DXBALL

-by StdDraw Library-

First of all, video of non-modified version of game:

https://www.youtube.com/watch?v=Lt6lvfIRg7s

Secondly, video of modified version of game:

https://youtu.be/nJP5gTtOMzE

The game we worked on, which is commonly known as Brick Breaker, is a visually rendered partially physics-based video game which is still getting renovated by various video game developers. In this report, I will talk about the rendering library we used on Java, which is called StdDraw, and after that I will be explaining my project with its various details with covering every key aspect.

What is StdDraw?

StdDraw is a basic Java library that simplifies drawing and creating interactive graphics. It's designed to be easy for beginners, making it a good choice for smaller projects or learning programming, which will easen our process greatly.

With StdDraw, you can draw shapes like points, lines, rectangles, and circles, as well as text. It also allows you to control things like color, position, and size. You can track mouse clicks and movements, which is useful for building interactive applications and games.

Although StdDraw doesn't support advanced features like 3D graphics or complex physics, it's well-suited for simple 2D projects like Brick Breaker. It helps you focus on the main logic and interactions of your game without getting bogged down in complicated setups.

Overall, StdDraw is a useful tool for creating 2D graphics and basic interactive experiences with minimal effort. This makes it extremely suitable for our project.

How my code work?

After the parametric that given at the assigment file, two boolean value can be seen directly, which is called isThrown and isItOver. This two variable is the main two loop condition for two main segment of game, the throw angle phase and the main game phase. First, we will talk about the throw angle selection phase.

Angle Selection Loop

```
or(int i=0; i<brick_colors.length; i++) {
   StdDraw.setPenColor(brick_colors[i]);
   StdDraw.filledRectangle(brick_coordinates[i][0], brick_coordinates[i][1], brick_halfwidth, brick_halfheight);
```

By iterating every block index, StdDraw draws them with given position and sizes. Because no block is destroyed, no need to check anything.

```
// Draw the ball
StdDraw.setPenColor(ball_color);
StdDraw.filledCircle(initial_ball_pos[0],initial_ball_pos[1],ball_radius);

// Draw the paddle
StdDraw.setPenColor(paddle_color);
StdDraw.filledRectangle(paddle_pos[0],paddle_pos[1],paddle_halfwidth,paddle_halfheight);
```

Because everything is staying still, we draw ball and paddle in a static position.

```
if (StdDraw.isKeyPressed(KeyEvent.VK_LEFT)) {
    if (throwAngle < 180.0) {
        throwAngle = throwAngle + 1.0;
    }
}
if (StdDraw.isKeyPressed(KeyEvent.VK_RIGHT)) {
    if (throwAngle > 0.0) {
        throwAngle = throwAngle - 1.0;
    }
}
if (StdDraw.isKeyPressed(KeyEvent.VK_SPACE)) {
    isThrown = true; // Throw the ball when space is pressed
```

We are taking inputs here at every frame with the help of .isKeyPressed:

KeyEvent import helps us here to convert Key names to values that StdDraw understands.

Arrow Keys: For determining throw angle we use arrow keys. With checking whether is it still in acceptable angle range, we are allowing for player to choose in which direction he or she wants to play.

Space: After choosing the angle, player can throw the ball with space button.

With making isThrown true, it exits the loop and start the main segment of game.

However, before that we have a code part for drawing throw direction line.

```
// Draw the line indicating the throw angle
double throwAngleRadians = Math.toRadians(throwAngle);
StdDraw.setPenColor(StdDraw.RED); // set pen color
StdDraw.setPenRadius(0.005); // set pen size
StdDraw.line(initial_ball_pos[0],initial_ball_pos[1], x1: initial_ball_pos[0]+LineLength*Math.cos(throwAngleRadians),
```

In StdDraw, we draw a line with giivng its start and end point coordinates, line starts at the center of ball, and the end points are calculated with the sin and cos values of current throw angle, this gives a player a good expectation of how ball is going to move at start of game.

Main Segment, Where Real Game Starts:

Condition for main loop of game is “isItOver”, which is really self- explatonary. Starting with false value, when a situation that ends the game occurs, it turns to true value and exits the loop.

However, before going in loop, the horizontal and vertical values of ball’s speed get separated accordingly to sin and cos values. The conversion of degrees to radians are a must because Math.sin and Math.cos works with radians.

```
double throwAngleRadians = Math.toRadians(throwAngle);
double ball_velocity_x = ball_velocity * Math.cos(throwAngleRadians); // Ball velocity in x direction
double ball_velocity_y = ball_velocity * Math.sin(throwAngleRadians); // Ball velocity in y direction
```

```
int score = 0; // Player's score
boolean victory = false; // Track if the player has won
boolean isPaused = false; // Track whether the game is paused
boolean spacePressed = false; // Track if the space key is pressed
long frame = 0; // Frame counter
```

Score, victory and frame numbers values are identified here, because this informations are used in and outside the loops. All of them used at their respective locations.

```
frame += 1; // Increment frame counter

// Handle pause functionality
if (frame >10) {
    if (StdDraw.isKeyPressed(KeyEvent.VK_SPACE)) {
        if (!spacePressed) {
            isPaused = !isPaused; // Toggle pause state
            spacePressed = true;
        }
    } else {
        spacePressed = false;
    }

    // If paused, display "PAUSED" and skip the rest of the loop
    if (isPaused) {
        StdDraw.setPenColor(StdDraw.BLACK);
        StdDraw.text( x: 37,  y: y_scale - 12,  text: "PAUSED");
        StdDraw.show();
        StdDraw.pause(pauseDuration);
        continue;
    }
}
```

Why there is a frame counter? Because when you press space to throw the ball, a bug arises. The game directly pauses because this is the same button for pausing the game at the main loop of game. With making possible to pause game after the tenth frame, I solved this issue easily. With putting a spacePressed variable, the problem of contionusly switching to pause and to unpause with a short press to space is solved. While the game is paused, a paused text appears at the leftside of screen and with “continue” command, it just prevents any other thing to happen, which causes to a working pause effect.

```
// Handle paddle movement
if(StdDraw.isKeyPressed(KeyEvent.VK_RIGHT)){
    if (paddle_pos[0] + paddle_speed + paddle_halfwidth<= x_scale) {
        paddle_pos[0] = paddle_pos[0] + paddle_speed;
    }
}
if(StdDraw.isKeyPressed(KeyEvent.VK_LEFT)){
    if (paddle_pos[0] - paddle_speed - paddle_halfwidth >= 0) {
        paddle_pos[0] = paddle_pos[0] - paddle_speed;
    }
}
```

Simple movement input direction for paddle, without allowing it to go outside the map, it changes the coordinates of paddle with the value of paddle speed.

```
// Handle ball collision with walls
if ((initial_ball_pos[1] + ball_velocity_y) >= y_scale - ball_radius) {
    ball_velocity_y = -ball_velocity_y; // Reverse y velocity if ball hits the top wall
}
if ((initial_ball_pos[1]) <= 0.0 ) {
    isItOver = true; // End game if ball hits the bottom wall
    victory = false;
}
if ((initial_ball_pos[0] + ball_velocity_x) >= x_scale - ball_radius) {
    ball_velocity_x = -ball_velocity_x; // Reverse x velocity if ball hits the right wall
}
if ((initial_ball_pos[0] + ball_velocity_x) <= 0.0 + ball_radius) {
    ball_velocity_x = -ball_velocity_x; // Reverse x velocity if ball hits the left wall
}
```

One of the game finishers happen here, if ball touches to bottom line of the screen, it will end the game with a “Game Over” message. It can be easily seen that at every other border of map ball changes direction. With reverting the horizontal or vertical aspect of ball’s speed, the ball’s “bounce” effect happens. However, one of the questions that can be asked that why at every border other than bottom, the ball’s radius calculated and at bottom, it is not? Because at bottom with ball radius calculated some visual bugs are appearing because of the fast pedal movement. To prevent this, we are giving a game over message after a brief interval of time.

```
double[][] hitboxreference = new double[][]{{initial_ball_pos[0] + ball_velocity_x-ball_radius,initial_ball_pos[1] + ball_velocity_y}, //point 0
        {initial_ball_pos[0] + ball_velocity_x,initial_ball_pos[1] + ball_velocity_y-ball_radius}, //point 1
        {initial_ball_pos[0] + ball_velocity_x + ball_radius,initial_ball_pos[1] + ball_velocity_y}, //point 2
        {initial_ball_pos[0] + ball_velocity_x,initial_ball_pos[1] + ball_velocity_y + ball_radius}}; //point 3
```

The purpose of defining these four points is collision detection. The game checks whether:

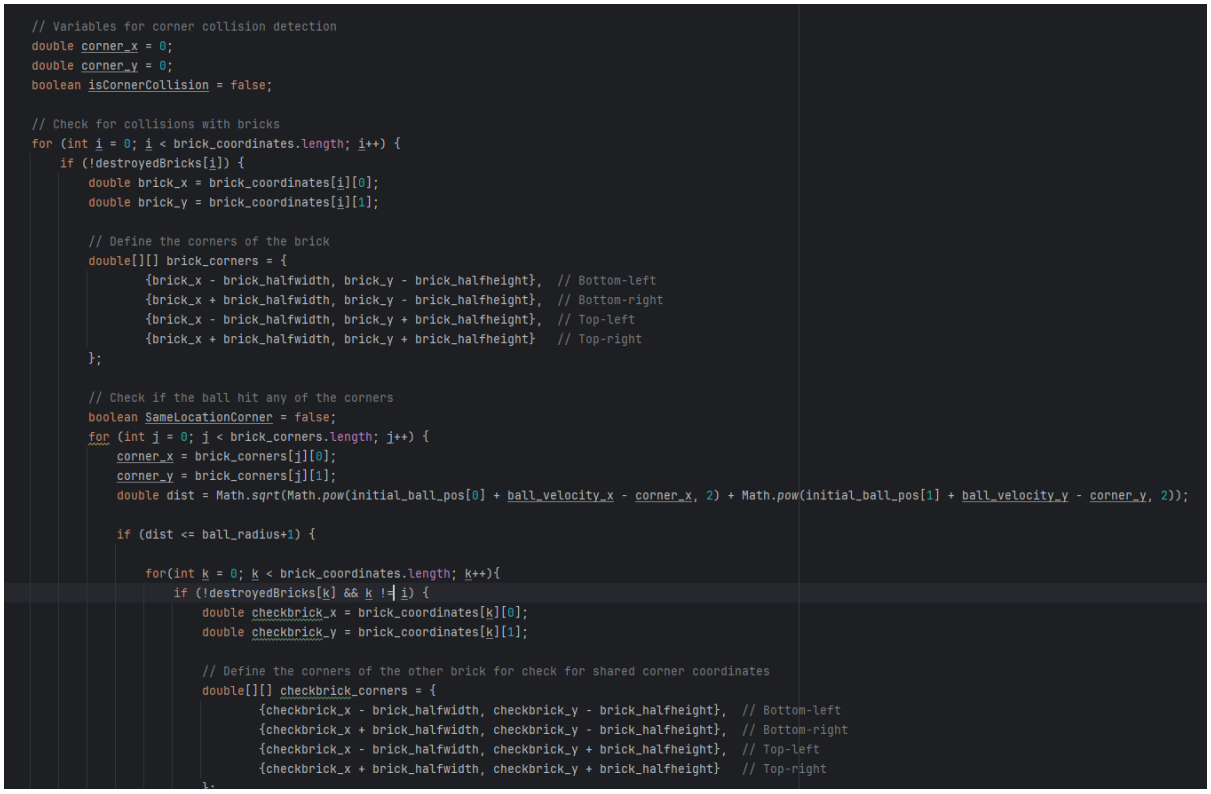
Point 0 (Left Edge) collides with a brick or wall on the left.

Point 1 (Bottom Edge) collides with the paddle or the bottom boundary.

Point 2 (Right Edge) collides with a brick or wall on the right.

Point 3 (Top Edge) collides with bricks or the top boundary.

By checking these four points, we can determine when and where the ball should bounce or whether it should cause a game-over event when it hits the bottom of the screen.



```
// Variables for corner collision detection
double corner_x = 0;
double corner_y = 0;
boolean isCornerCollision = false;

// Check for collisions with bricks
for (int i = 0; i < brick_coordinates.length; i++) {
    if (!destroyedBricks[i]) {
        double brick_x = brick_coordinates[i][0];
        double brick_y = brick_coordinates[i][1];

        // Define the corners of the brick
        double[][] brick_corners = {
                {brick_x - brick_halfwidth, brick_y - brick_halfheight},  // Bottom-left
                {brick_x + brick_halfwidth, brick_y - brick_halfheight},  // Bottom-right
                {brick_x - brick_halfwidth, brick_y + brick_halfheight},  // Top-left
                {brick_x + brick_halfwidth, brick_y + brick_halfheight}   // Top-right
        };

        // Check if the ball hit any of the corners
        boolean SameLocationCorner = false;
        for (int j = 0; j < brick_corners.length; j++) {
            corner_x = brick_corners[j][0];
            corner_y = brick_corners[j][1];
            double dist = Math.sqrt(Math.pow(initial_ball_pos[0] + ball_velocity_x - corner_x, 2) + Math.pow(initial_ball_pos[1] + ball_velocity_y - corner_y, 2));

            if (dist <= ball_radius+1) {

                for(int k = 0; k < brick_coordinates.length; k++){
                    if (!destroyedBricks[k] && k != i) {
                        double checkbrick_x = brick_coordinates[k][0];
                        double checkbrick_y = brick_coordinates[k][1];

                        // Define the corners of the other brick for check for shared corner coordinates
                        double[][] checkbrick_corners = {
                                {checkbrick_x - brick_halfwidth, checkbrick_y - brick_halfheight},  // Bottom-left
                                {checkbrick_x + brick_halfwidth, checkbrick_y - brick_halfheight},  // Bottom-right
                                {checkbrick_x - brick_halfwidth, checkbrick_y + brick_halfheight},  // Top-left
                                {checkbrick_x + brick_halfwidth, checkbrick_y + brick_halfheight}   // Top-right
                        };
```

This section of the code is responsible for detecting and handling **corner collisions** between the ball and the bricks, ensuring that when the ball hits the corner of a brick, it behaves correctly while also accounting for shared corners between adjacent bricks. The process starts by initializing three key variables: `corner_x` and `corner_y`, which store the coordinates of the detected corner collision, and `isCornerCollision`, a boolean flag that tracks whether the ball has collided with a corner. The code then iterates through all the bricks in the game, skipping over those that have already been destroyed. For each brick, the center coordinates are retrieved, and then the four corners of the brick are calculated using the brick's half-width and half-height. Once the corners are determined, the program checks if the ball's next position (based on its velocity) is close enough to any of these corners by computing the **Euclidean distance** between the ball's next position and each corner. If this distance is less than or equal to the **ball's radius + 1**, it means the ball has made contact with the corner. However, before finalizing this collision, the code performs an additional check to determine whether the detected corner is also a shared corner with another adjacent brick. To do this, it iterates through all the bricks again, checking their corner positions. If another brick has a corner at the exact same coordinates as the one that was hit, the program marks this as a **shared corner** and sets `SameLocationCorner` to `true`. This is a crucial step because if the corner is shared, it is likely that the ball is making contact with multiple bricks at the same time, which could lead to unintended behavior in the physics simulation, such as multiple bricks being destroyed at once or an incorrect bounce angle. If the detected corner is not shared with another brick, the program confirms the collision by setting `isCornerCollision` to `true`, marking the brick as destroyed, increasing the score by 10 points, and breaking out of the loop to ensure that only one brick

is processed per frame. However, if the corner is shared with another brick, the collision is ignored (`isCornerCollision = false`), preventing any destruction or scoring changes. This entire system ensures **realistic corner collision physics**, preventing unintended behaviors where the ball might bounce incorrectly or delete multiple bricks unfairly. Without this corner-handling mechanism, the ball might pass through bricks without bouncing correctly, or worse, cause multiple bricks to disappear due to overlapping collision detections. By ensuring that only non-shared corners result in destruction and collision effects, the game maintains fair gameplay, correct physics, and prevents erratic ball movement. The logic also ensures that when the ball collides with a corner, it correctly responds by changing its velocity direction, though the specific velocity update logic handled elsewhere in the program. Overall, this section of the code plays a **critical role in ensuring the smooth functioning of brick destruction and ball movement**, preventing bugs, and maintaining proper collision physics in the game.

```
if (isCornerCollision) {
    // Normalize normal vector

    double normal_x, normal_y;
    normal_x = initial_ball_pos[0] - 2*ball_velocity_x - corner_x;
    normal_y = initial_ball_pos[1] - 2*ball_velocity_y - corner_y;

    double norm_length = Math.sqrt((normal_x * normal_x) + (normal_y * normal_y));
    normal_x /= norm_length;
    normal_y /= norm_length;

    // Reflect velocity using formula: V' = V - 2(V . N)N
    double dotProduct = ball_velocity_x * normal_x + ball_velocity_y * normal_y;
    ball_velocity_x -= 2 * dotProduct * normal_x;
    ball_velocity_y -= 2 * dotProduct * normal_y;
} else{
    // Handle regular collisions with bricks
    boolean directionChangeVertical = false;
    boolean directionChangeHorizontal = false;
    for (int c = 0; c < hitboxreference.length; c++) {
        for (int i = 0; i < brick_coordinates.length; i++) {
            if (!destroyedBricks[i]) {
                if ((brick_coordinates[i][0] - brick_halfwidth <= hitboxreference[c][0]) && (hitboxreference[c][0] <= brick_coordinates[i][0] + brick_halfwidth)) {
                    if ((brick_coordinates[i][1] - brick_halfheight <= hitboxreference[c][1]) && (hitboxreference[c][1] <= brick_coordinates[i][1] + brick_halfheight)
                        destroyedBricks[i] = true;
                        score += 10;

                        if ((c == 0) || (c == 2) && (!directionChangeHorizontal)) {
                            ball_velocity_x = -ball_velocity_x;
                            directionChangeHorizontal = true;
                        }

                        if ((c == 1) || (c == 3) && (!directionChangeVertical)) {
                            ball_velocity_y = -ball_velocity_y;
                            directionChangeVertical = true;
                        }
                    }
                }
```

This section of the code is crucial for handling ball collisions with bricks, ensuring the game's physics feel smooth and realistic. It first checks whether the ball has **collided with a corner** of a brick (`isCornerCollision == true`). If this is the case, the program determines a **normal vector** from the collision point by subtracting the predicted ball position from the corner coordinates. This vector is then **normalized**, meaning its length is adjusted to 1 while preserving its direction. Normalization is essential for accurate calculations when reflecting the ball's movement. Using the **vector reflection formula** $V' = V - 2(V \cdot N)N$, where V is the balls velocity and N is the normal vector, the ball's velocity is updated to reflect the correct bounce angle off the brick corner. This ensures that the ball

reacts naturally when striking a brick at an angle rather than simply reversing one of its velocity components.

However, if no corner collision is detected, the program proceeds to check for **regular brick collisions** using the predefined **hitbox reference points** around the ball. These hitbox points approximate the ball's edges and help determine whether the ball is inside the boundaries of a brick. The program iterates through all existing bricks and checks whether any of these hitbox points fall within a brick's coordinate range. If a collision is found, the respective brick is **marked as destroyed**, and the **score increases by 10** to reflect successful destruction. The ball's velocity is then adjusted based on the collision location: if the ball hits **horizontally (left or right sides of a brick), the x-velocity is inverted**, whereas if the ball hits **vertically (top or bottom sides of a brick), the y-velocity is inverted**.

To ensure that the ball does not reverse both velocity components unnecessarily, two boolean flags (`directionChangeVertical` and `directionChangeHorizontal`) are used. These flags ensure that a **single collision does not cause multiple unwanted velocity changes**, preventing erratic movement. By carefully handling both **corner and direct collisions**, this logic ensures that the game provides a **realistic and engaging physics system**, making each bounce and interaction with bricks feel intuitive and predictable for the player.

```
// Define the corners of the paddle
double[][] paddleCorners = {
        {paddle_pos[0] - paddle_halfwidth, paddle_pos[1] - paddle_halfheight}, // Bottom-left
        {paddle_pos[0] + paddle_halfwidth, paddle_pos[1] - paddle_halfheight}, // Bottom-right
        {paddle_pos[0] - paddle_halfwidth, paddle_pos[1] + paddle_halfheight}, // Top-left
        {paddle_pos[0] + paddle_halfwidth, paddle_pos[1] + paddle_halfheight}  // Top-right
};

// Check for corner collisions with the paddle
boolean isPaddleCornerCollision = false;
for (int i = 0; i < paddleCorners.length; i++) {
    corner_x = paddleCorners[i][0];
    corner_y = paddleCorners[i][1];
    double dist = Math.sqrt(Math.pow(initial_ball_pos[0] + ball_velocity_x - corner_x, 2) + Math.pow(initial_ball_pos[1] + ball_velocity_y - corner_y, 2));

    if (dist <= ball_radius) {
        isPaddleCornerCollision = true;
        break;
    }
}

// Handle paddle corner collision
if (isPaddleCornerCollision) {
    // Reflect the ball's velocity based on the normal vector at the collision point
    double normal_x = initial_ball_pos[0] -  ball_velocity_x - paddle_pos[0];
    double normal_y = initial_ball_pos[1] -  ball_velocity_y - paddle_pos[1];
    double norm_length = Math.sqrt((normal_x * normal_x) + (normal_y * normal_y));
    normal_x /= norm_length;
    normal_y /= norm_length;

    double dotProduct = ball_velocity_x * normal_x + ball_velocity_y * normal_y;
    ball_velocity_x -= 2 * dotProduct * normal_x;
    ball_velocity_y -= 2 * dotProduct * normal_y;
} else {
    // Regular paddle collision detection
    if ((paddle_pos[0] - paddle_halfwidth < initial_ball_pos[0] + ball_velocity_x - ball_radius + 1) && (initial_ball_pos[0] + ball_velocity_x - ball_radius < paddle_pos[
        if ((paddle_pos[1] - paddle_halfheight < initial_ball_pos[1] + ball_velocity_y - ball_radius + 1) && (initial_ball_pos[1] + ball_velocity_y - ball_radius < paddle
            ball_velocity_y = -ball_velocity_y;
        }
    }
```

This section of the code is responsible for handling collisions between the ball and the paddle, ensuring that the ball reacts naturally when it makes contact. First, the program defines the four **corner points** of the paddle using its position (`paddle_pos`) and

dimensions (`paddle_halfwidth` and `paddle_halfheight`). These corners help determine whether the ball collides specifically with a paddle's **edge** rather than its flat surface. To detect such a collision, the program iterates through each of these corners and calculates the **Euclidean distance** between the ball's predicted next position and the corner in question. If this distance is **less than or equal to the ball's radius**, it indicates a **corner collision**, and the `isPaddleCornerCollision` flag is set to `true`, causing the loop to break early. When a **corner collision occurs**, the ball's velocity is reflected using a normal vector similar to how brick corner collisions are handled. The normal vector is computed by subtracting the predicted ball position from the paddle's position, and then it is **normalized** to ensure its length is 1. The reflection formula $V' = V - 2(V \cdot N)N$ is then applied to modify the ball's velocity, ensuring that the ball bounces away at the correct angle. However, if the ball **does not collide with a corner**, the program instead checks for a **regular paddle collision**, meaning the ball is hitting the flat surface of the paddle rather than an edge. This is done by verifying whether the ball's next position falls within the **horizontal and vertical range** of the paddle. If the ball is within these boundaries, it means that the collision has occurred, and the **y-velocity is inverted**, causing the ball to bounce upwards as expected. The small adjustments (`+1` and `-1` in the comparisons) help prevent **collision inaccuracies**, ensuring that minor floating-point errors do not allow the ball to phase through the paddle. This dual-layered approach, where both **corner and flat-surface collisions** are considered separately, provides a more realistic and smooth bouncing effect, making the gameplay feel more polished and engaging for the player.

```
// Update the ball's position
initial_ball_pos[0] = initial_ball_pos[0] + ball_velocity_x;
initial_ball_pos[1] = initial_ball_pos[1] + ball_velocity_y;
```

This part of the code simply updates the ball's position by adding its current velocity components (`ball_velocity_x` and `ball_velocity_y`) to its respective x and y coordinates in `initial_ball_pos`. This ensures that the ball moves in the direction dictated by its velocity on each frame, creating smooth motion. The velocity values determine both the speed and trajectory of the ball, and they are updated elsewhere in the code based on collisions with walls, bricks, and the paddle. This straightforward calculation is fundamental to the ball's movement and ensures continuous gameplay.

```
// Draw the ball
StdDraw.setPenColor(ball_color);
StdDraw.filledCircle(initial_ball_pos[0],initial_ball_pos[1],ball_radius);

// Draw the paddle
StdDraw.setPenColor(paddle_color);
StdDraw.filledRectangle(paddle_pos[0],paddle_pos[1],paddle_halfwidth,paddle_halfheight);

// Draw the remaining bricks
for(int i=0; i<brick_colors.length; i++) {
    if (!destroyedBricks[i]) {
        StdDraw.setPenColor(brick_colors[i]);
        StdDraw.filledRectangle(brick_coordinates[i][0], brick_coordinates[i][1], brick_halfwidth, brick_halfheight);
    }
}
```

This section of the code is responsible for rendering the key visual elements of the game: the ball, the paddle, and the bricks. First, the ball is drawn as a filled circle at its current position (`initial_ball_pos[0], initial_ball_pos[1]`) with the defined radius (`ball_radius`), using the specified `ball_color`. Next, the paddle is drawn as a filled rectangle at its designated position (`paddle_pos[0], paddle_pos[1]`), with its width and height defined by `paddle_halfwidth` and `paddle_halfheight`, respectively, ensuring it appears correctly on the screen. Lastly, the loop iterates through all the bricks, checking if each brick is still present in the game by verifying the `destroyedBricks` array. If a brick has not been destroyed, it is drawn as a filled rectangle at its assigned coordinates (`brick_coordinates[i][0], brick_coordinates[i][1]`) with its specific color from the `brick_colors` array. This ensures that only the remaining bricks are displayed, creating a dynamic visual representation of the game as bricks are broken throughout gameplay.

```
// Check if all bricks are destroyed
boolean allBricksDestroyed = true;
for (int i = 0; i < destroyedBricks.length; i++) {
    if (!destroyedBricks[i]) {
        allBricksDestroyed = false;
        break;
    }
}
if (allBricksDestroyed) {
    isItOver = true;
    victory = true;
}
```

This section of the code is responsible for checking if all the bricks in the game have been destroyed, determining whether the player has won. The boolean variable `allBricksDestroyed` is initially set to `true`, assuming that all bricks are gone. Then, a loop iterates through the `destroyedBricks` array, which keeps track of the status of each brick. If it finds any brick that has not been destroyed (`!destroyedBricks[i]`), it sets `allBricksDestroyed` to `false` and breaks out of the loop early to improve efficiency. After the loop, if `allBricksDestroyed` remains `true`, the game is marked as over by

setting `isItOver = true`, and the `victory` flag is set to `true`, indicating that the player has successfully broken all bricks and won the game.

```
// Clear the score display area
StdDraw.setPenColor(StdDraw.WHITE);
StdDraw.filledRectangle( x: x_scale-50, y: y_scale-20, halfWidth: 50, halfHeight: 20);

// Define fonts for the end game message
Font scoreFont = new Font( name: "Serif", Font.BOLD,  size: 30);
Font font = new Font( name: "Serif", Font.BOLD,  size: 60);
StdDraw.setFont(font);
StdDraw.setPenColor(StdDraw.BLACK);

// Display victory or game over message
if (victory){
    StdDraw.text( x: x_scale/2, y: y_scale/2, text: "VICTORY!");
    StdDraw.setFont(scoreFont);
    StdDraw.text( x: x_scale/2, y: y_scale/6,  text: "Score: " + score);
} else{
    StdDraw.text( x: x_scale/2, y: y_scale/3.5,  text: "-GAME OVER-");
    StdDraw.setFont(scoreFont);
    StdDraw.text( x: x_scale/2, y: y_scale/6,  text: "Score: " + score);
}

// Show the final frame
StdDraw.show();
```

In this part, if Victory is true and loop is ended, the game finishes wiht a victory message, however, if game ends with a false victory value, the game ends with a Game Over message and score counter.

Modified

New rainbow level has been added, paddle has been smaller and velocity is now higher, with the placement of bricks, corner collisions are becoming more frequently now, so it is a lot harder.

This is all the explained process of my Dxball Code. Thank you for your time.