```
int screen = 0; // 0--Main Menu, 1--Level 1, 2--Level 2
Button[] buttons = new Button[5];
PhysicsEngine engine;
Bird bird;
Brick[] bricks;
Catapult catapult;

void setup() {
  //size(1600, 900);
  size(1000, 800);
  engine = new PhysicsEngine();

  buttons[0] = new Button(300, 300, 50, "SQUARE_BUTTON", "1",
"none"); //300, 300
  buttons[1] = new Button(400, 300, 50, "SQUARE_BUTTON", "2",
"none"); //400, 300
  buttons[2] = new Button(700, 100, 50, "CIRCULAR_BUTTON", "none",
"pause"); //1300, 100
  buttons[3] = new Button(800, 100, 50, "CIRCULAR_BUTTON", "none",
"restart"); //1400, 100
  buttons[4] = new Button(900, 100, 50, "CIRCULAR_BUTTON", "none",
"mainMenu"); //1500, 100

  bird = new Bird(200, 700);
  engine.addObject(bird);

  bricks[0] = new Brick(100, 100, "stone");
  for (Brick brick : bricks) {
    engine.addObject(brick);
  }
}

void draw() {
  background(200);

  switch(screen) {
    case 0:
      displayMainMenu();
      break;
    case 1:
      displayLevel1();
      break;
    case 2:
      displayLevel2();
      break;
  }
}

void displayMainMenu() {
  background(200);
  buttons[0].draw();
  buttons[1].draw();
}

void displayLevel1() {
  background(200);
  buttons[2].draw();
  buttons[3].draw();
  buttons[4].draw();
```

```
  //catapult.draw();

  engine.update();
  for (PhysicsObject obj : engine.objects) {
    obj.display();
  }
}

void displayLevel2() {
  background(200);
  buttons[2].draw();
  buttons[3].draw();
  buttons[4].draw();
}

void mousePressed() {
  for (Button button : buttons) {
    if (button.try_click(mouseX, mouseY)) {
      button.buttonClicked = true;
    }
  }

  //if (bird.try_click(mouseX, mouseY)) {
  //  catapult.dragging = true;
  //}
}

void mouseReleased() {
  for (int i = 0; i < buttons.length; i++) {
    if (buttons[i].buttonClicked) {
      buttons[i].buttonClicked = false;
      if (i == 0) {
        println("Level 1");
        screen = 1;
      } else if (i == 1) {
        println("Level 2");
        screen = 2;
      } else if (i == 2) {
        // Pause functionality
      } else if (i == 3) {
        // Restart functionality
        resetLevel();
      } else if (i == 4) {
        println("Main Menu");
        screen = 0;
      }
    }
  }
  catapult.mouseReleased(bird);
}

void mouseDragged() {
  catapult.updateDrag(mouseX, mouseY, bird);
}

void resetLevel() {
  //
}
bricks[0] = new Brick(100, 100, "stone"): NullPointerException
```

The **`NullPointerException`** is occurring because the **`bricks`** array is not initialized before attempting to add bricks to it. You need to initialize the **`bricks`** array in the **`setup()`** function. Here's the corrected code:

java Copy code

```
int screen = 0; // 0--Main Menu, 1--Level 1, 2--Level 2 Button[] buttons = new
Button[5]; PhysicsEngine engine; Bird bird; Brick[] bricks; Catapult catapult; void
setup() { //size(1600, 900); size(1000, 800); engine = new PhysicsEngine(); buttons[0] =
new Button(300, 300, 50, "SQUARE_BUTTON", "1", "none"); //300, 300 buttons[1] = new
Button(400, 300, 50, "SQUARE_BUTTON", "2", "none"); //400, 300 buttons[2] = new
Button(700, 100, 50, "CIRCULAR_BUTTON", "none", "pause"); //1300, 100 buttons[3] = new
Button(800, 100, 50, "CIRCULAR_BUTTON", "none", "restart"); //1400, 100 buttons[4] = new
Button(900, 100, 50, "CIRCULAR_BUTTON", "none", "mainMenu"); //1500, 100 bird = new
Bird(200, 700); engine.addObject(bird); bricks = new Brick[1]; // Initialize the bricks
array with the desired size bricks[0] = new Brick(100, 100, "stone"); // Initialize the
brick objects for (Brick brick : bricks) { engine.addObject(brick); } } void draw() {
background(200); switch(screen) { case 0: displayMainMenu(); break; case 1:
displayLevel1(); break; case 2: displayLevel2(); break; } } void displayMainMenu() {
background(200); buttons[0].draw(); buttons[1].draw(); } void displayLevel1() {
background(200); buttons[2].draw(); buttons[3].draw(); buttons[4].draw();
//catapult.draw(); engine.update(); for (PhysicsObject obj : engine.objects) {
obj.display(); } } void displayLevel2() { background(200); buttons[2].draw();
buttons[3].draw(); buttons[4].draw(); } void mousePressed() { for (Button button :
buttons) { if (button.try_click(mouseX, mouseY)) { button.buttonClicked = true; } } //if
(bird.try_click(mouseX, mouseY)) { // catapult.dragging = true; //} } void
mouseReleased() { for (int i = 0; i < buttons.length; i++) { if
(buttons[i].buttonClicked) { buttons[i].buttonClicked = false; if (i == 0) {
println("Level 1"); screen = 1; } else if (i == 1) { println("Level 2"); screen = 2; }
else if (i == 2) { // Pause functionality } else if (i == 3) { // Restart functionality
resetLevel(); } else if (i == 4) { println("Main Menu"); screen = 0; } } }
catapult.mouseReleased(bird); } void mouseDragged() { catapult.updateDrag(mouseX,
mouseY, bird); } void resetLevel() { // }
```

This should resolve the **`NullPointerException`** by ensuring the **`bricks`** array is properly initialized before accessing its elements.

```
boolean checkCollision(PhysicsObject other) {
  if (!isCircle && !other.isCircle) {
    // Box vs Box
    return (pos.x < other.pos.x + other.w &&
        pos.x + w > other.pos.x &&
        pos.y < other.pos.y + other.h &&
        pos.y + h > other.pos.y);
  } else if (isCircle && other.isCircle) {
    // Circle vs Circle
    float distance = PVector.dist(pos, other.pos);
    return distance < radius + other.radius;
  } else if (!isCircle && other.isCircle || isCircle && !other.isCircle) {
    // Box vs Circle
    println("Collision!");
    PhysicsObject box = isCircle ? other : this;
    PhysicsObject circle = isCircle ? this : other;

    float testX = circle.pos.x;
    float testY = circle.pos.y;

    if (circle.pos.x < box.pos.x) testX = box.pos.x;
```

```
    else if (circle.pos.x > box.pos.x + box.w) testX = box.pos.x + box.w;

    if (circle.pos.y < box.pos.y) testY = box.pos.y;
    else if (circle.pos.y > box.pos.y + box.h) testY = box.pos.y + box.h;

    float distX = circle.pos.x - testX;
    float distY = circle.pos.y - testY;
    float distance = sqrt(distX*distX + distY*distY);

    return distance <= circle.radius;
  }
```

This method must return a result of type boolean

The method **`checkCollision`** indeed needs to return a boolean value in all code paths. Currently, the method does not return a value at the end if none of the conditions match, which leads to the compile error. To fix this, ensure that a boolean value is always returned. Here's the updated version of your **`checkCollision`** method:

java Copy code

```
boolean checkCollision(PhysicsObject other) { if (!isCircle && !other.isCircle) { // Box vs Box return (pos.x < other.pos.x + other.w && pos.x + w > other.pos.x && pos.y < other.pos.y + other.h && pos.y + h > other.pos.y); } else if (isCircle && other.isCircle) { // Circle vs Circle float distance = PVector.dist(pos, other.pos); return distance < radius + other.radius; } else { // Box vs Circle println("Collision!"); PhysicsObject box = isCircle ? other : this; PhysicsObject circle = isCircle ? this : other; float testX = circle.pos.x; float testY = circle.pos.y; if (circle.pos.x < box.pos.x) testX = box.pos.x; else if (circle.pos.x > box.pos.x + box.w) testX = box.pos.x + box.w; if (circle.pos.y < box.pos.y) testY = box.pos.y; else if (circle.pos.y > box.pos.y + box.h) testY = box.pos.y + box.h; float distX = circle.pos.x - testX; float distY = circle.pos.y - testY; float distance = sqrt(distX * distX + distY * distY); return distance <= circle.radius; } }
```

In this method, each **`if`** branch returns a boolean value, ensuring that the method always produces a result. This will prevent the compile error related to missing return values.

explain your logic for Box vs Circle

The logic for the "Box vs Circle" collision detection follows a specific approach to determine if a circle intersects with a rectangle (box). Here's a detailed explanation:

1. **Identify Box and Circle:**
   - First, we determine which object is the box and which is the circle.

   java Copy code

   ```
   PhysicsObject box = isCircle ? other : this; PhysicsObject circle = isCircle ? this : other;
   ```

2. **Find the Closest Point on the Box to the Circle:**
   - We need to find the point on the rectangle (box) that is closest to the center of the circle. This involves clamping the circle's center coordinates to the edges of the rectangle.
   - We start by assuming the closest point on the box is the circle's center:

     java Copy code

```
float testX = circle.pos.x; float testY = circle.pos.y;
```

3. **Clamp the Circle's Center Coordinates to the Box Edges:**
   - If the circle's center is to the left of the box, the closest point on the box is the left edge:

     java Copy code

     ```
     if (circle.pos.x < box.pos.x) testX = box.pos.x;
     ```

   - If the circle's center is to the right of the box, the closest point on the box is the right edge:

     java Copy code

     ```
     else if (circle.pos.x > box.pos.x + box.w) testX = box.pos.x + box.w;
     ```

   - Similarly, for the vertical position, if the circle's center is above the box, the closest point on the box is the top edge:

     java Copy code

     ```
     if (circle.pos.y < box.pos.y) testY = box.pos.y;
     ```

   - If the circle's center is below the box, the closest point on the box is the bottom edge:

     java Copy code

     ```
     else if (circle.pos.y > box.pos.y + box.h) testY = box.pos.y + box.h;
     ```

4. **Calculate the Distance Between the Circle's Center and the Closest Point:**
   - With the coordinates of the closest point determined, calculate the distance from the circle's center to this point:

     java Copy code

     ```
     float distX = circle.pos.x - testX; float distY = circle.pos.y - testY; float
     distance = sqrt(distX * distX + distY * distY);
     ```

5. **Check if the Distance is Less Than or Equal to the Circle's Radius:**
   - If the distance is less than or equal to the radius of the circle, then there is a collision (the circle intersects or touches the box):

     java Copy code

     ```
     return distance <= circle.radius;
     ```

By following this method, we ensure accurate collision detection between a rectangle and a circle. The key idea is to find the closest point on the rectangle to the circle's center and then check if the distance from this point to the circle's center is less than or equal to the circle's radius. If it is, a collision has occurred.

should you also use the edge of the circle (radius + circle.pos.x and radius + circle.pos.y)

In the context of the box vs. circle collision detection, the edge of the circle is implicitly considered when calculating the distance from the circle's center to the closest point on the box. Let's break down why we don't need to directly use **`radius + circle.pos.x`** and **`radius + circle.pos.y`** in this specific method:

Explanation of Box vs. Circle Collision Detection

1. **Find the Closest Point on the Box to the Circle's Center:**
   - We find the closest point on the box to the circle's center by clamping the circle's center coordinates to the edges of the box.
2. **Calculate the Distance from the Circle's Center to this Closest Point:**
   - We calculate the distance from the circle's center to this closest point.
3. **Compare this Distance to the Circle's Radius:**
   - We check if this distance is less than or equal to the circle's radius to determine if there is a collision.

By following these steps, the method effectively checks if any part of the circle intersects with the box. Here's a detailed example:

Detailed Example

Consider the following example where a circle and a box are close to each other:

- **Circle:** Center at (cx, cy) with radius `r`
- **Box:** Top-left corner at (bx, by) with width `w` and height `h`

Steps in the Code

1. **Initialization:**
   - Determine which object is the circle and which is the box.

```java
PhysicsObject box = isCircle ? other : this; PhysicsObject circle = isCircle ? this : other;
```

2. **Closest Point on the Box:**
   - Start by assuming the closest point on the box is the circle's center.

```java
float testX = circle.pos.x; float testY = circle.pos.y;
```

   - Clamp the circle's center coordinates to the edges of the box.

```java
if (circle.pos.x < box.pos.x) testX = box.pos.x; else if (circle.pos.x > box.pos.x + box.w) testX = box.pos.x + box.w; if (circle.pos.y < box.pos.y) testY = box.pos.y; else if (circle.pos.y > box.pos.y + box.h) testY = box.pos.y + box.h;
```

3. **Distance Calculation:**
   - Calculate the distance from the circle's center to this closest point.

```java
float distX = circle.pos.x - testX; float distY = circle.pos.y - testY; float distance = sqrt(distX * distX + distY * distY);
```

4. **Collision Check:**
   - Check if this distance is less than or equal to the circle's radius.

java Copy code

```
return distance <= circle.radius;
```

Conclusion

This method inherently considers the radius of the circle in the distance calculation step. The distance from the circle's center to the closest point on the box is compared to the circle's radius to determine if there is a collision. Therefore, there's no need to directly use **`radius + circle.pos.x`** and **`radius + circle.pos.y`** in the collision detection logic.

Here's the final method for clarity:

java Copy code

```
boolean checkCollision(PhysicsObject other) { if (!isCircle && !other.isCircle) { // Box vs Box return (pos.x < other.pos.x + other.w && pos.x + w > other.pos.x && pos.y < other.pos.y + other.h && pos.y + h > other.pos.y); } else if (isCircle && other.isCircle) { // Circle vs Circle float distance = PVector.dist(pos, other.pos); return distance < radius + other.radius; } else { // Box vs Circle PhysicsObject box = isCircle ? other : this; PhysicsObject circle = isCircle ? this : other; float testX = circle.pos.x; float testY = circle.pos.y; if (circle.pos.x < box.pos.x) testX = box.pos.x; else if (circle.pos.x > box.pos.x + box.w) testX = box.pos.x + box.w; if (circle.pos.y < box.pos.y) testY = box.pos.y; else if (circle.pos.y > box.pos.y + box.h) testY = box.pos.y + box.h; float distX = circle.pos.x - testX; float distY = circle.pos.y - testY; float distance = sqrt(distX * distX + distY * distY); return distance <= circle.radius; } }
```

```
if (circle.pos.y < box.pos.y - box.h / 2) {
        closestY = box.pos.y - box.h / 2;
      } else if (circle.pos.y > box.pos.y + box.h / 2) {
        closestY = box.pos.y + box.h / 2;
      }
```

why do the if and else if statements does the same thing?

The **`if`** and **`else if`** statements in your code are not doing the same thing; they are checking for different conditions and clamping the circle's center to different edges of the box. Here's a detailed breakdown of what each statement does:

java Copy code

```
if (circle.pos.y < box.pos.y - box.h / 2) { closestY = box.pos.y - box.h / 2; } else if (circle.pos.y > box.pos.y + box.h / 2) { closestY = box.pos.y + box.h / 2; }
```

Explanation

1. **Clamping the Circle's Center to the Box Edges:**
   - The purpose of these statements is to find the closest point on the box to the circle's center along the y-axis. This ensures that the point used for distance calculation is correctly positioned on the edge of the box, either at the top, bottom, or within the box's height.
2. **Detailed Conditions:**
   - **`if (circle.pos.y < box.pos.y - box.h / 2)`**:
     - This condition checks if the circle's center (**`circle.pos.y`**) is above the top edge of the box (**`box.pos.y - box.h / 2`**). If true, it means the closest point on the box to the

circle's center is the top edge. Hence, **`closestY`** is set to the y-coordinate of the top edge of the box.

java Copy code

```
closestY = box.pos.y - box.h / 2;
```

- **`else if (circle.pos.y > box.pos.y + box.h / 2)`**:
  - This condition checks if the circle's center (**`circle.pos.y`**) is below the bottom edge of the box (**`box.pos.y + box.h / 2`**). If true, it means the closest point on the box to the circle's center is the bottom edge. Hence, **`closestY`** is set to the y-coordinate of the bottom edge of the box.

java Copy code

```
closestY = box.pos.y + box.h / 2;
```

Complete Example with X and Y Clamping

To provide more clarity, here's how the complete clamping logic looks for both x and y coordinates:

java Copy code

```
boolean checkCollision(PhysicsObject other) { if (!isCircle && !other.isCircle) { // Box vs Box return (pos.x < other.pos.x + other.w && pos.x + w > other.pos.x && pos.y < other.pos.y + other.h && pos.y + h > other.pos.y); } else if (isCircle && other.isCircle) { // Circle vs Circle float distance = PVector.dist(pos, other.pos); return distance < radius + other.radius; } else { // Box vs Circle PhysicsObject box = isCircle ? other : this; PhysicsObject circle = isCircle ? this : other; float closestX = circle.pos.x; float closestY = circle.pos.y; // Clamping x coordinate if (circle.pos.x < box.pos.x - box.w / 2) { closestX = box.pos.x - box.w / 2; } else if (circle.pos.x > box.pos.x + box.w / 2) { closestX = box.pos.x + box.w / 2; } // Clamping y coordinate if (circle.pos.y < box.pos.y - box.h / 2) { closestY = box.pos.y - box.h / 2; } else if (circle.pos.y > box.pos.y + box.h / 2) { closestY = box.pos.y + box.h / 2; } // Calculate distance float distX = circle.pos.x - closestX; float distY = circle.pos.y - closestY; float distance = sqrt(distX * distX + distY * distY); // Check for collision return distance <= circle.radius; } }
```

Summary

- The **`if`** condition ensures that if the circle's center is above the top edge of the box, it sets **`closestY`** to the top edge.
- The **`else if`** condition ensures that if the circle's center is below the bottom edge of the box, it sets **`closestY`** to the bottom edge.
- These conditions ensure that the distance calculation uses the closest point on the box to the circle's center, which is necessary for accurate collision detection.

after colliding, the objects are being pushed to the wrong direction.

When objects collide and are pushed in the wrong direction, it often means there is an issue with how the response to the collision is being calculated. This can involve resolving the collision by moving the objects apart and adjusting their velocities to reflect the collision.

Here's a breakdown of the steps you should take to ensure objects are pushed in the correct direction after a collision:

1. **Calculate the collision normal**: This is the vector that points from one object's center to the other object's center.
2. **Resolve the collision**: Move the objects so they no longer overlap.
3. **Adjust velocities**: Reflect the velocities of the objects based on the collision normal to simulate a realistic bounce.

Below is an example of how you might implement this in your code. This assumes you have a method for handling collisions and resolving them:

java Copy code

```
void resolveCollision(PhysicsObject a, PhysicsObject b) { // Calculate the normal of
the collision PVector normal = PVector.sub(b.pos, a.pos); normal.normalize(); //
Calculate the penetration depth float penetrationDepth = (a.radius + b.radius) -
PVector.dist(a.pos, b.pos); // Resolve the collision by moving the objects apart PVector
correction = normal.copy().mult(penetrationDepth / 2); a.pos.sub(correction);
b.pos.add(correction); // Calculate relative velocity PVector relativeVelocity =
PVector.sub(b.velocity, a.velocity); // Calculate velocity along the normal float
velocityAlongNormal = PVector.dot(relativeVelocity, normal); // Do not resolve if
velocities are separating if (velocityAlongNormal > 0) { return; } // Calculate
restitution (elasticity) float restitution = 0.8; // for example, you can adjust this
value // Calculate impulse scalar float impulseScalar = -(1 + restitution) *
velocityAlongNormal; impulseScalar /= a.invMass + b.invMass; // Apply impulse PVector
impulse = normal.copy().mult(impulseScalar);
a.velocity.sub(impulse.copy().mult(a.invMass));
b.velocity.add(impulse.copy().mult(b.invMass)); } // Example collision check and resolve
in your main loop void checkAndResolveCollisions() { for (int i = 0; i <
engine.objects.size(); i++) { for (int j = i + 1; j < engine.objects.size(); j++) {
PhysicsObject objA = engine.objects.get(i); PhysicsObject objB = engine.objects.get(j);
if (objA.checkCollision(objB)) { resolveCollision(objA, objB); } } } }
```

Explanation

1. **Calculate the normal of the collision**:
   - `PVector normal = PVector.sub(b.pos, a.pos);`
   - This vector points from object `a` to object `b`.
2. **Resolve the collision**:
   - Calculate the penetration depth to determine how much the objects overlap.
   - Move the objects apart along the collision normal to eliminate overlap.
3. **Adjust velocities**:
   - Calculate the relative velocity between the objects.
   - Determine the velocity along the collision normal.
   - Calculate the impulse based on the collision normal and apply it to adjust the velocities of the objects.

Important Points

- **Mass and Inverse Mass**: The `invMass` properties represent the inverse of the mass. For a stationary object, set `invMass` to 0. For dynamic objects, set it to `1/mass`.
- **Restitution**: This coefficient represents the elasticity of the collision. A value of 1 means a perfectly elastic collision, while 0 means perfectly inelastic.
- **Impulse Application**: The impulse is applied to both objects, scaled by their inverse masses.

This approach ensures that objects are correctly separated and their velocities are updated realistically after a collision. Adjust the restitution and mass values to fit your specific simulation needs.

This is too difficult to implement. instead, use the library Fisica / JBox2D physics engine from Processing. you need class Bird extends FCircle, class Brick extends FBox, and class Pig extends FCircle. give me some starter code for these

Certainly! Using the Fisica library, which is a wrapper for the JBox2D physics engine in Processing, simplifies physics simulations. Here's some starter code to help you get started with the **`Bird`**, **`Brick`**, and **`Pig`** classes, and setting up the scene.

Setting Up Fisica in Processing

First, make sure you have the Fisica library installed in your Processing environment. You can do this via the Library Manager in Processing.

Starter Code

java Copy code

```
import fisica.*; FWorld world; Bird bird; Brick brick; Pig pig; void setup() {
size(800, 600); Fisica.init(this); world = new FWorld(); world.setGravity(0, 800); //
Create the bird bird = new Bird(200, 300, 20); bird.setFillColor(color(255, 0, 0));
world.add(bird); // Create a brick brick = new Brick(400, 500, 100, 20);
brick.setFillColor(color(100, 100, 100)); world.add(brick); // Create a pig pig = new
Pig(600, 300, 30); pig.setFillColor(color(0, 255, 0)); world.add(pig); // Add ground
FBox ground = new FBox(width, 10); ground.setStatic(true); ground.setPosition(width / 2,
height - 5); ground.setFillColor(color(50, 50, 50)); world.add(ground); } void draw() {
background(255); world.step(); world.draw(); } // Bird class class Bird extends FCircle
{ Bird(float x, float y, float radius) { super(radius); setPosition(x, y); } } // Brick
class class Brick extends FBox { Brick(float x, float y, float width, float height) {
super(width, height); setPosition(x, y); } } // Pig class class Pig extends FCircle {
Pig(float x, float y, float radius) { super(radius); setPosition(x, y); } }
```

Explanation

1. **Setup Fisica and the World:**
   - **`Fisica.init(this);`** initializes the Fisica library.
   - **`world = new FWorld();`** creates a new physics world.
   - **`world.setGravity(0, 800);`** sets the gravity in the world.
2. **Creating Objects:**
   - **Bird:** An instance of **`FCircle`** with specific position and radius.
   - **Brick:** An instance of **`FBox`** with specific position, width, and height.
   - **Pig:** Another instance of **`FCircle`** with specific position and radius.
   - These objects are added to the world using **`world.add(object);`**.
3. **Ground:**
   - A static **`FBox`** to act as the ground, added to the world to prevent objects from falling infinitely.
4. **Draw Loop:**
   - **`world.step();`** advances the physics simulation.
   - **`world.draw();`** renders the objects.

Additional Features

To interact with the objects, you can add mousePressed and keyPressed functions to manipulate the bird, bricks, and pigs:

java Copy code

```
void mousePressed() { // Launch the bird on mouse press bird.setVelocity(500, -500); // Example values, adjust as needed } void keyPressed() { // Reset the bird's position on key press if (key == 'r') { bird.setPosition(200, 300); bird.setVelocity(0, 0); } }
```

Adding More Functionality

You can expand this starter code by:

- Adding more bricks and pigs.
- Implementing collision detection and response specific to your game mechanics.
- Creating different levels with varying layouts of bricks and pigs.
- Adding a slingshot mechanism to launch the bird.

This should provide a solid foundation to start building your physics-based game using Fisica and Processing.

bird.setDensity(0.5);
bird.setPosition(startPos.x, startPos.y)
NullPointerException

The **`NullPointerException`** indicates that either **`startPos`** or one of its components is not initialized before being used. To prevent this, ensure that all variables are properly initialized before usage. Here's an updated version of the code with proper initialization and the handling of the bird's position and density:

Complete Example with Proper Initialization

java Copy code

```
import fisica.*; FWorld world; Bird bird; Brick brick; Pig pig; PVector startPos; void setup() { size(800, 600); Fisica.init(this); world = new FWorld(); world.setGravity(0, 800); // Initialize start position for the bird startPos = new PVector(200, 300); // Create the bird bird = new Bird(startPos.x, startPos.y, 20); bird.setDensity(0.5); bird.setFillColor(color(255, 0, 0)); world.add(bird); // Create a brick brick = new Brick(400, 500, 100, 20); brick.setFillColor(color(100, 100, 100)); world.add(brick); // Create a pig pig = new Pig(600, 300, 30); pig.setFillColor(color(0, 255, 0)); world.add(pig); // Add ground FBox ground = new FBox(width, 10); ground.setStatic(true); ground.setPosition(width / 2, height - 5); ground.setFillColor(color(50, 50, 50)); world.add(ground); } void draw() { background(255); world.step(); world.draw(); } void mousePressed() { // Launch the bird on mouse press bird.setVelocity(500, -500); // Example values, adjust as needed } void keyPressed() { // Reset the bird's position on key press if (key == 'r') { bird.setPosition(startPos.x, startPos.y); bird.setVelocity(0, 0); } } // Bird class class Bird extends FCircle { Bird(float x, float y, float radius) { super(radius); setPosition(x, y); } } // Brick class class Brick extends FBox { Brick(float x, float y, float width, float height) { super(width, height); setPosition(x, y); } } // Pig class class Pig extends FCircle { Pig(float x, float y, float radius) { super(radius); setPosition(x, y); } }
```

Explanation

1. **Initialization of `startPos`:**
   - **`startPos = new PVector(200, 300);`** initializes the starting position of the bird.
2. **Bird Initialization:**

- The bird's density is set using `bird.setDensity(0.5);` after the bird is created and its position is set to `startPos`.

3. **Resetting Bird Position:**
   - In the `keyPressed` function, pressing 'r' resets the bird's position to `startPos` and sets its velocity to zero.

Key Points

- Ensure `startPos` is initialized before it is used to set the bird's position.
- The bird's density is set immediately after its creation.
- `mousePressed` and `keyPressed` functions provide basic interaction by launching and resetting the bird.

This code should work without the `NullPointerException` and give you a basic setup to start building your game using Fisica and Processing.

```
import fisica.*;

int screen = 0; // 0--Main Menu, 1--Level 1, 2--Level 2
boolean isPaused = false;
Button[] buttons = new Button[5];
FWorld world;
FCircle bird;
int birdDiameter = 30;
Catapult catapult;
PVector startPos = catapult.getPosition();
FBox catapultArm;
ArrayList<FBox> bricks = new ArrayList<FBox>();
boolean dragging = false;
PVector dragStart, dragEnd;

void setup() {
  size(1600, 900);
  Fisica.init(this);

  buttons[0] = new Button(300, 300, 50, "SQUARE_BUTTON", "1",
"none"); //300, 300
  buttons[1] = new Button(400, 300, 50, "SQUARE_BUTTON", "2",
"none"); //400, 300
  buttons[2] = new Button(1300, 100, 50, "CIRCULAR_BUTTON",
"none", "pause"); //1300, 100
  buttons[3] = new Button(1400, 100, 50, "CIRCULAR_BUTTON",
"none", "restart"); //1400, 100
  buttons[4] = new Button(1500, 100, 50, "CIRCULAR_BUTTON",
"none", "mainMenu"); //1500, 100

  world = new FWorld();
  world.setGravity(0, 500);
  world.setGrabbable(false);

  FBox ground = new FBox(width, 20);
  ground.setStatic(true);
  ground.setPosition(width / 2, height - 10);
  world.add(ground);

  catapult = new Catapult(150, height - 120);
  catapult.addToWorld(world);
```

```
  bird = new FCircle(birdDiameter);
  bird.setDensity(0.5);
  bird.setPosition(startPos.x, startPos.y - 10);
  bird.setFill(255, 0, 0);
  bird.setStatic(true);
  world.add(bird);

  for (int i = 0; i < 5; i++) {
    Brick brick = new Brick(60, 30);
    brick.setPosition(400 + i * 70, height - 150);
    world.add(brick);
    bricks.add(brick);
  }
}


void draw() {
  switch(screen) {
    case 0:
      displayMainMenu();
      break;
    case 1:
      displayLevel1();
      break;
    case 2:
      displayLevel2();
      break;
  }
}

void displayMainMenu() {
  background(255);
  buttons[0].draw();
  buttons[1].draw();
}

void displayLevel1() {
  buttons[2].draw();
  buttons[3].draw();
  buttons[4].draw();

  background(255);
  world.step();
  world.draw();

  if (dragging) {
    strokeWeight(5);
    line(startPos.x + 20, startPos.y, mouseX + 15, mouseY);
    line(startPos.x - 20, startPos.y, mouseX - 15, mouseY);
    bird.setPosition(mouseX, mouseY);
  }
}

void displayLevel2() {
  background(255);
  buttons[2].draw();
  buttons[3].draw();
  buttons[4].draw();
  // Add code to display and update level 2 elements
}
```

```
void mousePressed() {
  for (Button button : buttons) {
    if (button.try_click(mouseX, mouseY)) {
      button.buttonClicked = true;
    }
  }

  if (dist(mouseX, mouseY, bird.getX(), bird.getY()) < birdDiameter) {
    dragging = true;
    dragStart = new PVector(mouseX, mouseY);
  }
}

void mouseReleased() {
  for (int i = 0; i < buttons.length; i++) {
    if (buttons[i].buttonClicked) {
      buttons[i].buttonClicked = false;
      if (i == 0) {
        println("Level 1");
        screen = 1;
      } else if (i == 1) {
        println("Level 2");
        screen = 2;
      } else if (i == 2) {
        isPaused = !isPaused;
      } else if (i == 3) {
        //resetLevel();
      } else if (i == 4) {
        println("Main Menu");
        screen = 0;
        //resetLevel();
      }
    }
  }

  if (dragging) {
    bird.setStatic(false);
    dragEnd = new PVector(mouseX, mouseY);
    PVector force = PVector.sub(dragStart, dragEnd).mult(10);
    bird.addImpulse(force.x, force.y);
    dragging = false;
  }
}
java.lang.reflect.InvocationTargetException
```

The **`java.lang.reflect.InvocationTargetException`** is a common error that occurs when there is an underlying problem in the code that causes a method to fail when invoked via reflection. This can often be due to a **`NullPointerException`** or other runtime exceptions in your Processing sketch.

In your provided code, the likely culprits for the error might be:

1. **`catapult.getPosition()`** returning **`null`**.
2. Errors in initializing or using the **`Button`** or **`Brick`** classes.

To debug and fix these issues, let's address a few potential problems and ensure proper initialization:

Updated Code with Error Handling and Initialization

java Copy code

```
import fisica.*; int screen = 0; // 0--Main Menu, 1--Level 1, 2--Level 2 boolean isPaused = false; Button[] buttons = new Button[5]; FWorld world; FCircle bird; int birdDiameter = 30; Catapult catapult; PVector startPos; FBox catapultArm; ArrayList<FBox> bricks = new ArrayList<FBox>(); boolean dragging = false; PVector dragStart, dragEnd; void setup() { size(1600, 900); Fisica.init(this); buttons[0] = new Button(300, 300, 50, "SQUARE_BUTTON", "1", "none"); //300, 300 buttons[1] = new Button(400, 300, 50, "SQUARE_BUTTON", "2", "none"); //400, 300 buttons[2] = new Button(1300, 100, 50, "CIRCULAR_BUTTON", "none", "pause"); //1300, 100 buttons[3] = new Button(1400, 100, 50, "CIRCULAR_BUTTON", "none", "restart"); //1400, 100 buttons[4] = new Button(1500, 100, 50, "CIRCULAR_BUTTON", "none", "mainMenu"); //1500, 100 world = new FWorld(); world.setGravity(0, 500); world.setGrabbable(false); FBox ground = new FBox(width, 20); ground.setStatic(true); ground.setPosition(width / 2, height - 10); world.add(ground); catapult = new Catapult(150, height - 120); catapult.addToWorld(world); startPos = catapult.getPosition(); bird = new FCircle(birdDiameter); bird.setDensity(0.5); bird.setPosition(startPos.x, startPos.y - 10); bird.setFill(255, 0, 0); bird.setStatic(true); world.add(bird); for (int i = 0; i < 5; i++) { Brick brick = new Brick(60, 30); brick.setPosition(400 + i * 70, height - 150); world.add(brick); bricks.add(brick); } } void draw() { switch(screen) { case 0: displayMainMenu(); break; case 1: displayLevel1(); break; case 2: displayLevel2(); break; } } void displayMainMenu() { background(255); for (Button button : buttons) { button.draw(); } } void displayLevel1() { background(255); for (Button button : buttons) { button.draw(); } if (!isPaused) { world.step(); } world.draw(); if (dragging) { strokeWeight(5); line(startPos.x + 20, startPos.y, mouseX + 15, mouseY); line(startPos.x - 20, startPos.y, mouseX - 15, mouseY); bird.setPosition(mouseX, mouseY); } } void displayLevel2() { background(255); for (Button button : buttons) { button.draw(); } // Add code to display and update level 2 elements } void mousePressed() { for (Button button : buttons) { if (button.try_click(mouseX, mouseY)) { button.buttonClicked = true; } } if (dist(mouseX, mouseY, bird.getX(), bird.getY()) < birdDiameter) { dragging = true; dragStart = new PVector(mouseX, mouseY); } } void mouseReleased() { for (int i = 0; i < buttons.length; i++) { if (buttons[i].buttonClicked) { buttons[i].buttonClicked = false; if (i == 0) { println("Level 1"); screen = 1; } else if (i == 1) { println("Level 2"); screen = 2; } else if (i == 2) { isPaused = !isPaused; } else if (i == 3) { //resetLevel(); } else if (i == 4) { println("Main Menu"); screen = 0; //resetLevel(); } } } if (dragging) { bird.setStatic(false); dragEnd = new PVector(mouseX, mouseY); PVector force = PVector.sub(dragStart, dragEnd).mult(10); bird.addImpulse(force.x, force.y); dragging = false; } } // Example Catapult class class Catapult { FBox base; Catapult(float x, float y) { base = new FBox(100, 20); base.setPosition(x, y); base.setStatic(true); } void addToWorld(FWorld world) { world.add(base); } PVector getPosition() { return new PVector(base.getX(), base.getY()); } } // Example Button class class Button { float x, y, size; String shape, label, action; boolean buttonClicked = false; Button(float x, float y, float size, String shape, String label, String action) { this.x = x; this.y = y; this.size = size; this.shape = shape; this.label = label; this.action = action; } void draw() { if (shape.equals("SQUARE_BUTTON")) { rect(x - size / 2, y - size / 2, size, size); } else if (shape.equals("CIRCULAR_BUTTON")) { ellipse(x, y, size, size); } fill(0); text(label, x, y); } boolean try_click(float mx, float my) { if (shape.equals("SQUARE_BUTTON")) { return mx > x - size / 2 && mx < x + size / 2 && my > y - size / 2 && my < y + size / 2; } else if (shape.equals("CIRCULAR_BUTTON")) { return dist(mx, my, x, y) < size / 2; } return false; } } // Example Brick class class Brick extends FBox { Brick(float width, float height) { super(width, height); } }
```

Key Changes and Additions:

1. **Initialization of `startPos`:**
   - Moved the initialization of **`startPos`** after creating the **`catapult`**.
2. **Added `Catapult`, `Button`, and `Brick` Classes:**
   - Provided basic implementations for **`Catapult`**, **`Button`**, and **`Brick`**.
3. **Fixed Button Drawing and Click Logic:**
   - Ensured buttons are drawn and can detect clicks correctly.
4. **Handling Pause:**
   - Added **`isPaused`** check in **`displayLevel1`** to pause the physics simulation.

5. **Dragging Logic:**
   - Corrected logic to set the bird's position while dragging and applying force on release.

This code should now properly initialize and handle the bird's position, as well as the button interactions and dragging mechanics, without causing the `NullPointerException` or `InvocationTargetException`.

```
import fisica.*;

int screen = 0; // 0--Main Menu, 1--Level 1, 2--Level 2
boolean isPaused = false;
Button[] buttons = new Button[5];
FWorld world;
FCircle bird;
int birdDiameter = 30;
Catapult catapult;
FBox catapultArm;
Brick[] bricks;
Pig[] pigs;
boolean dragging = false;
PVector dragStart, dragEnd;
int score = 0;
int highScoreLevel1 = 0;
int highScoreLevel2 = 0;

void setup() {
  size(1600, 900);
  Fisica.init(this);

  buttons[0] = new Button(300, 300, 50, "SQUARE_BUTTON", "1",
"none"); //300, 300
  buttons[1] = new Button(400, 300, 50, "SQUARE_BUTTON", "2",
"none"); //400, 300
  buttons[2] = new Button(1300, 100, 50, "CIRCULAR_BUTTON",
"none", "pause"); //1300, 100
  buttons[3] = new Button(1400, 100, 50, "CIRCULAR_BUTTON",
"none", "restart"); //1400, 100
  buttons[4] = new Button(1500, 100, 50, "CIRCULAR_BUTTON",
"none", "mainMenu"); //1500, 100

  world = new FWorld();
  world.setGravity(0, 500);
  world.setGrabbable(false);

  FBox ground = new FBox(width, 20);
  ground.setStatic(true);
  ground.setPosition(width / 2, height);
  world.add(ground);

  catapult = new Catapult(150, height - 120);
  catapult.addToWorld(world);

  PVector startPos = catapult.getPosition();
  bird = new FCircle(birdDiameter);
  bird.setDensity(0.5);
  bird.setPosition(startPos.x, startPos.y - 6);
  bird.setFill(255, 0, 0);
  bird.setStatic(true);
```

```
  world.add(bird);

  bricks = new Brick[10];
  bricks[0] = new Brick(30, 120, "wood");
  bricks[0].setPosition(500, height - 75);
  bricks[1] = new Brick(30, 120, "wood");
  bricks[1].setPosition(650, height - 75);
  bricks[2] = new Brick(150, 30, "wood");
  bricks[2].setPosition(575, height - 145);
  bricks[3] = new Brick(30, 60, "glass");
  bricks[3].setPosition(540, height - 190);
  bricks[4] = new Brick(30, 60, "glass");
  bricks[4].setPosition(610, height - 190);
  bricks[5] = new Brick(150, 30, "wood");
  bricks[5].setPosition(575, height - 225);
  bricks[6] = new Brick(30, 120, "wood");
  bricks[6].setPosition(540, height - 305);
  bricks[7] = new Brick(30, 120, "wood");
  bricks[7].setPosition(610, height - 305);
  bricks[8] = new Brick(150, 30, "wood");
  bricks[8].setPosition(575, height - 375);
  bricks[9] = new Brick(30, 30, "wood");
  bricks[9].setPosition(575, height - 410);
  for (Brick brick : bricks) {
    world.add(brick);
  }

  pigs = new Pig[3];
  pigs[0] = new Pig(50);
  pigs[0].setPosition(575, height - 300);
  pigs[1] = new Pig(50);
  pigs[1].setPosition(575, height - 190);
  pigs[2] = new Pig(50);
  pigs[2].setPosition(575, height - 100);
  for (Pig pig : pigs) {
    world.add(pig);
  }
}


void draw() {
  switch(screen) {
    case 0:
      displayMainMenu();
      break;
    case 1:
      displayLevel1();
      break;
    case 2:
      displayLevel2();
      break;
  }
}

void displayMainMenu() {
  background(255);
  buttons[0].draw();
  buttons[1].draw();

  fill(0);
  textSize(32);
```

```
  text("High Score Level 1: " + highScoreLevel1, 20, 100);
  text("High Score Level 2: " + highScoreLevel2, 20, 150);
}

void displayLevel1() {
  background(255);
  buttons[2].draw();
  buttons[3].draw();
  buttons[4].draw();

  world.step();
  world.draw();

  fill(0);
  textSize(32);
  text("Score: " + score, 20, 40);

  if (dragging) {
    strokeWeight(5);
    PVector startPos = catapult.getPosition();
    line(startPos.x + 20, startPos.y, mouseX + 15, mouseY);
    line(startPos.x - 20, startPos.y, mouseX - 15, mouseY);
    bird.setPosition(mouseX, mouseY);
  }

  for (Pig pig : pigs) {
    if (pig.alive) {
      score += pig.checkCollision(ground, bird, world);
    }
  }
}

void displayLevel2() {
  background(255);
  buttons[2].draw();
  buttons[3].draw();
  buttons[4].draw();

  world.step();
  world.draw();

  fill(0);
  textSize(32);
  text("Score: " + score, 20, 40);

  if (dragging) {
    strokeWeight(5);
    PVector startPos = catapult.getPosition();
    line(startPos.x + 20, startPos.y, mouseX + 15, mouseY);
    line(startPos.x - 20, startPos.y, mouseX - 15, mouseY);
    bird.setPosition(mouseX, mouseY);
  }

  for (Pig pig : pigs) {
    if (pig.alive) {
      score += pig.checkCollision(ground, bird, world);
    }
  }
}

void mousePressed() {
```

```
  for (Button button : buttons) {
    if (button.try_click(mouseX, mouseY)) {
      button.buttonClicked = true;
    }
  }

  if (dist(mouseX, mouseY, bird.getX(), bird.getY()) < birdDiameter) {
    dragging = true;
    dragStart = new PVector(mouseX, mouseY);
  }
}

void mouseReleased() {
  for (int i = 0; i < buttons.length; i++) {
    if (buttons[i].buttonClicked) {
      buttons[i].buttonClicked = false;
      if (i == 0) {
        println("Level 1");
        // Store high score before switching levels
        if (score > highScoreLevel1) {
          highScoreLevel1 = score;
        }
        score = 0;
        screen = 1;
      } else if (i == 1) {
        println("Level 2");
        // Store high score before switching levels
        if (score > highScoreLevel2) {
          highScoreLevel2 = score;
        }
        score = 0;
        screen = 2;
      } else if (i == 2) {
        isPaused = !isPaused;
      } else if (i == 3) {
        //resetLevel();
      } else if (i == 4) {
        println("Main Menu");
        // Store high score before switching to main menu
        if (screen == 1 && score > highScoreLevel1) {
          highScoreLevel1 = score;
        } else if (screen == 2 && score > highScoreLevel2) {
          highScoreLevel2 = score;
        }
        score = 0;
        screen = 0;
        //resetLevel();
      }
    }
  }

  if (dragging) {
    bird.setStatic(false);
    dragEnd = new PVector(mouseX, mouseY);
    PVector force = PVector.sub(dragStart, dragEnd).mult(10);
    bird.addImpulse(force.x, force.y);
    dragging = false;
  }
}
```

score += pig.checkCollision(ground, bird, world): The variable "ground" does not exist. only show me the code you changed

Here are the changes made to address the **`ground`** variable not existing within the **`checkCollision`** method call:

1. **Change the `ground` reference in `checkCollision` calls to use the correct object reference.**
2. **Ensure that `Pig` class includes `checkCollision` logic which correctly increments the score.**

Changes:

In `setup()` function:

java Copy code

```
FBox ground; // Declare ground at the class level void setup() { size(1600, 900); Fisica.init(this); // Initialize buttons and world... ground = new FBox(width, 20); // Initialize ground here ground.setStatic(true); ground.setPosition(width / 2, height); world.add(ground); // Initialize catapult, bird, bricks, and pigs... }
```

In `displayLevel1()` and `displayLevel2()` functions:

java Copy code

```
for (Pig pig : pigs) { if (pig.alive) { score += pig.checkCollision(world, bird); } }
```

In the `Pig` class (assuming you have such a class), add a method for collision detection and scoring:

java Copy code

```
class Pig extends FCircle { boolean alive = true; Pig(float radius) { super(radius); setFill(0, 255, 0); } int checkCollision(FWorld world, FCircle bird) { if (this.isTouching(bird)) { world.remove(this); alive = false; return 100; // Example score for hitting a pig } return 0; } }
```

Summary of Changes:

1. Declare **`ground`** at the class level.
2. Initialize **`ground`** in the **`setup()`** function.
3. Update **`checkCollision`** calls in **`displayLevel1()`** and **`displayLevel2()`** to correctly reference the **`world`** and **`bird`**.
4. Add a **`checkCollision`** method in the **`Pig`** class that handles collision detection and scoring.

This ensures the **`checkCollision`** method works without errors, and scores are correctly updated when pigs are hit by the bird.