Introduction to Graphics

In this lab, you will use the Graphics class to create some pictures. You will also have to look at the public documentation for several other classes to learn how to use them. Finally, you will have to learn how to calculate positions and sizes in terms of variables so that your pictures can be placed anywhere and any size.

Coordinates on the Screen

The screen and windows have a coordinate system similar to the one that you use in your math classes but with two key differences. Every pixel on the screen can be referenced using an x and y coordinate pair which indicate the horizontal and vertical distance from a fixed origin, just as points in the Cartesian plane. However, these screen coordinates are all non-negative integers rather than the real numbers that you are used to. For the most part, this is of small consequence. If you need to plot a point whose coordinates are not integers, you can just round to the nearest integer and the user’s eyes will probably not be able to tell the difference. It also means that you can only plot points in what you would normally think of as the first quadrant. The second difference is of larger consequence: in the Cartesian plane, the origin (0, 0) is at the lower left corner of the first quadrant. In computer graphics, the origin is at the TOP left corner. This means that while x coordinates increase from left to right as you are used to, y coordinates increase as you move down the screen rather than as you move up. It takes some getting used to, but it will become second nature after a while.

Colors

Every pixel on the screen is set to a color. In Java (and most computer applications), colors are represented using the RGB or RGBA systems. RGB stands for “Red Green Blue” and A stands for “Alpha”. In this system, every color is represented with 3 (4) integers between 0 and 255. The red, green and blue numbers describe the hue of the color. White is given by R = G = B = 255. Black is R = G = B = 0. All colors (well, enough that the eye cannot discern differences) can be described using different combinations of R, G, and B. If you open Paint and select the “Edit Colors” button, you can find the RGB values for a large number of colors. Alternatively, you can find a listing here (or many other places on the web): <http://www.colorpicker.com/>. The Alpha number gives the opaqueness of the color. For the most part, we will not be using the alpha number, but you can use it to create interesting effects.

In Java, the Color class is used to encapsulate colors. There are several predefined colors available. You used them in the Turtle exercises to set the color of the Turtle. Some examples are Color.WHITE, Color.BLUE, Color.ORANGE etc. Of course, there are only a few available in this way. You can create new Colors in any combination of RGB (and A) values that you like by using new:

Color purple = new Color( 128, 0, 255 ); // creates a color to represent purple

Color purpleTranslucent = new Color( 128, 0, 255, 128 ); // same, but specifies the alpha value

The Graphics Class

In Java, the Graphics class provides several useful methods for drawing images in a window. In this project, you will be provided with a Graphics object, usually called g. Your job is to use the functions provided by the object to draw a picture. Here are some of the methods that you will find useful. You can find documentation for all the methods of the Graphics class in Netbeans (just type “g.” and wait for a dialog box to come up with the names and descriptions of the methods) or go to the documentation page: <http://docs.oracle.com/javase/7/docs/api/java/awt/Graphics.html>.

setColor( Color c ) – Consumes a Color object. Sets the current drawing color. You must use this if you hope to see anything on the screen (the default color is the same as the background, so nothing shows up unless you change the color to something else). Here are some examples:

public void paint( . . . Graphics g . . . )

{

g.setColor( Color.BLACK ); // sets the color to black

. . .

g.setColor( new Color( 128, 0, 255 ) ); // sets the color to purple

. . .

Color brown = new Color( 128, 64, 0 );

g.setColor( brown );

. . .

}

drawRect( int ULCx, int ULCy, int width, int height ) – Draws a rectangle with its upper left corner at (ULCx, ULCy) and extending width pixels to the right and height pixels down.

fillRect( int ULCx, int ULCy, int width, int height ) – same as drawRect but fills the rectangle rather than outlining it.

drawOval( int ULCx, int ULCy, int width, int height ) – draws an ellipse inscribed in the rectangle described by the coordinates and the size.

fillOval( int ULCx, int ULCy, int width, int height ) – same as drawOval but filled rather than outlined.

drawArc( int ULCx, int ULCy, int width, int height, int start, int extent ) – draws part of an ellipse (see drawOval) starting at the angle start and extending for extent degrees. Angles are measured counter-clockwise from “east”. For example:

g.drawArc( 100, 100, 100, 100, 45, 90 );

produces the arc shown below. It starts at 45 degrees ccw from east and extends for 90 degrees. Note that the circle has radius 50 (the width and height are both 100, but they indicate the vertical and horizontal diameters of the circle, not the radii. The point in the top left is (100, 100).



fillArc( int ULCx, int ULCy, int width, int height, int start, int extent ) – Like drawArc but fills in the sector described by the arc.

drawLine( int x1, int y1, int x2, int y2 ) – draws a line from (x1, y1) to (x2, y2).

In addition to these functions, there are functions for drawing and filling polygons. Because they use arrays (and we haven’t covered arrays yet), I’ve provided functions for drawing triangles. They can be found in the GraphicsHelperFunctions class.

drawTriangle( Graphics g, int x1, int y1, int x2, int y2, int x3, int y3 ) – draws a triangle with vertices at (x1, y1), (x2, y2) and (x3, y3).

fillTriangle( Graphics g, int x1, int y1, int x2, int y2, int x3, int y3 ) – same as drawTriangle but fills the triangle.

To use them, you have to use the name of the class followed by the function:

GraphicsHelperFunctions.drawTriangle( g, 50, 100, 200, 100, 130, 150 );

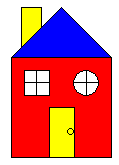
Exercises:

BasicHouse – to be started in class

Open HouseDisplay.java and find the paintHouse method:

public void paintHouse( int size, int Cx, int Cy, Graphics g )

The paintHouseMethod consumes three ints which describe the size (width) of the house and the coordinates of its center (Cx, Cy). Our job is to use these parameters to draw a house that looks like this:



The first part (the red square) is already coded for you:

g.setColor( Color.RED );

g.fillRect( Cx – size/2, Cy – size/2, size, size );

g.setColor( Color.BLACK );

g.drawRect( Cx – size/2, Cy – size/2, size, size );

Notice that the functions fillRect and drawRect require the coordinates of the upper left corner. We are given the center of the square (Cx, Cy). To find the ULC, we have to subtract size/2 from each of these coordinates. The width and height of the square are both size.

To test this function, open IntroductionToGraphics.java, find the main function and uncomment the line:

runBasicHouse();

On your own, try to draw the yellow door (the width is size/4 and the height is size/2), and the blue roof of the house (calculate the coordinates of its vertices using Cx, Cy, and size).

To make sure that you are using the size and coordinate parameters correctly, you can run the pulsing house function. Uncomment this line in the main function:

runPulsingHouse();

The house should now be centered rather than to the left of the window and should get bigger and smaller. Notice that your picture drawing code does not do anything to animate the picture. It simply draws the picture in the current size and location.

Finishing this picture is part of the homework for the lab.

BouncingBall – to be done in class

In this exercise, we will create a picture of a ball which moves around the screen:



Open BouncingBallDisplay.java. The paint method of this class consumes a BouncingBall object:

public void paint( BouncingBall b, Graphics g, int w, int h )

{

. . .

}

The BouncingBall class is included in the project. You do not need to implement anything in it. However, it provides several methods which we will need to complete the drawing code:

class BouncingBall {

. . .

Color getColor() { . . . } // returns the Color of the ball

int getX() { . . . } // returns the x coordinate of the center of the ball

int getY() { . . . } // returns the y coordinate of the center of the ball

int getRadius() { . . . } // returns the radius of the ball

. . .

}

Use these methods to draw a picture of a ball of the appropriate color in the appropriate location on the window. To test, uncomment the line:

runBouncingBall();

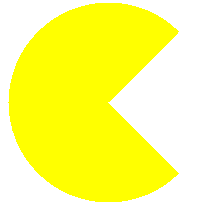
in the main function. You should see a single blue ball moving around the screen. If you like, you can uncomment the line:

runBouncingBalls();

to see a screen-saver like program with lots of balls of different sizes and shapes.

PacMan – to be done in class

In this exercise, we will draw a PacMan:



Open PacManDisplay.java. The paint method here consumes a PacMan object:

public void paint( PacMan p, Graphics g, int w, int h ) { . . . }

The PacMan class provides several methods:

class PacMan

{

. . .

public int getRadius() { . . . } // gets the radius of the PacMan

public int getMouthAngle() { . . . } // gets the angle that the mouth makes

public int getX() { . . . } // gets the x coordinate of the center of PacMan

public int getY() { . . . } // gets the y coordinate of the center of PacMan

. . .

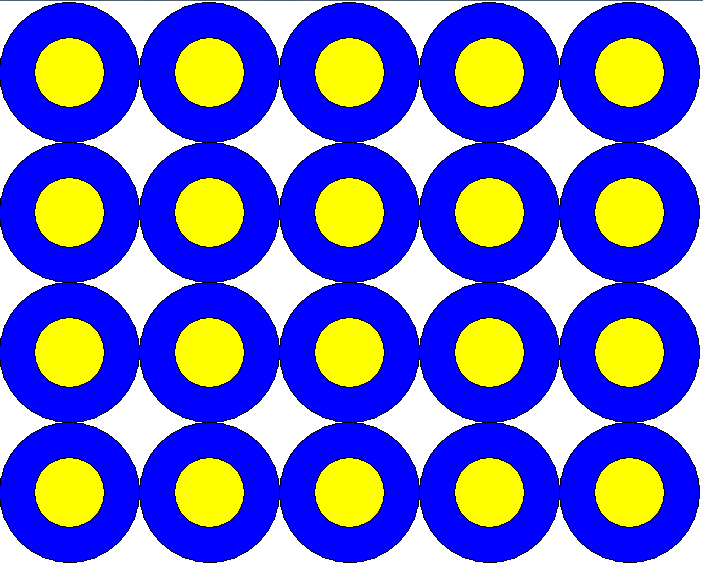
}

Use these methods and the fillArc method of the Graphics class to draw the PacMan. To test, uncomment the line

runPacMan();

in the main function. You should see the PacMan opening and closing its mouth and moving slowly across the screen.

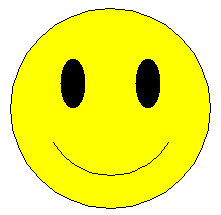
TargetRectangleDisplay – to be done in class (time permitting)



In this exercise, we’ll draw a rectangular array of targets. Open TargetRectangleDisplay.java. The paintTargets method consumes two ints, numRows and numCols which represent the number of rows and columns in the rectangular array, and two more ints, w and h which represent the width and height of the window. The radius of the blue circles should be the smaller of w/numCols and h/numRows. The radius of the yellow circles is half the radius of the blue circles. The rectangular array should be centered at the center of the window.

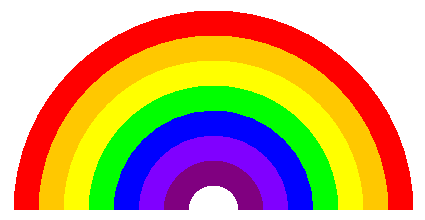
Start by computing the radius of the blue circles, and then use a nested for loop to draw the rectangle of targets.

SmileyFace – to be done at home



Open SmileyFace.java. The paint method consumes an integer, r, which gives the radius of the SmileyFace, and two other ints, w and h which give the width and height of the window. Draw the smiley face in the middle of the window (centered at w/2, h/2).

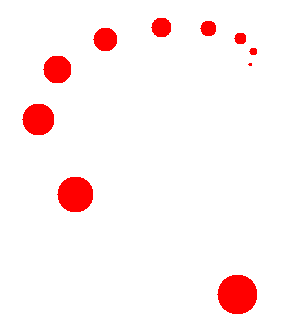
Rainbow



Open Rainbow.java. The paint method consumes an integer r which gives the radius of the rainbow and the width and height of the window. Draw the rainbow in the center of the window.

For reference, the Indigo and Violet rings (ie the inner two bands) have RGB values (128, 0, 255) and (128, 0, 128) respectively.

SnakeCursor



In this exercise, you will draw a snake of balls. As you move your cursor around the window, the snake will follow your cursor.

Open SnakeCursorDisplay.java. The paint method consumes a SnakeCursor object. A SnakeCursor consists of several balls (0, 1, 2, … N), each with a different location and radius (ball 0 is the largest with ball 1 slightly smaller and so on). The SnakeCursor class has several methods which will be of use to you:

class SnakeCursor {

. . .

public int getNumBalls() { . . . } // returns the number of balls in the snake

public int getX( int i ) { . . . } // returns the x coordinate of the center of ball i

public int getY( int i ) { . . . } // returns the y coordinate of the center of ball i

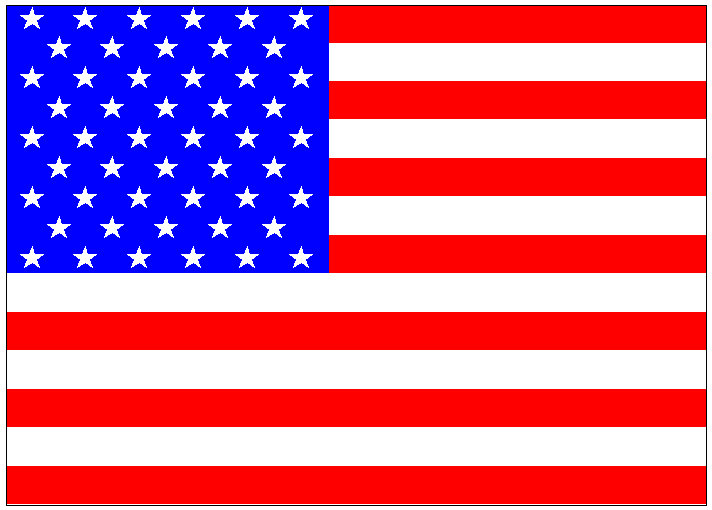
public int getBallRadius( int i ) { . . . } // returns the radius of ball i

. . .

}

Complete the paint method in SnakeCursorDisplay so that it draws all the balls in the snake cursor. Hint: You will need a for loop.

AmericanFlag



Open AmericanFlag.java. The paintFlag method consumes two ints, fw and fh, which represent the width and height of the flag, and two other ints, w and h, which represent the width and height of the screen. Your job is to draw the American Flag shown above centered at the center of the window.

To help with this task, the GraphicsHelperFunctions class includes a function for drawing and filling stars:

GraphicsHelperFunctions

{

. . .

// draws and fills a star centered at (x,y) with radius r

public static void fillStar( Graphics g, int x, int y, double r ) { . . . }

. . .

}

Also, for reference:

The blue field is 6/13 the width of the whole flag, and 7/13 the height of the whole flag.

In the rows of 6 stars, the first star is centered 1/12 of the width of the blue field from the left edge, and the horizontal distance between each star is 1/6 of the width of the blue field.

In the rows of 5 stars, the first star is offset 1/6 of the blue field from the left edge and the horizontal distance between stars is 1/6 of the width of the blue field.

The first row of stars is down 1/18 of the height of the blue field from the top.

The vertical distance between rows of stars is 1/9 of the height of the blue field.

Hint: This is a test of how well you can use for loops.

SimpleRadar



In this exercise, you will draw a picture of a “radar” which rotates counter-clockwise. Open SimpleRadarDisplay.java. The paint method consumes a SimpleRadar object. This object provides several methods for you to use in your picture:

class SimpleRadarDisplay {

. . .

public int getInnerRadius() { . . . } // returns the inner radius of the radar (see picture)

public int getOuterRadius() { . . . } // returns the outer radius of the radar (see picture)

public int getStartAngle() { . . . } // returns the starting angle of the radar (see picture)

public int getNumBlocks() { . . . } // returns the number of blocks in the radar (the picture

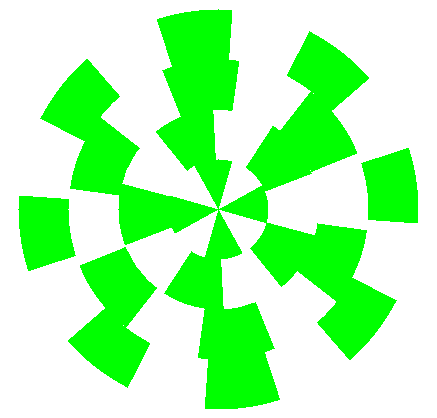
// has 5)

. . .

}

Hint: use a for loop!

Radar



This is a more complicated version of the SimpleRadar exercise. In this exercise, the Radar has several rings of blocks. Each ring has a different number of blocks and a different starting angle and moves in a different direction at a different speed. The Radar has one radius, and the rings all have the same width (in the example picture, each ring has a width of ¼ the radius since there are 4 rings).

The Radar object provides these functions:

class Radar {

. . .

public int getRadius() { . . . } // returns the radius for the whole radar

public int getNumRings() { . . . } // returns the number of rings (4 in the example)

public int getNumSegments( int r ) { . . . }// returns the number of segments in ring r

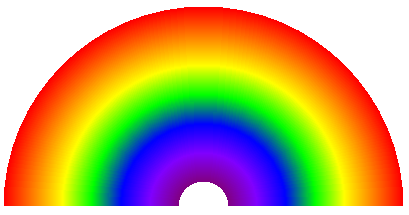
public int getStartAngle( int r ) { . . . } // returns the start angle for ring r

. . .

}

Hint: more for loops! Use a nested for loop.

Bonus: SmoothRainbow



Same as Rainbow but smooooth.