHalfHeight);

// 32.) Draw ball
StdDraw.setPenColor(ballColor);
StdDraw.filledCircle(ballPos[0], ballPos[1], ballRadius);
```

→*at point 33.)* : Drawing bricks by for loop according to given brick coordinates. (.rectangle(), .filledRectangle())

```
// 33.) Draw bricks
for (int i = 0; i < brickCoordinates.length; i++) {
    if (!brickVisible[i]) {
        StdDraw.setPenColor(brickColors[i % brickColors.length]);
        StdDraw.filledRectangle(brickCoordinates[i][0], brickCoordinates[i][1], brickHalfWidth, brickHalfHeight);
        StdDraw.setPenColor(Color.BLACK);
        StdDraw.rectangle(brickCoordinates[i][0], brickCoordinates[i][1], brickHalfWidth, brickHalfHeight);
    }
}
```

→*at point 34.) and 35.)* : Texting score and game pause message.

```
// 34.) Draw score
StdDraw.setPenColor(Color.BLACK);
StdDraw.text( x: 750,  y: 380,  text: "Score: " + score);

// 35.)Game pause message
if (gameHasPaused) {
    StdDraw.setPenColor(Color.BLACK);
    StdDraw.text( x: 50,  y: 380,  text: "PAUSED");
}
```

→*at point 36.)*: Texting game end message.

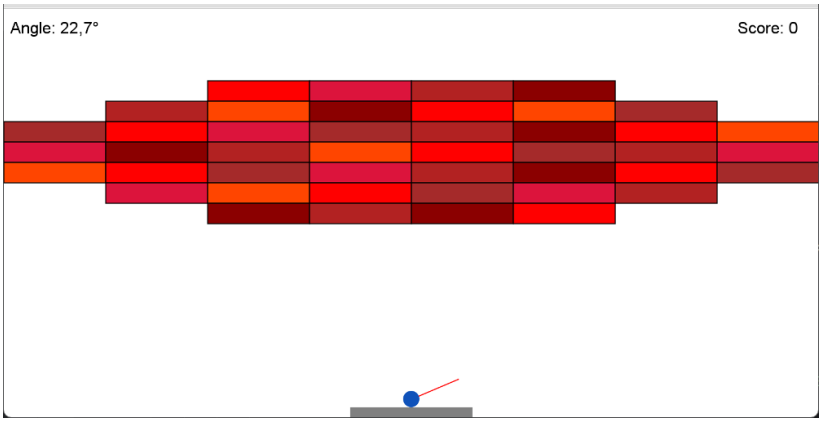
```
// 36.) Game end messages
if (gameOver) {
    StdDraw.setPenColor(new Color( r: 0,  g: 0,  b: 0,  a: 150));
    StdDraw.filledRectangle( x: xScale / 2,  y: yScale / 2,  halfWidth: xScale / 2,  halfHeight: yScale / 2);
    StdDraw.setPenColor(Color.WHITE);

    if (victory) {
        StdDraw.text( x: xScale / 2,  y: yScale / 2 + 30,  text: "VICTORY!");
    } else {
        StdDraw.text( x: xScale / 2,  y: yScale / 2 + 30,  text: "GAME OVER!");
    }

    StdDraw.text( x: xScale / 2,  y: yScale / 2 - 30,  text: "Score: " + score);
}
```

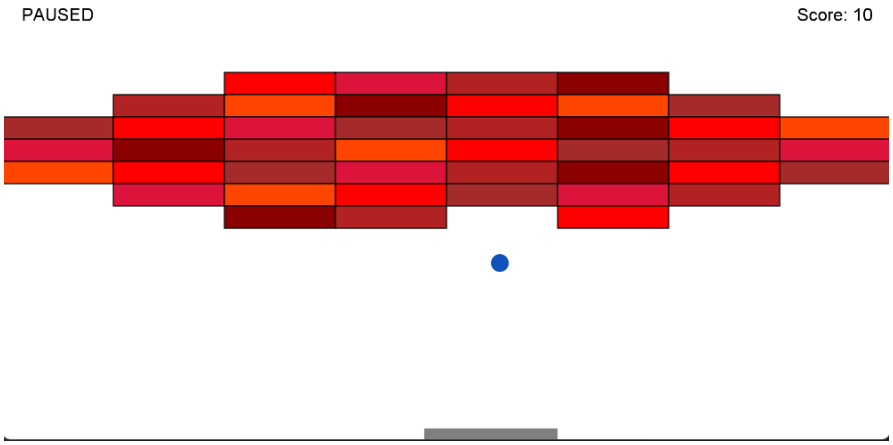
→ At point 37.) :

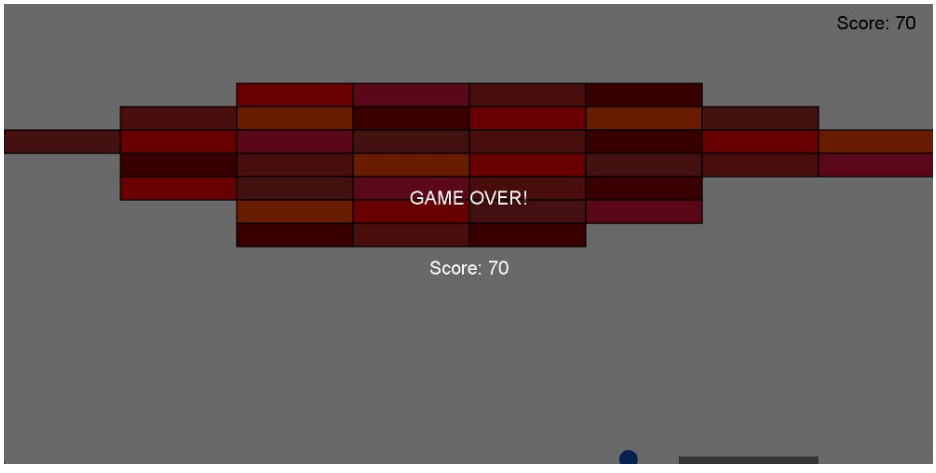
.show() updates the display with all the drawing commands that were issued since the last frame. The game uses double buffering (enabled earlier with StdDraw.enableDoubleBuffering()), which means:

- All drawing operations are performed on an off-screen buffer
- This call swaps the off-screen buffer with the on-screen buffer
- This prevents flickering by ensuring all elements appear simultaneously

```
//37.)
StdDraw.show();
StdDraw.pause( t: 18);
```


Angle: 22,7°
Score: 0


PAUSED
Score: 10


Score: 70
GAME OVER!
Score: 70


Score: 440
VICTORY!
Score: 440

Modified Game

This modification enables the ball to change its color randomly between choosen colors before the game starts if it hits the bricks. And bricks color are modified and restricted to 2 different colors.

→ Ten different colors are created and intial ballColor is choosen.

```
// Define 10 different custom ball colors for random selection
Color[] ballColors = {
        new Color( r: 0,  g: 0,  b: 0),
        new Color( r: 15,  g: 82,  b: 186),
        new Color( r: 220,  g: 20,  b: 60),
        new Color( r: 34,  g: 139,  b: 34),
        new Color( r: 255,  g: 215,  b: 0),
        new Color( r: 255,  g: 69,  b: 0),
        new Color( r: 138,  g: 43,  b: 226),
        new Color( r: 0,  g: 139,  b: 139),
        new Color( r: 255,  g: 20,  b: 147),
        new Color( r: 128,  g: 0,  b: 0)
};
// Initial ball color
Color ballColor = ballColors[1];
// Random number generator for color selection
Random random = new Random();
```

→ *At point 21.):*

after collision is detected , ballColor changes randomly everytime collision occured between the ball and a brick.

```
//21.)
if (collidingCount > 0) {
    // Create variables to track overall collision direction
    boolean hasXCollision = false;
    boolean hasYCollision = false;
    boolean hasCornerCollision = false;


    // Change ball color randomly when hitting any brick
    ballColor = ballColors[random.nextInt(ballColors.length)];
```

→ Brick Colors are arrenged to only yellow and red.

```
// Modified brick colors array to only contain yellow and red
Color[] colors = {
        new Color( r: 255,  g: 215,  b: 0),
        new Color( r: 220,  g: 20,  b: 60)
};
```

→ By this for loop and if statement half of the bricks is red and half of it is yellow.

```
// Modified brick colors array to alternate between yellow and red
Color[] brickColors = new Color[brickCoordinates.length];
for(int i = 0; i < brickColors.length; i++) {
    if( brickCoordinates[i][0]>400){
        brickColors[i]=colors[0];
    }else{
        brickColors[i]=colors[1];
    }
}
```