**Homework 2: Object Oriented Graphics**

**18% of grade. Tuesday, Jan. 22, 2013 - Tuesday, Feb. 12, 2013**

In this assignment, you are going to create an object-oriented graphics system that provides a retained-object model to automatically handle redrawing. You will create a set of Java classes out of which graphical objects can be constructed. Your classes will be responsible for handling refresh when an area of the screen is damaged and needs to be redrawn.

All the interface and Java files you need are either in [hw2-files-swing.zip](http://www.cs.cmu.edu/~bam/uicourse/830spring13/hw2-files-swing.zip) or [hw2-files-android.zip](http://www.cs.cmu.edu/~bam/uicourse/830spring13/hw2-files-android.zip) (depending on which OS you are targeting). Both of these are set up as Eclipse projects.

**Graphical Objects**

You will provide the following graphical object classes:

* OutlineRect: draws an unfilled rectangle of a given color and line thickness.
* FilledRect: draws a filled rectangle of a given color.
* Line: draws a line of a given color and line thickness.
* Icon: draws an image.
* Text: draws a string with a given font and color.
* ... extra credit for implementing other graphical objects, see below

These classes should have the following required constructors:

public OutlineRect (int x, int y, int width, int height, \*\*\* color, int lineThickness); public FilledRect (int x, int y, int width, int height, \*\*\* color); public Line (int x1, int y1, int x2, int y2, \*\*\* color, int lineThickness); public Icon (\*\*\* image, int x, int y); public Text (String text, int x, int y, \*\*\* font, /\*Android only\*/ int fontSize; //Swing has the size in the Font, but Android doesn't \*\*\* color);

Where the color parameter either a java.awt.Color under Swing/awt or else an int under Android with a value from android.graphics.Color.

The image parameter for Icon is either a java.awt.Image under awt/Swing or a android.graphics.Bitmap under Android. There is a helper routine in TestFrame called loadImageFully() which loads the image file by name. Under Android, that routine assumes the images are in the "assets" folder of the Android project.

The font parameter for Text is a java.awt.Font under awt/Swing or a android.graphics.Typeface under Android. Note that since Android's Typeface does *not* include a size, we change the interface to creating a Text object to include an extra size parameter, but only under Android.

Based on Jeff Stylos's research ([see reference](http://portal.acm.org/citation.cfm?id=1248828)), each class should also support a default constructor with no parameters, that sets all the fields to an appropriate default value, e.g., public OutlineRect();

Your graphical objects should be drawn using the appropriate graphics model: Java 2D for the desktop, and android.graphics for Android. Each object corresponds to a method of the drawing packages: java.awt.Graphics2D or android.graphics: OutlineRect corresponds to drawRect(), FilledRect to fillRect() in awt and drawRect with setStyle(Paint.Style.FILL) in Android, etc.

For OutlineRect, FilledRect, and Icon, the (x,y) point is the top-left corner of the of the object (so it is *inclusive*). For Line, (x1,y1) and (x2,y2) are the endpoints of the line. For Text, (x,y) is the position of the first character's *baseline*.

Each class should have a get*Param*() and set*Param*() method for each parameter in its constructor. For example, OutlineRect should have the following methods:

public int getX (); public int getY (); public int getWidth (); public int getHeight (); public \*\*\* getColor (); // returns a java.awt.Color under Swing/awt or else an int under Android. public int getLineThickness (); public void setX (int newX); public void setY (int newY); public void setWidth (int newWidth); public void setHeight (int newHeight); public void setColor (\*\*\* newColor); // takes a java.awt.Color under Swing/awt or else an int under Android. public void setLineThickness (int newLineThickness);

All your graphical object classes must implement the GraphicalObject interface:

public interface GraphicalObject { public void draw (\*\*\*\*\*\*\* graphics, \*\*\*\* clipShape); public BoundaryRectangle getBoundingBox (); public void moveTo (int x, int y); public Group getGroup (); public void setGroup (Group group); public boolean contains (int x, int y); public void setAffineTransform(\*\*\*\* af); public \*\*\*\* getAffineTransform(); }

where the graphics parameter is either Graphics2D (awt) or Canvas (Android), and clipShape is either a Shape in awt or a Path in Android; and setAffineTransform and getAffineTransform use either java.awt.geom.AffineTransform or android.graphics.Matrix.

draw() draws the graphical object on a drawing surface, clipped to the specified clipping shape (usually a rectangle -- unless you implement rotations, discussed below), which will be in the coordinate system of the group that the graphical object is in (see below). (The clipping shape is in the coordinate system of the group's container). Note that users of this API will *not* call draw -- it is only called by your toolkit itself. Users will just set properties, and your toolkit will automatically call draw at the appropriate times.

getBoundingBox() returns the smallest rectangle that contains all the pixels drawn by the graphical object. Your graphical objects will have to accurately compute their bounding boxes, taking into account line thickness and the drawing behavior of the graphics package. For Text objects, you might use java.awt.FontMetrics in awt or Paint.FontMetrics in Android to compute the bounding box. The bounding box should be the smallest box such that if one were to draw a white filled rectangle using that box, all the pixels would be erased. Or if I set a clip rectangle to that box and drew the object, none of the object would get clipped. Note that the BoundaryRectangle returned by getBoundingBox() is different from the OutlineRect graphical object you have to implement. Also note that it is designed based on a java.awt.Rectangle, in having a width and a hieght, which is *different*from the android.graphics.rect, which has a right and a bottom.

The methods getBoundingBox, contains, and moveTo all use the coordinate system of the parent of the graphical object (the group that this object is in). That is, they are in the same coordinates of the X and Y for a rectangle. It is OK if contains() just checks if the point is inside the bounding box.

Setting the width and height of text or image objects can be defined to be a no-op. Note that the getX and getY and setX and setY for text objects will be the *baseline* position, but of course, the x and y in the Rectangle returned by getBoundingBox must be for the outside of the text object. However, calling MoveTo on a text object should always move the top left corner to the specified position passed to MoveTo (therefore, mytext.moveTo(10,10) and mytext.setY(10) will not go to the same Y value).

Note that by default, drawing a filled rectangle and an outline rectangle using the same values for width and height actually draws *different size rectangles*, which I think is weird. Please have the width and height parameters of *both* OutlineRect and FilledRect draw the same size object. In particular, if x is 0 and the width is 3, then pixels 0, 1, and 2 should be drawn by both.

moveTo() moves the graphical object so that the top-left corner of its bounding box is at (x,y). You have to determine how this affects the coordinates of the underlying graphical object. Calling moveTo() on a Line object should move both endpoints so that the line has the same angle and length after the move.

getGroup() and setGroup() get and set the group to which the graphical object belongs. If the object doesn't belong to a group, getGroup() returns null. Groups are described in more detail next.

setAffineTransform is used to tell the object the transform that should be used to draw the object itself, and getAffineTransform returns the transform that was previously set. The object should assume the identity transform if setAffineTransform is not called. You will probably only need to use this method if you implement RotateGroup, below.

**Groups**

You will also provide the following grouping classes:

* SimpleGroup: draws a group of graphical objects at fixed positions.
* LayoutGroup: automatically lays out its children horizontally or vertically.
* ScaledGroup: shrinks or enlarges a group of graphical objects by a scale factor.
* ... extra credit for implementing other grouping objects, see below

These classes have the following required constructors:

public SimpleGroup (int x, int y int width, int height); public LayoutGroup (int x, int y, int width, int height, int layout, int offset); public ScaledGroup (int x, int y, int width, int height, double scaleX, double scaleY);

Plus the default constructors, of course. Each group class should provide set*Param*() and get*Param*() methods for each parameter in its constructor.

A group defines a new coordinate system for its children. The coordinates of the children are interpreted relative to the group's origin (the x, y point passed to the group constructor). Children should be clipped to the bounding box of the group.

All grouping classes should implement the Group interface:

public interface Group extends GraphicalObject { public void addChild (GraphicalObject child); public void removeChild (GraphicalObject child); public void resizeChild (GraphicalObject child); public void bringChildToFront (GraphicalObject child); public void resizeToChildren (); public void damage (BoundaryRectangle damagedArea); public java.util.List<GraphicalObject> getChildren(); public Point parentToChild (Point pt); public Point childToParent (Point pt); public static final int HORIZONTAL = 1; public static final int VERTICAL = 2; }

Where Point is java.awt.Point or android.graphics.Point.

Notice that the Group interface extends GraphicalObject, so your group classes must implement the GraphicalObject methods too.

addChild() and removeChild() add and remove graphical objects from the group. These methods should call setGroup() on the child. If one attempts to add an object to more than one group, this should raise an exception. removeChild() causes the child object to now be in *no*group.

resizeChild() notifies the group that the bounding box of one of its children has changed.

bringChildToFront() changes the display order of the children so that the specified child is drawn in front of other children. The default drawing order draws the last child added in front.

resizeToChildren() changes the group's width and height to fit around its children as tightly as possible. The children should not be repositioned, so children at negative (x,y) positions will still lie outside the group's bounding box after this method is called. This is a helper function for the users of your toolkit, and should not be called automatically.

damage() collects rectangles that need to be redrawn. When a graphical object changes, it should call the damage() method of the group it belongs to, passing its old bounding box and its new bounding box if it has moved. If a group object is part of another group object, it should propagate the damaged rectangle up to its parent group. The next time draw() is called on the group, it should call draw() on all children. Note that objects should *not* call draw when properties change -- draw will be called later, often with a collection of objects to be redrawn. (See the discussion in lecture about the tradeoffs of calling draw() on all children vs. calling draw() on only the children that intersect the damaged areas. In this assignment, we are just redrawing all children, but with an appropriate clipping rectangle.)

The damage rectangle passed up from a group should use coordinates with respect to the group's container.

When draw() is called on the group, it should call draw() on all children. Note that it will have to transform the coordinates of the rectangle to make sure they are correct for the objects inside the group.

The top-level call to draw() is the redraw call in the TestFrame.java file. Note that the test programs like TestOutlineRect.java call redraw after each change.

LayoutGroup has two extra parameters that determine how its children are positioned. The layout parameter has the following possible values, defined as constants in the Group interface. HORIZONTAL means that children are placed side by side from left to right, with top edges aligned. VERTICAL means that children are arranged top to bottom, with left edges aligned. The offset parameter specifies how much space to put between each child. Offsets may be negative, in which case adjacent children should overlap. The offset is not used before the first child, which should always be placed at position (0,0). Children should be laid out in drawing order, so bringChildToFront() should make the specified child the last object in the layout.

Extra credit is given for implementing other kinds of automatic layout; see below.

ScaledGroup draws its children scaled in x and y. Scale factors less than 1.0 shrink the children, and scale factors greater than 1.0 enlarge them. Scaling affects both positions and size, so two children of a ScaledGroup with scaleX=0.5 are not only half as wide but also half as far apart (in the x direction, at least). Be sure to take scale factors into account when you compute bounding boxes and damage rectangles.

Note that groups are transparent, so the parts of a group in which there are no graphical objects should show whatever was there before the group was drawn.

getChildren() returns a list of the group's children. List refers to java.util.List of GraphicalObjects, which is an interface implemented by several java.util classes, including Vector, LinkedList, and ArrayList. The children should be listed in display order, so the last child in the list is the frontmost. The caller should treat the return value of getChildren() as immutable.

parentToChild() and childToParent() translate between the group's coordinate system and its parent's coordinate system. parentToChild() takes a point in the parent coordinate system and maps it down to the group's coordinate system. For example, if the group is located at (5,10) in its parent, then parentToChild(Point(5,10)) should return Point(0,0). Similarly, childToParent() maps a point in the group's coordinate system up to the parent coordinate system. Most groups will implement these methods as simple translations, but ScaledGroups should take scaling into account as well. (Note: since Point represents integer coordinates, you'll lose some precision if you put a ScaledGroup inside another ScaledGroup. If accurate scaling were important to our toolkit, we'd want to use floating-point coordinates.).

**Other Important Notes**

* Do not change the GraphicalObject or Group interfaces. Your code should work with GraphicalObjects and Groups written by other developers. The test driver used for grading will include GraphicalObject and Group implementations that your code has never seen before.
* The test programs supplied to you should not need to be changed. If you seem to need to change them, your design is probably wrong.
* Be careful with ungrouped objects. Your graphical objects should not throw any exceptions if they are used without being added to a group.
* If you are working for the desktop (Swing/AWT), I am using the newest version of Eclipse, Version: Juno Service Release 1, with the Java SE Dev Kit 7 (64 bit) on Windows.
* If you work in Android, I am using the newest version of the Android Developer Tools, Build: v21.0.1-543035, which includes the Eclipse Platform, JDT, CDT, EMF, GEF and WTP. The simulator I am using is:
  + Device: Nexus S (4.0", 480x800: hdpi)
  + Target: Android 4.2 - API Level 17
  + CPU/ABI: ARM (armeabi-v7a)
  + Keyboard: ***not*** hardware keyboard present
  + Skin: ***not*** Display a skin with hardware controls
  + no cameras
  + Memory Options: RAM 343; VM Heap: 32
  + Internal Storage: 200 MiB
  + SD Card: ***none***
  + Emulation Options: ***none***
  + and when I run it, the launch options I use: all unchecked (***not*** Scale display to real size, ***not*** wipe user data)

**Extra Credit**

For extra credit, you can implement more graphical objects, groups, and layouts than the minimum requirement. Here are some ideas. You may have others.

Graphical objects:

* Ellipse: draws an ellipse or circle.
* Arc: draws a segment of an ellipse.
* Polyline: draws a sequence of line segments with a given thickness, color, join style, and end cap style.

Implement a method on GraphicalObject that determines if an (x,y) point *hits* the object (i.e., is part of the drawn part of the object). That is, gObj.hits(x,y) returns true if and only iff x,y is a pixel drawn by gObj. Easy for FilledRects, somewhat or quite tricky for all other kinds of objects.

Groups:

* RotatedGroup: rotates the entire group of children by a given angle around the center of the group.

Layouts:

* Fill: arranges children left-to-right, wrapping to next line when the group width is reached.
* Grid: arranges children in a grid with a fixed number of rows and columns.

It is not necessary to make the program thread-safe. In this homework, everything will be done in the same thread. However, extra points if you do make everything thread-safe so things can go in different threads, and if you provide a good test program that demonstrates it.

**Resources**

You will be provided with source code for the GraphicalObject and Group interfaces. You will also receive source code for some test programs. These programs have main() methods in Swing or are part of the main activity in Android, so you can run them directly. The tests we provide are far from complete, so you should certainly write your own. We will use these test programs and others to grade your project. You can modify the test programs for your own use, but your code shouldn't depend on any changes you make.

**If you implement additional graphical objects, groups, or layouts, please provide a custom test program for each one that demonstrates its features.**

Files:

* GraphicalObject.java -- public interface GraphicalObject
* BoundaryRectangle.java -- abstracts the java.awt.Rectangle behavior (subclass in Awt, implements it in Android)
* Group.java -- public interface Group extends GraphicalObject
* TestFrame.java -- a test harness that can display GraphicalObjects in a window. All the other test programs derive from this class.
* TestOutlineRect.java -- a test program that moves an OutlineRect object around.
* TestSimpleGroup.java -- a test program that moves OutlineRects around in a SimpleGroup.
* TestLayoutGroup.java -- a test program that moves OutlineRects around in a LayoutGroup.
* TestAllObjects.java -- a test program that creates and displays all 5 graphical objects.
* duke.gif, duke.gif -- sample GIF files used by TestAllObjects.

All the files can be found in this ZIP archive, which we have arranged using an Eclipse project.

[hw2-files-swing.zip](http://www.cs.cmu.edu/~bam/uicourse/830spring13/hw2-files-swing.zip) or [hw2-files-android.zip](http://www.cs.cmu.edu/~bam/uicourse/830spring13/hw2-files-android.zip) (depending on which OS you are targeting)

*(Last revision of this page: 1/21/2013)*

Back to [Homework Overview](http://www.cs.cmu.edu/~bam/uicourse/830spring13/homeworks.html)  
Back to [05-830 main page](http://www.cs.cmu.edu/~bam/uicourse/830spring13/index.html)