Submission to the ACM Java Task Force
A Student-Friendly Graphics Abstraction

Eric Roberts
Stanford University
eroberts@cs.stanford.edu

Summary

This proposal outlines a graphics package that adopts a different underlying model than the standard Java graphics classes. In my experience, this alternative model, which I described in a paper at ITiCSE'98 [5], has proven to be much easier for students to master and, moreover, allows them to create more impressive displays they can build using collections of graphical objects. The structure, which resembles the resolution-independent graphical model from PostScript [1], has proven to be successful over more than a decade of use in both C and Java implementations.

1. The problem

In the Java Task Force "Taxonomy of Problems in Teaching Java" [7], the most significant problem associated with the existing APIs—listed in the taxonomy as problem A1—is the difficulty involved in teaching the graphics model to novices. In my 1998 paper on "Designing a Java Graphics Library for CS1" [5], I identified three problems in the standard Java graphics model that make it hard to use at the introductory level:

1. *Each graphical component is responsible for repainting its contents whenever it needs to be redisplayed.* The Java abstract windowing toolkit (AWT), in both its original implementation and the more advanced Swing package introduced in Java 1.2, adopts a model in which each component must be able to refresh the display whenever a repaint request is generated. This approach is in contrast to the model used for example in my C-based graphics library from the early 1990s [4], in which graphical displays are persistent: once a drawing operation is invoked, the display mechanism keeps track of the image and repaints it as necessary. The problem with the Java approach is that students must use data structures to remember what needs to be refreshed on the screen. Novices, however, have no understanding of data structures, making it difficult to keep track of this information.

2. *The graphics context is relatively statelesss, particularly in its failure to include the notion of a current point.* The Java graphics model adopts an inconsistent policy in terms of the amount of local state maintained within a **`Graphics`** object used for drawing. Some aspects of the state, such as color and font information, are maintained internally as part of each **`Graphics`** context; others, such as the current pen position, must be maintained by the client. For example, if you want to draw a line under the Java model, you must pass the absolute coordinates of both endpoints to the **`drawLine`** method. An alternative design, which is used in Stanford's C-based graphics package, is to maintain the current position as part of the graphics state. Lines can then be drawn relative to the current position by specifying displacements rather than absolute endpoints. This strategy has two advantages for the novice user. First, it makes it easier to draw the same figure in several different positions on the screen. To draw multiple copies of a given figure, all a student has to do is move the pen to the appropriate starting positions and then call a common method that draws that figure relative to its starting point, often without requiring any coordinate arithmetic at all. Second, maintaining the notion of a current point makes it easier for students to visualize the graphical operations as if they themselves were in the position of the pen. In his book, *Mindstorms* [3], Seymour Papert argues that the LOGO Turtle, in part because it maintains a tangible geometric state, allows students to "identify with the Turtle and . . . bring their knowledge about their bodies and how they move into the work of learning formal geometry." Although the students in an introductory computer science course are no longer young children, they nonetheless can more easily understand the process of a library that maintains a state into which they can project themselves.

3. *The coordinate system is pixel-based and non-Cartesian, both of which serve to make it less intuitive for novices.* The most significant problem raised by the pixel-based design of the Java coordinate model is that all coordinates are expressed as integers. When you design a graphical display for the screen, particularly one that involves arcs or diagonal lines, real numbers often come up naturally in the calculations. While rounding the computed coordinate values to integers is not difficult, students often perform those calculations in such a way that errors accumulate over time. Moreover, the Java coordinate system is different from the Cartesian coordinate system students learn in high school in that its origin is in the upper-left corner. While many calculations are easy to invert, other geometrical computations, such as those involving angles (which continue to be specified in the Cartesian counterclockwise style), tend to remain confusing.

2. Issues raised by object-oriented graphical strategies

In an objects-first curriculum, the obvious solution strategy to the problems outlined in the preceding section is to create a library of graphical objects that know how to display themselves on the screen. Several leading colleges and universities have developed such packages and used them successfully in courses. Williams College, for example, has developed the Objectdraw graphical library [2]. Brown University has several years of experience with the NGP graphics package [8], which offers similar functionality.

I believe that would be valuable for the Task Force to endorse an object-oriented model along these lines. A well-designed, reasonably standard, object-based graphics package would prove immensely valuable to the community. In this proposal, however, I'm arguing that the Task Force might also choose to endorse a more traditional graphical package that supports a drawing model similar to that offered by Java, but without several of its shortcomings.

In my ITiCSE'98 paper [5], I proposed an object-oriented library that I called the *collage model.* Like the packages at Brown and Williams, the collage model provided a set of class definitions representing geometrical shapes and other graphical elements, which students could instantiate and then position in a window framework of some sort. My implementation of the collage model has nothing to recommend it over the more recent Objectdraw and NGP packages, which have been validated through more extensive use. The collage model, however, was only one of two strategies defined in the ITiCSE paper. The other was an extension to the AWT **`Graphics`** class designed for use in an introductory course. It is that model—or more properly, a later evolution of it—that I am proposing here.

My reason for seeking to endorse an arguably more primitive model alongside the object-oriented one grows out of my own experience with the Java graphics packages I designed. Both models were available as part of the MiniJava system [6] that we have been using as part of the Bermuda Curriculum Project over the past four years (**`http://bermuda.stanford.edu`**). The fact of the matter is that, given access to and documentation for both models, no one—not the students here developing courses, the Bermuda-based teachers offering them, or the students in those classrooms—ended up using the collage model. For some reason, it did not have the same appeal.

One possible explanation, of course, is that all the individuals associated with the project had greater familiarity with the more conventional drawing model than they did with the object-oriented collage paradigm. While that observation is certainly true, my sense is that the real reason is that students found it easier to create sophisticated graphical designs with the conventional drawing model than they did with the collage model. This reaction is not really all that surprising when you consider that one has far more latitude to create an interesting picture in the real world if one is drawing with colored pencils than if one is restricted to pasting up predefined shapes. For most students, it's harder to make an artistic collage than it is an artistic drawing.

3. Implentation overview

The graphics package described in this proposal includes the six class and interface definitions listed in Figure 1. For the student, the simplest way of using the library is to define a subclass of **`XCanvas`** in much the same way that one would ordinarily define a subclass of **`JComponent`** in a Swing application. The **`paint`** method for the extended subclass would then update the image on the screen, just as it does in a standard application. So far, nothing is different from in standard Java except for the change in the name of the base class. Any code that worked using **`JComponent`** should work in the same way using **`XCanvas`**.

There advantages of using **`XCanvas`** are twofold. First, the **`Graphics`** object created by the **`getGraphics`** method of an **`XCanvas`** object is guaranteed to be of type **`XGraphics`**. That means that clients of the package can use the extended capabilities of the **`XGraphics`** class to, for example, adjust the coordinate system, specify coordinates as real numbers, or have the system keep track of the current pen position. Second, an **`XCanvas`** object can operate in a persistent graphics mode in which all drawing operations are recorded in an offscreen memory buffer from which the actual display contents can be refreshed. This strategy is nothing more than double-buffering used to provide persistent images rather than to reduce screen flicker. From the student's point ofview, the effect is that the screen image is automatically maintained. If the window is hidden and then restored, the repaint request restores the contents of the display from the offscreen memory, thereby eliminating the need to maintain data structures representing the image. In our experience, this mechanism has made it possible for students to design sophisticated graphical images from the very beginning of the introductory course.

Figure 1. Classes in the graphics package

`XCanvas`	This class is a subclass of the **`JComponent`** class in the Swing library that implements the persistent bitmap model described in this proposal. The **`getGraphics`** method of an **`XCanvas`** object is guaranteed to return an **`XGraphics`** object.
`XGraphics`	This class represents an extended form of the standard **`java.awt.Graphics`** class that supports the real-valued graphics model described in this proposal. Because it is a **`Graphics`** subclass, it is possible to call all of the standard methods, which are defined to have exactly the same behavior as in the standard class. Most new methods supported by **`XGraphics`** are simply the standard ones redefined to take real-valued arguments. There are, however, several additional methods, as shown in Figure 2.
`XGraphicsModel`	This interface defines the behavior of a class that offers the complete functionality of the **`XGraphics`** class.
`XDimension` **`XPoint`** **`XRectangle`**	These classes are simple equivalents to the standard AWT classes **`Dimension`**, **`Point`**, and **`Rectangle`** whose components are **`double`**s rather than **`int`**s.

References

1. Adobe Systems. PostScript Language Reference Manual. Cupertino, CA: Adobe Press, 1986.
2. Kim B. Bruce, Andrea Danyluk, and Thomas Murtagh. A library to support a graphics-based object-first approach to CS1. Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education, Charlotte, NC, February 2001, pp. 6-10.
3. Seymour Papert. *Mindstorms.* New York: Basic Books, 1980.
4. Eric Roberts. A C-based graphics library for CS1. Proceedings of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education, Nashville, TN, March 1995, pp. 163-167.
5. Eric Roberts and Antoine Picard. Designing a Java graphics library for CS 1. Proceedings of the 3rd Annual Conference on Innovation and Technology in Computer Science Education, Dublin, Ireland, August 1998, pp. 213-218.
6. Eric Roberts. An overview of MiniJava. Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education, Charlotte, NC, February 2001, pp. 1-5.
7. Eric Roberts. Taxonomy of problems in teaching Java. ACM Java Task Force, February 27, 2004.
8. Andreas van Dam, et al. Documentation for the NGP package. **`http://cs.brown.edu/courses/cs015/2002/Docs/NGP`**.

Figure 2. Selected extended methods in XGraphics

`void movePen(double x, double y)`
`void adjustPen(double dx, double dy)`

The **`movePen`** method moves the pen to the specified x and y coordinates; **`adjustPen`** moves the pen using the specified displacements from its current position.

`void translate(double tx, double ty)`
`void scale(double sx, double sy)`
`void setCartesianFlip(boolean flip)`

These methods enable the client to modify the coordinate system by introducing scaling and translation parameters. The **`setCartesianFlip`** method inverts the y-axis to support traditional Cartesian coordinates

---

`void startFilledRegion()`
`void endFilledRegion()`

These methods allow the client to fill an arbitrary shape bounded by lines and arcs. When **`startFilledRegion`** is called, any subsequent calls to **`drawLine`** and **`drawArc`** do not actually affect the display but instead are used to define a polygonal region. When **`endFilledRegion`** is called, the interior of that region is filled with the current color.

`void saveGraphicsState()`
`void restoreGraphicsState()`

This pair of methods saves and restores the graphics state in a stack-like fashion, making it possible for user-defined methods to make changes to the local state and yet easily restore the assumptions of the caller.