Csci 1933 Project #2: A Bouncing Ball Screen Saver

Due date: 10/8/2017, before midnight

100 points.

Summary. For Project #2, we will write a screen saver application that animates balls bouncing and colliding inside a box. This project will require you to use inheritance to extend the Circle class as well as an AnimationFrame class we have written for you, and will give you practice applying concepts like composition, arrays, and handling user input.

Part I. Defining and Implementing a Ball class (40 points)

The Ball class <u>should extend the Circle class</u> from the last project (or download our version), and should maintain state information and implement the physics of each bouncing ball. The state information should include the size, color, position and velocity (speed in both X and Y dimensions) of the ball and all of the methods necessary to interact with a ball such as checking if two balls have collided and functionality for updating state after a collision.

Methods inherited from Circle:

- Name: setColor; input: color of the shape (type Color); output: none
- Name: setPos; input: x, y position of the center (both doubles) output: none
- Name: setRadius; input: radius (double) output: none
- Name: getColor; input: none; output: color of the shape (type Color);
- Name: getXPos; input: none; output: x position of the center (double)
- Name: getYPos; input: none; output: y position of the center (double)
- Name: getRadius; input: none; output: radius (double)
- Name: calculatePerimeter; input: none; output: perimeter of the circle (type double)
- Name: calculateArea; input: none; output: area of the circle (double)

New methods:

- Name: setSpeedX; input: speed of the ball's movement along the horizontal axis (type double); output: none
- Name: setSpeedY; input: speed of the ball's movement along the vertical axis (type double); output: none
- Name: getSpeedX; input: none; output: speed of the ball's movement along the horizontal axis (type double)
- Name: getSpeedY; input: none; output: speed of the ball's movement along the vertical axis (type double)
- Name: updatePosition; input: number of time units passed (type double); output: none; This method should update the ball's position based on its current velocity and the time elapsed.

- Name: intersect; input: another Ball object to check if they have collided (type Ball); output: Boolean; This method should check if the passed Ball object overlaps with the current Ball, which will be useful for detecting collisions among Balls.

- Name: collide; input: another Ball object with which the current Ball should collide (type Ball); output: none; This method should confirm that the balls have collided and implement the physics of two balls colliding— the state (i.e. X and Y speeds) of each ball should be updated accordingly at the end of this method.

- Name: Ball (constructor) ; input: x position (double), y position (double), radius (double); color (Color) output : object of type Ball

Data members: any that are necessary to maintain the object's state and implement the methods above.

Part II. Defining and Implementing a BallScreenSaver class (40 points)

You will need to implement a second class that will actually support that animation of the bouncing balls. We have implemented an abstract base class, AnimationFrame, that should help you get started. This class has all of the core functionality necessary for creating an application frame and updating a window that can animate graphics. You don't need to worry about the details of how this class is implemented, but you will need to understand how to make it draw animation. The key pieces that enable classes derived from AnimationFrame to implement animation are the *action()* and *draw()* methods. First, an object of this type must be instantiated and the *start()* method called. Then, at a rate determined by the member variable, *fps* (frames per second), the *action()* method is periodically called and should perform the logic for updating the animation frame (e.g. update the positions of all bouncing balls). After each call to *action()*, the method *draw()* is called, and should be used to redraw any graphics inside the frame. As long as the frames per second rate is higher than ~15, the graphics in the window should look reasonably animated. Remember that most monitors won't refresh more than ~60 times per second anyway, so it will not help to update your graphics more often than that.

Your task is to define and implement a class called BallScreenSaver that extends our AnimationFrame class to display balls bouncing around inside a box. The methods you will need to implement/override are:

- action(): input: none; output: none; As described above, this method is called periodically (default rate 50 times/sec) and actually updates the states of any objects being animated.

- draw(): input: graphics object reference (type Graphics); output: none; As described above, this method actually draws the graphics for each frame after the action method is called to update state.

- processKeyEvent(): input: event object corresponding with a keyboard event (type KeyEvent); output: none; This method is automatically called whenever a key is pressed while the animation is running. You will need to override this method to support the requirements described below.

Hint: all of the above methods should override either existing or abstract methods in the class AnimationFrame, so the method signatures should be identical to the base class.

- BallScreenSaver (constructor): input: window frame width (int), window frame height (int), name of the application window (String), the number of bouncing balls to be animated (int) output : object of type BallScreenSaver

Data members: any that are necessary to implement the methods above.

<u>Requirements for your animation:</u>

Your BallScreenSaver class should support the following functionality:

- Variable number of balls as input (constructor) parameter (Hint: you can assume that all balls have the same radius and mass)
- Balls should be placed inside a rectangular box, with random initial velocities and random positions
- Balls should elastically bounce off walls and each other (for hints, see the "Physics of 2D collisions" section below)
- Keyboard control for object speed: The left and right arrows on the keyboard should be used to control the speed of all balls in the animation. When the right arrow is pressed, both the X and Y speed of all balls should be increased by 10% in their current direction. When the left arrow is pressed, both the X and Y speed of all objects should be decreased by 10%. (Hint: the processKeyEvent method will be very useful here!)
- Ball color upon collision: In each animation, all balls except one should be green. One ball should be red, and the red color should "spread" with each collision. Specifically, anytime a red ball collides with a green ball, the green ball should switch to red. For example, if you were to let your screen saver run forever, all balls will eventually be red.

Part III. Implementing a CollisionLogger class (20 points)

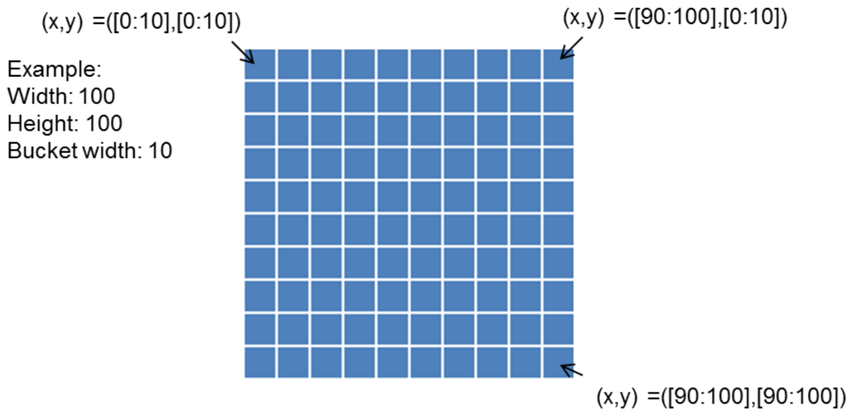
Finally, you will need to finish an implementation of a CollisionLogger class, which will be used to count the number of collisions that occurs in each "bucket" of a two-dimensional, discretized grid. We have provided a skeleton version of this class for you—you just need to add the appropriate data members and complete the implementation of the methods described below. **You are required** to use a two-dimensional array of integers as a data member of this class.

CollisionLogger (constructor): inputs: int screenWidth, int screenHeight, int bucketWidth inputs: the screen width (type int), the screen height (type int) and the bucketWidth (type int); output: an instance of the CollisionLogger class.

collide: inputs: two different Ball objects; output: none. This method should log a collision at the position of the two balls. You can assume the average X and average Y position of the two balls for the collision location.

getNormalizedHeatMap: inputs: none; output: a two-dimensional array of ints. This should return the current state of the collision log but rescaled such that the bucket with the largest number of collisions has value 255, and buckets without any collisions have a value of 0.

Below we provide an example of where different x,y coordinate pairs would be placed for a given CollisionLogger of the corresponding height, width, and bin size.



You should complete the implementation of the CollisionLogger class we've given you, and use it properly to log collisions in your screen saver. Notice that in our BouncingBall example class (described below), we demonstrate for you how to set up the CollisionLogger, and we have configured it such that a PNG image with the collision log heatmap is printed each time the "p" key is pressed. Your BallScreenSaver implementation should also include this functionality (feel free to reuse our example code from BouncingBall).

Hints:

An example screen saver class:

To show you a simple example of how you might use our AnimationFrame class, we have implemented a very simple screen saver class, BouncingBall, that has just one ball bouncing around inside a box. Our *action()* and *draw()* methods look like this:

```
public void action(){
      //This method is called once every frame to update the state of
      the BouncingBall.

      //update both X and Y positions
      ballX+=ballXSpeed/getFPS();
      ballY+=ballYSpeed/getFPS();

      //handle collisions with the border
      if ( ballX<BORDER || ballX+ballSize>getHeight()-BORDER ){
            ballXSpeed*=-1;
      }
      if ( ballY<BORDER || ballY+ballSize>getWidth()-BORDER ){
            ballYSpeed*=-1;
      }

}

public void draw(Graphics g){
      //This method is called once every frame to draw the Frame.

      //This is how you use the graphics object to draw
      g.setColor(Color.black);
      g.fillRect(0,0,getWidth(),getHeight());
      g.setColor(Color.white);
      g.drawRect(BORDER,BORDER,getWidth()-BORDER*2,getHeight()-
      BORDER*2);
      g.fillOval((int)ballX, (int)ballY, ballSize, ballSize);
}
```

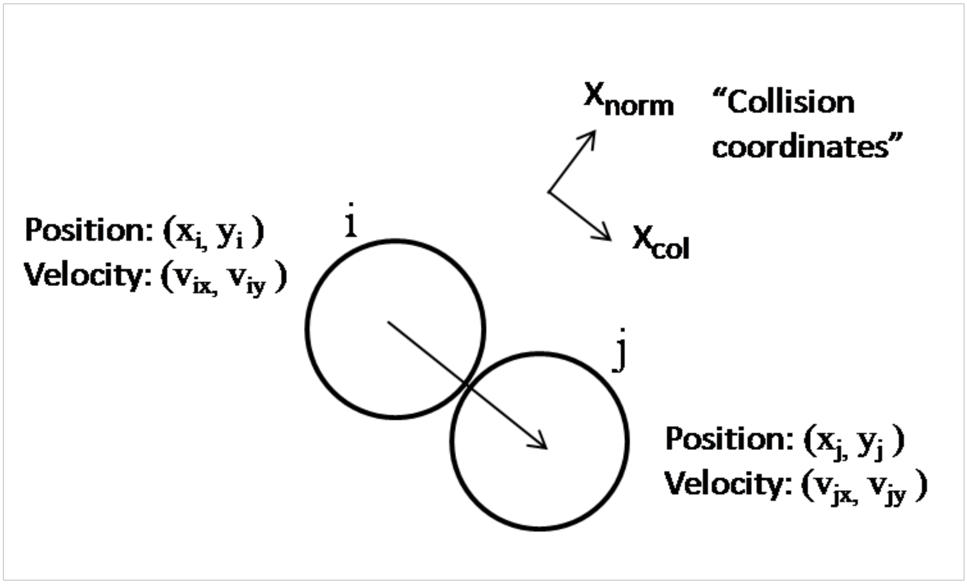
Your screen saver will be much more exciting, but notice how the methods described above are implemented to support the animation. Note that to actually use the BouncingBall class, we just write the following inside a main method:

```
BouncingBall bb= new BouncingBall(800,800,"Bouncing Ball");
bb.start();
```

A tutorial on the physics of 2D collisions:

Since this isn't a Physics class, we'll help you out with the physics of 2D collisions. For an in-depth discussion of the physics of 2D elastic collisions, see this: http://en.wikipedia.org/wiki/Elastic_collision . We'll give you a basic summary and the formulas you will need to update objects' velocities after a collision.

Consider two balls of equal radius and the given positions and velocities that are about to collide. The basic principle we rely on to compute the velocities after the collision is the conservation of momentum and conservation of kinetic energy (the collision is elastic). A "trick" which makes the calculation much easier is to translate both balls' velocities into a coordinate system aligned along the direction of the collision (X_{col} and X_{norm} vectors below). Since the ball's are of equal mass, the calculation becomes trivial in that coordinate system: the components of the velocity along the collision direction are simply flipped between the two objects, and the component perpendicular (X_{norm}) stays the same for both objects. The math for this is worked out in detail below.



First, let's compute unit vectors for the new coordinate system:

$$D = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

$$X_{col} = \left(\frac{x_i - x_j}{D}, \frac{y_i - y_j}{D} \right) \equiv (\Delta x, \Delta y)$$

$$X_{norm} = \left(-\frac{y_i - y_j}{D}, \frac{x_i - x_j}{D} \right) = (-\Delta y, \Delta x)$$

Now, to translate the velocities of both balls into this coordinate system, we need to do the following matrix multiplication, which simply projects each ball's velocity vector onto the two perpendicular directions:

$$V -> V^C :$$

$$V^C = \begin{pmatrix} X_{col} \\ X_{norm} \end{pmatrix} * V \equiv \begin{pmatrix} \Delta x & \Delta y \\ -\Delta y & \Delta x \end{pmatrix} * V$$

So, these are the actual velocities for balls *i* and *j* in the new coordinate system:

$$V_i^C = \begin{pmatrix} \Delta x & \Delta y \\ -\Delta y & \Delta x \end{pmatrix} \begin{pmatrix} v_{ix} \\ v_{iy} \end{pmatrix} = \begin{pmatrix} v_{ix}\Delta x + v_{iy}\Delta y \\ -v_{ix}\Delta y + v_{iy}\Delta x \end{pmatrix} \equiv \begin{pmatrix} v_{ix}{}^C \\ v_{iy}{}^C \end{pmatrix}$$

$$V_j^C = \begin{pmatrix} \Delta x & \Delta y \\ -\Delta y & \Delta x \end{pmatrix} \begin{pmatrix} v_{jx} \\ v_{jy} \end{pmatrix} = \begin{pmatrix} v_{jx}\Delta x + v_{jy}\Delta y \\ -v_{jx}\Delta y + v_{jy}\Delta x \end{pmatrix} \equiv \begin{pmatrix} v_{jx}{}^C \\ v_{jy}{}^C \end{pmatrix}$$

Now, we need to actually compute the new velocities for both balls in this new coordinate system. The second component (corresponding to the X_{norm} direction) stays constant for both balls, but the first component (corresponding to the X_{col} direction) flips:

$$V_i^C{}_{after} = \begin{pmatrix} v_{jx}{}^C \\ v_{iy}{}^C \end{pmatrix} = \begin{pmatrix} v_{jx}\Delta x + v_{jy}\Delta y \\ -v_{ix}\Delta y + v_{iy}\Delta x \end{pmatrix}$$

$$V_j^C{}_{after} = \begin{pmatrix} v_{ix}{}^C \\ v_{jy}{}^C \end{pmatrix} = \begin{pmatrix} v_{ix}\Delta x + v_{iy}\Delta y \\ -v_{jx}\Delta y + v_{jy}\Delta x \end{pmatrix}$$

Now, we just need to translate these velocities back into the original coordinate system of our animation frame, and we're done. To do this, we simply multiply by the inverse of the matrix from before, i.e.:

$$V^C -> V:$$

$$V_{i\,after} = \begin{pmatrix} \Delta x & \Delta y \\ -\Delta y & \Delta x \end{pmatrix}^{-1} V_i^C{}_{after} = \begin{pmatrix} \Delta x & -\Delta y \\ \Delta y & \Delta x \end{pmatrix} V_i^C{}_{after} = \begin{pmatrix} v_{jx}{}^C \Delta x - v_{iy}{}^C \Delta y \\ v_{jx}{}^C \Delta y + v_{iy}{}^C \Delta x \end{pmatrix}$$

$$V_{j\,after} = \begin{pmatrix} \Delta x & \Delta y \\ -\Delta y & \Delta x \end{pmatrix}^{-1} V_j^C{}_{after} = \begin{pmatrix} \Delta x & -\Delta y \\ \Delta y & \Delta x \end{pmatrix} V_j^C{}_{after} = \begin{pmatrix} v_{ix}{}^C \Delta x - v_{jy}{}^C \Delta y \\ v_{ix}{}^C \Delta y + v_{jy}{}^C \Delta x \end{pmatrix}$$

Submitting your finished assignment:

Once you've completed Project #2, create a zip file with all of your Java files (.java), including one that implements your main program, and submit it through Moodle.

Working with a partner:

As discussed in lecture, you may work with one partner to complete this assignment (max team size = 2). If you choose to work as a team, please only turn in one copy of your assignment. At the top of your class definition that implements your main program, include in the comments both of your names and student IDs. In doing so, you are attesting to the fact that both of you have contributed substantially to completion of the project and that both of you understand all code that has been implemented.

Some guidelines about how we will grade your project:

To give you a sense for the grade your assignment will receive, here are some examples of what will be required for each grade level:

<u>A-level assignment:</u>

- Properly defined Ball class with correctly implemented methods (passes unit test)
- Properly defined BallScreenSaver class with implementations of the action and draw methods
- Multiple balls and box and ball collisions detected, states updated properly
- Keyboard control updates ball speeds properly
- Ball colors change upon collisions with the red ball
- Properly implemented CollisionLogger class
- Correct use of the CollisionLogger class in your BallScreenSaver
- Good programming style (spacing and comments)

<u>B-level assignment:</u>

- Properly defined Ball class with correctly implemented methods (passes unit test)
- Properly defined BallScreenSaver class with implementations of the action and draw methods
- Partially working animation with at least one ball object
- Collision detection working, partially working state updates
- Properly implemented CollisionLogger class
- Partially working use of CollisionLogger inside BallScreenSaver

<u>C-level assignment:</u>

- Properly defined Ball class with correctly implemented methods (passes unit test)
- Partially defined BallScreenSaver class with implementations of the action and draw methods
- Partially defined CollisionLogger class