Introduction to the Assignment

In this lab, you will create a program to play the video game Asteroids, a classic Atari game introduced in 1979. In writing this program you will gain experience using inheritance. In particular, you will write some classes that extend others, as well as creating a superclass to capture similarities between some classes in your program.

Here are some snapshots from this program:

Here is the starting screen. The ship is in the center and pointing up. The asteroids start in random locations and move in straight line, but in a random direction.

The user can use the up arrow key to accelerate and the down arrow key to decelerate. The left and right arrow keys are used to rotate the spaceship.
<table>
<thead>
<tr>
<th>Image</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.jpg" alt="Image" /></td>
<td>The user can shoot with the space bar. Bullets will go in the direction that the space ship is pointing. If a bullet hits an asteroid, the asteroid will disappear.</td>
</tr>
<tr>
<td><img src="image2.jpg" alt="Image" /></td>
<td>This is the window after the user destroys all the asteroids.</td>
</tr>
<tr>
<td><img src="image3.jpg" alt="Image" /></td>
<td>Here is the window if an asteroid collides with the spaceship. The spaceship is no longer displayed, but the asteroids continue to move around.</td>
</tr>
</tbody>
</table>
The Program Design

This project will have 7 classes, 2 of which I provide to you in their entirety. These classes are:

- Game - a class that provides the framework for animation. I provide this class.
- AsteroidsGame - this class extends Game. You will put the functionality that is specific to Asteroids in this class.
- Ship - a class to manipulate the spaceship that the user controls
- Asteroid - a class to manipulate the asteroids flying around the display
- Bullet - a class to manipulate the bullets the ship shoots to destroy the asteroids.
- AsteroidsGameShape - a superclass of Ship, Asteroid and Bullet that captures the similarities between these shapes, how they are drawn, how they move, etc.
- TransformCalculator - a class that I provide to you that does the trigonometry necessary to determine where the various shapes are on the screen when they are rotated and moved.

You can download Game.java and TransformCalculator.java from the course website. These classes are completely written for you. **YOU SHOULD NOT CHANGE THEM!** The APIs for these classes are attached at the end of this document.

You will also be using some classes and interfaces that Java provides that you are not yet familiar with. These are described briefly below.

**KeyListener.** KeyListener is an interface that allows you to write methods that react to the user typing on a keyboard. Your AsteroidsGame class will implement KeyListener and define the keyPressed method to handle the key input the user uses to move and rotate the ship as well as shoot the bullets.

**Polygon.** Polygon is a class that defines a closed shape consisting of multiple straight lines joined at corners. Your ship and asteroids will have polygons for their shapes.

**Ellipse2D.Double.** Ellipse2D.Double is a class that allows you to create ellipses (ovals). Your bullets will have shapes that are instances of this class.

**Shape.** Shape is an interface that is implemented by Polygon and Ellipse2D.Double. In some cases, you will want to just work with shapes independently of whether they are polygons or ellipses.

**Point2D.Double.** Provides an x, y pair to identify a location in a coordinate system.

**Graphics2D.** This is a subclass of the Graphics class we have previously used that provides a few extra methods that are very useful for this lab:

```java
public void translate (x, y)
public void rotate (double theta)
public void draw (Shape s)
```

If you observe how Asteroids works, you will notice that the ship both moves and rotates. In order to draw a ship, you might think that you would need to calculate where the corners of the ship are to determine where they should appear on the screen. This would involve doing some trigonometry. (I heard that groan!) Fortunately, using the Graphics2D class, it is actually much simpler. Instead of rotating the shape, you essentially rotate the piece of "paper" you are drawing on, something like the diamond-shaped drawing at the left below. Then, when you reverse the rotation of the "paper", you end up with the shape on the right.
Moving the ship works in a similar way. You actually move the “paper” rather than the ship. The advantage of doing this is that you always draw your shapes the same way, independent of where they appear on the screen or how they are oriented. In particular, you should draw your shapes with their center at (0, 0). For example, if the shape is 20 pixels wide and 40 pixels tall, its 3 corners are (-10, 20), (0, -20), and (10, 20), like this:

```
To draw the ship in the middle of the screen and rotated 45 degrees to the left as in the rotated figure above, we would say:
```
```
g2D.translate (windowWidth/2, windowHeight/2);
g2D.rotate (-Math.PI/4);
g2D.draw (shipShape);
```

This code assumes that windowWidth and windowHeight are set to the dimensions of the window. The argument to rotate is given in radians. A complete rotation (360 degrees) is \(2\pi\) radians. shipShape is a Polygon that defines the 3 endpoints of the ship.

**Writing the Program**

Create a new Eclipse project called Asteroids. You should download Game.java and TransformCalculator.java from the course website and import them into your project. **YOU SHOULD NOT CHANGE THEM.**

**Step 1: Draw a spaceship**

Begin by creating a class AsteroidsGame that extends Game. In this lab, the drawing we will be doing is similar to the drawing that you did for the IdCard lab. In particular, you should define a main method that creates a JFrame and a paintComponent method to paint the various shapes on the display. So, begin by creating a blank window of an appropriate size with a black background. You should place the code to do this in the AsteroidsGame class that you create.

You should notice that you have a compiler error in you AsteroidsGame class, saying “The type AsteroidsGame must implement the inherited abstract method Game.nextFrame().” In the next step, there will be instructions about what should go in this method. For now, you
should declare the method, but keep its body empty. Run your program. You should see a window with a black background.

Next, create a Ship class. The Ship constructor should take parameters that identify the left coordinate of the ship and the top of the ship. These are the values that you will pass to the translate method of the Graphics2D class as shown above. The ship class should have an instance variable name “shape” that will hold a Polygon. In the Ship constructor, you should construct a Polygon and assign it to the shape variable. Then, call the addPoint method defined in the Polygon class to add the endpoints of the lines that make up the ship:

```
public void addPoint(int x, int y)
```

Remember that the values you pass in here should have (0,0) in the center of the ship as in our example above. They should not use the left and top values passed in to the Ship constructor.

Finally, define a paint method in the Ship class that draws the ship. This method should take a Graphics object as a parameter. The first thing that it should do is “coerce” the graphics object into a Graphics2D object using this code:

```
Graphics2D g2D = (Graphics2D) g;
```

Then, you can use this code to draw your shape.

```
g2D.translate (windowWidth/2, windowHeight/2);
g2D.draw(shape);
```

Change the constructor in AsteroidsGame to construct a Ship and save it in an instance variable. Change the paintComponent method in AsteroidsGame to call the paint method in Ship in addition to drawing the black background. Now, when you run your program there should be a ship in the middle of the window.

**Step 2: Add controls to move the spaceship**

At this point, your main method should look very similar to the main method that you used in your IDCard lab. You will need to add one more line of code at the end of the main method to get the animation running, which is necessary to get the spaceship moving. As we saw with the SnowAnimation in class, an animation works by repeatedly calling nextFrame(). The Game class provides this behavior, but you need to add a call to the start() method defined in the Game class to make this happen. Since AsteroidsGame extends Game, remember that you can call start() just as if it was defined locally.

The user will use the up-arrow and down-arrow keys to move the ship. Up-arrow will accelerate the ship up to some maximum speed. Down-arrow will slow it down and eventually stop it. The ship cannot go in reverse.

To handle the user typing on the keys, your AsteroidsGame class must “implement KeyListener”. This interface requires you to define 3 methods:

```
public void keyPressed (KeyEvent event)
public void keyReleased (KeyEvent event)
public void keyTyped (KeyEvent event)
```

The only one you need to define is keyPressed. The other two need to be declared, but their bodies should be empty. As with other listeners, you will need to add a listener so that these methods actually get called. The method that you need to call is:

```
public void addKeyListener (KeyListener l)
```

To determine which key a user has typed, call getCharCode from the KeyEvent class:

```
public int getCharCode()
```
The values of interest for this lab are:

<table>
<thead>
<tr>
<th>VK_UP</th>
<th>up arrow key</th>
</tr>
</thead>
<tbody>
<tr>
<td>VK_DOWN</td>
<td>down arrow key</td>
</tr>
<tr>
<td>VK_LEFT</td>
<td>left arrow key</td>
</tr>
<tr>
<td>VK_RIGHT</td>
<td>right arrow key</td>
</tr>
<tr>
<td>VK_SPACE</td>
<td>space bar</td>
</tr>
</tbody>
</table>

If the user types the up arrow key, you should call a method in the Ship class to increase the speed of the ship, but don’t go above some maximum value. If the user types the down arrow key, decrease the ship’s speed, but don’t go below 0. Remember that speed is controlled by how far a ship moves on consecutive frames. So, to make the ship move faster, you will increase this distance. To go slower, decrease it.

Remember to add a key listener to the AsteroidsGame in its constructor. In addition, add this line to the constructor:

```java
setFocusable(true);
```

This is required for your game to accept any input.

Now, you are ready for some animation! Change the nextFrame method in AsteroidsGame to call a nextFrame (or move) method in Ship. In Ship’s nextFrame method, decrease (since the ship is pointing straight up) the value of the top of the ship by the distance it should move between frames.

Now, when you run your program and type the up arrow key, the ship should move up the window until it disappears completely.

What you want to do next, is cause the ship to wrap around so that if it goes off the top, it should reappear at the bottom. How can you write code to check when it has gone off the top? Where should it reappear? Write the code to do that.

Also, test that the down-arrow key slows the ship down and eventually stops it.

**Step 3: Add controls to rotate the spaceship**

Next add the controls to rotate the ship. Right-arrow should rotate the ship right. Left-arrow should rotate the ship left. You will need an instance variable in your ship to remember the amount of rotation. You will also need to add the call the Graphic2D’s rotate method to cause the rotation to actually happen. Test your right and left arrow keys.

Now, let’s see what happens if we try to move a rotated ship. Yikes! It still goes straight up and straight down, not in the direction the ship is pointing! Unfortunately, we can’t avoid doing some trig here. Essentially, we need to take the ship’s speed and apply some of it vertically and some of it horizontally. Here is a method that will do that. Speed is the number of pixels that you would want to move if it was going straight up. rotation is the amount of rotation, top is where the top of the ship is before it moves and left is its left edge before it moves. top & left here are the same values that you would pass to the translate method of Graphics2D. This method updates top and left to cause the ship to move in the direction it is pointing. Try it.

```java
protected void calculateXY() {
    double xOffset = speed * Math.sin(rotation);
}
```
double yOffset = -speed * Math.cos(rotation);

top = top + yOffset;
left = left + xOffset;
}

Now, you may also notice that if the ship moves off the screen on the left, right or bottom, it is gone forever. You need to adjust your move method to have the ship wraparound in any direction.

**Step 4: Add asteroids**

Now you need to define another class, Asteroid, to be the asteroids out to destroy the ship. Before you get too worried about this, notice that the asteroids are a lot like ships! In fact, they are even easier. In what ways are asteroids like ships?

- They both have polygons for shapes.
- They both have a location.
- They both have a rotation.
- They both wrap around when they go off the screen.

In what ways are they different from the Ship?

- They move at a constant speed.
- Their rotation never changes.

So, in fact, they have a lot in common with ships and are actually simpler!

You may be tempted to copy-and-paste the body of Ship into the Asteroid class, but don’t do that! Instead, create a new class called AsteroidGameShape and move everything that is similar about ships and asteroids from the Ship class into this new class. Then say that Ship “extends AsteroidGameShape”. It’s probably best to do this one method at a time, running your program after each change to make sure it still works.

Now, create your Asteroid class and say that it “extends AsteroidGameShape” as well. Add to Asteroid anything specific to asteroids.

Then, go to the AsteroidGames class and construct an asteroid. On each frame, call the asteroid’s nextFrame method. Also, call its paint method from AsteroidGame’s paint. Start with just one asteroid. Try it. Do you have an asteroids flying around? If not, maybe you need to change what is in the AsteroidGameShape. Maybe more of the Ship class should be moved into the new superclass, or maybe you’ve moved too much already. Think about similarities and differences to determine exactly what goes into each of these three classes.

Once that works, change AsteroidsGame to hold an array of asteroids. Move each of them in nextFrame and paint each of them in paintComponent.

Congratulations! You do not yet have a complete game, but you have done some pretty cool animations and you have gotten quite a bit of experience working with inheritance. Everything from here on is extra credit, but I give more details for those of you who would like a more fully functional game.

**Extra Credit Step 5 (Moderate difficulty): Detect collisions between asteroid and space ship**

If the ship collides with an asteroid the ship should disappear from the screen and the text “Game Over” or something similar should appear. You should define a “collidesWith” method in your AsteroidGamesShape class that takes another AsteroidsGameShape as a parameter. You should call this method in AsteroidGame’s nextFrame method, once for each asteroid, checking to see if it has hit the ship.
The simplest way to detect collisions (and what I recommend that you do) is to get the “bounding box” for the shape. A bounding box is the smallest upright rectangle that encloses the shape. The type of a bounding box is Rectangle2D, which is defined by Java. It provides a method:

```java
public boolean intersects (Rectangle2D r)
```

So, if you have the bounding box of both the ship and an asteroid, you can call this intersects method. If it returns true, the two shapes have collided (or at least we will claim that they have). To find the bounding box for an AsteroidsGameShape, create an instance variable named “bounds” whose type is Rectangle2D. Then add this line of code to the end of the paint method in AsteroidsGameShape:

```java
bounds = TransformCalculator.calculateRotatedBounds(
    shape.getBounds2D(), left, top, rotation);
```

This is the bounding box that you should use in your collision detection.

**Extra Credit Step 6 (Moderate difficulty): Add controls to shoot bullets**

The user types the space bar to shoot a bullet. The bullet should start at the nose of the ship and travel in the same direction that the ship is facing. It should move faster than the ship.

You will need to create a new class called Bullet. Notice that it is also an AsteroidsGameShape! However, it moves differently than the others, because the bullets do not wrap around. Therefore, you will need to override the move method that is in your AsteroidsGameShape.

The ship should not be able to shoot bullets after it is destroyed.

**Extra Credit Step 7 (Moderate difficulty): Detect collisions between bullets and asteroids**

This is very similar to the collision detection that you already did for step 5. You will need to check each bullet against each asteroid. Also, when you destroy an asteroid, not only should it disappear from the screen, but it also shouldn’t be able to collide with the ship. (Sounds obvious, right?)

A bullet should only be able to destroy 1 asteroid.

If the user destroys all the asteroids, a “Game Won” message should be displayed.

**Other Extra credit ideas**

You should not work on extra credit until you have all the steps above working well and you have added comments and used good style. You will not receive extra credit points if you have not done a good job on the basic assignment.

- Simple: Add a button to allow the user to restart the game, add another kind of asteroid with a different shape, allow the user to hyperspace the ship.
- Moderate: Display a score, introduce multiple levels
- Challenging: Have large asteroids splinter into multiple pieces.

**Grading**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Step 1: Draw a spaceship</td>
</tr>
<tr>
<td>15</td>
<td>Step 2: Add controls to move the spaceship</td>
</tr>
</tbody>
</table>
Step 3: Add controls to rotate the spaceship

Step 4: Add asteroids

Comments (including javadoc comments) and assert statements

Style

Extra credit (simple, moderate, challenging, respectively)

Turning in Your Work

Create a jar file, being sure to include your source code. Then, go to ella. Click on COMSC 102 in the toolbar menu across the top. Then click on Assignments in the left column. Click the submit link for this assignment. Click the Add Attachment button. Use the Browse... button to upload a local file. You should submit your jar file. Click Continue. Then click the Submit button.

API for the Game and TransformCalculator classes

Class Game

Game provides the basic functionality to create games involving animation. Subclasses should define the nextFrame method. This method is called every 30 seconds, giving the game an opportunity to update the display and do whatever operations it needs to during the animation.

Game

public Game()
    Constructs a new Game.

public void run()
    Runs the animation. This method should not be called directly. Instead, to start the animation, call the start() method.

public void start()
    Starts the animation

public void stop()
    Stops the animation.

Class TransformCalculator

TransformCalculator provides some utility routines to translate from the coordinates used by shapes to actual screen coordinates.

public static java.awt.geom.Rectangle2D calculateRotatedBounds(java.awt.geom.Rectangle2D unRotatedBounds,
    double horizontalOffset, double verticalOffset, double rotation)
Find the bounding box for a rotated shape.

**Parameters:**
- `unRotatedBounds` - the bounds before rotation and translation
- `horizontalOffset` - the distance to translate horizontally
- `verticalOffset` - the distance to translate vertically
- `rotation` - the amount to rotate, in radians

**Returns:**
- a bounding box that surrounds the rotated rectangle

```java
public static java.awt.geom.Point2D transform(java.awt.geom.Point2D pt,
                                              double horizontalOffset, double verticalOffset, double rotation)
```

Transform a point by applying a translation and rotation.

**Parameters:**
- `pt` - the original point. This should not be null.
- `horizontalOffset` - the distance to translate the point horizontally
- `verticalOffset` - the distance to translate the point vertically
- `rotation` - the rotation to apply, in radians

**Returns:**
- the point after applying the transformation