Designing a Java Graphics Library for CS1

Eric Roberts
Department of Computer Science
Stanford University
eroberts@cs.stanford.edu

Antoine Picard
Department of Computer Science
Stanford University
picard@cs.stanford.edu

Maria Fredricsson
Department of Computer Science
Stanford University
mariaf@cs.stanford.edu

ABSTRACT

In recent years, there has been considerable interest in using Java in introductory computer science courses. The advantages of choosing Java, however, must be balanced against two significant drawbacks: the instability caused by the rapid evolution of the Java toolkits and the complexity of the graphics model for new students. This paper outlines a strategy for teaching Java that eliminates these problems. The first component of that strategy is a set of low-level classes, **`DBCanvas`** and **`XGraphics`**, which together offer a double-buffered graphics model that is well matched to student intuition. The second component is a set of higher-level classes, principally **`Collage`** and **`Widget`**, which make it easy to introduce object-oriented techniques at the beginning of CS1. The sources for these classes are available on the Web at the URL **`http://cse.stanford.edu/java`**.

1. INTRODUCTION

Since 1992, the introductory computer science courses at Stanford have been taught in ANSI C. Although certain characteristics of C make it difficult to teach at the introductory level, most of the problems can be avoided by using a set of libraries to hide unnecessary complexity [9]. Of those libraries, the most useful is the portable graphics library, which was described at SIGCSE '95 [10] and integrated into *The Art and Science of C,* the textbook used in conjunction with the course [11]. The graphics library makes it possible to hold the interest and enthusiasm of students who have grown up with powerful personal computers and fancy user interfaces.

The C-based graphics library has been highly successful as a pedagogical tool but has led to portability problems. Although there is a single **`graphics.h`** interface common to all platforms, achieving the desired portability has forced us to maintain three separate implementations of the corresponding **`graphics.c`** file—one for X Windows, one for the Macintosh, and one for Windows-based PCs—along with several minor variants to cover different compilers or hardware platforms. Keeping these versions up to date with advances in compiler technology has proven to be a significant challenge.

One strategy that holds promise in terms of increasing portability is to use Java instead of C in CS1. Since Sun Microsystems released the initial version in 1995 [3], Java has generated considerable excitement in colleges and universities, many of which have adopted Java as the basis for their CS1/CS2 sequence [5, 13]. Java has several pedagogical advantages. For one thing, Java is a much cleaner language, which makes it easier to teach at the introductory level. Java's design also includes several features that make it more suitable for introductory students, including an object-oriented data model, automatic garbage collection, and greater type safety. Finally, Java is explicitly designed for use in an interactive, graphical environment. As such, its standard libraries include classes for drawing graphical images on the screen and for assembling user interface components. Because Java already incorporates graphics as part of its standard library, some of the portability problems we encounter using our C-based approach are likely to disappear.

Unfortunately, incorporating Java into the introductory computer science curriculum is not as easy as one might hope. The major stumbling block is instability, not so much in the Java language, but in the associated library packages that define the Java toolkit. Java toolkits continue to evolve at a pace that makes it difficult for textbook authors and introductory level instructors to keep up. The differences, for example, between versions 1.0 and 1.1 of the Java Development Kit (JDK) are considerable and have significant implications for how that material is presented. Future directions are clouded even more by the emergence of competing libraries and the uncertainty as to which of those libraries will be supported by the leading browsers and standalone Java environments.

Beyond the instability issue, Java's graphics model has a number of features that make it harder for beginning students to understand. From the perspective of using Java at the introductory level, the most problematic features are the following:

- "Forgetful bitmaps" and the lack of an automatic double-buffering mechanism
- The relative statelessness of the graphics context
- The use of a resolution-dependent, pixel-based, non-Cartesian coordinate system

Each of these shortcomings is discussed in detail in section 2.

Based on our experience with the semantics of the C graphics libraries, we believe that student comprehension in Java-based introductory courses can be enhanced by introducing a simpler graphics model that is more accessible to students at the introductory level. Section 3 of this paper outlines two new lower-level graphics classes we have developed here at Stanford and shows how these classes can be incorporated into the Java graphics model without introducing unnecessary incompatibility. Section 4 then describes a higher-level approach for teaching graphics in which the traditional drawing metaphor is replaced by one in which the user assembles a collage consisting of graphical objects.

Although we as yet have no direct experience, we believe that using these new classes will make it much easier to teach Java in the introductory computer science curriculum. To test the validity of the concept, we plan to integrate these tools into an upcoming summer-session version of the introductory course, which is much smaller than the course during the academic year and therefore provides an easier context for experimentation. We are publishing the ideas now—in advance of teaching the course—because of the extraordinary pace of change in technology. Given how fast Java

ITiCSE '98 Dublin, Ireland

is evolving, adopting the traditional approach of reporting on past results guarantees that the material is obsolete on publication.

2. PROBLEMS IN JAVA'S GRAPHICS MODEL

As noted in the introduction, the Java graphics model embodied in the **java.awt** package has several features that make it difficult to use at the introductory level. Because many of these features are in fact appropriate for more advanced programmers, they cannot properly be called flaws in the design. On the other hand, they represent barriers to using Java-based graphics in the context of the CS1/CS2 sequence. The most problematic features of the Java graphics model are outlined in the subsections that follow.

2.1 Forgetful bitmaps

As in most object-oriented graphics libraries, the **java.awt** package uses an event-driven paradigm in which graphical displays are represented as "forgetful bitmaps" whose contents must be refreshed in response to events. For example, if you obscure a window containing a Java applet and then re-expose it, the applet is responsible for redisplaying the contents of the window. For simple applets, this process is accomplished by defining a **paint** method that issues the necessary graphics calls. For more complicated applets, the requirement that applets must update the display on demand requires students to maintain a significant amount of state information. Designing the data structure to maintain this state is much more difficult than calling a set of methods to create a static display.

To some extent, the use of forgetful bitmaps follows directly from the event-driven paradigm. Although introductory students can learn to think in an event-driven style, there is evidence to suggest that students prefer to implement graphical applications using a procedurally based approach [15].

To simplify the process of updating the state of the graphical display, the easiest approach is to use a standard technique called *double buffering,* which uses a separate offscreen bitmap for all program-generated changes to the display. Whenever a particular part of the display needs to be updated, it is sufficient to copy the contents of the offscreen bitmap into the bitmap representing the onscreen display.

Unfortunately, implementing double buffering is beyond the scope of most introductory courses. As such, this technique is typically unavailable to students unless it is provided as part of a library package.

2.2 Statelessness of the graphics context

In designing a graphics package, one of the issues the designer must resolve is how much internal state is maintained by the graphics library. The Java **Graphics** class adopts a somewhat inconsistent policy in this regard. Some aspects of the state, such as the current color and font information, are maintained internally as part of each particular **Graphics** context. On the other hand, the **Graphics** class does not keep track of the current pen position. For example, if you want to draw a line, you must pass the absolute coordinates of both endpoints to the **drawLine** method. An alternative design, which is used in our 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.

For novice programmers, it helps to have the graphics library maintain as much internal state as possible. Doing so reduces the number of parameters needed for each call, thereby reducing conceptual complexity for the student. This attitude is reflected in the design of the Turtle Graphics model used in LOGO, in which the graphics library maintains state information representing both position and direction. In his book, *Mindstorms* [8], 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 a CS1 course are no longer young children, they nonetheless can more easily understand the process of a graphics library that maintains a state into which they can project themselves. In writing a graphics program, it helps a great deal for students to be able to reason along the lines of "I'm here now; I need to move up three units and over four."

In addition to the conceptual advantages of a graphics model that includes more state, keeping track of a current point also makes it easier for students to draw the same figure in several different positions on the screen. To draw multiple copies of a given figure, all the student has to do is move the pen to the appropriate starting positions and then call a common routine to draw each figure in a relative mode.

2.3 The Java coordinate system

The coordinate system used in Java, which is similar to that used in other graphics libraries used with bitmapped monitors, mirrors the structure of the hardware. Coordinates are measured using integer values representing the actual pixels on the screen. Particularly when used by students who have no experience with similar coordinate systems in other graphics libraries, the Java approach has the following shortcomings:

- *The coordinate system is resolution-dependent.* Because all coordinate values are expressed in pixels, the size of a displayed figure depends on the size of an individual pixel. A figure drawn on a monitor with 72 pixels per inch will appear much smaller if monitor technology improves to the point that monitors with significantly higher resolution become commonplace.
- *Coordinates must be integral.* For many applications, the fact that all coordinate values must be integers makes it more difficult for students to draw closed figures. 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.
- *The coordinate system has a non-Cartesian origin.* The Java coordinate system has its origin in the upper-left corner, which differs from the Cartesian coordinate system students learn in high school. While many calculations are easy to invert, other geometrical computations, such as those that involve angles, tend to remain confusing.

It is interesting to note that the PostScript® graphics model [1] follows the approach used in the C-based graphics library, which differs from the Java model in each of these respects. It is also clear that some of the new Java toolkits are evolving in a similar direction. The **Graphics2D** class in JDK 1.2 allows for real-valued coordinates with scaling and translation. Unfortunately, many of these new interfaces are more complex than the existing **Graphics** class. Moreover, there is no convincing evidence that the facilities provided by JDK 1.2 will be implemented widely by browsers, particularly given the ongoing battles between Sun and Microsoft.

3. EXTENDING THE GRAPHICS CLASSES

In the summer of 1997, we began to experiment with Java to determine whether it would be suitable as a language for our introductory course. At first, we were reticent to propose any significant extensions to the functionality provided by the classes

in the **java.awt** package. After all, one of the principal advantages of Java is that it already includes as standard many of the features that we had to create for ourselves in C. In the end, however, we concluded that there were compelling reasons to develop new library classes that would allow us to teach Java more effectively at the introductory level.

The primary motivation for introducing new library classes is to avoid the deficiencies enumerated in section 2, particularly the lack of an automatic facility for double buffering. At the same time, we also concluded that—somewhat paradoxically—the enormous instability in Java environments and toolkits encourages the development of custom libraries as a means of maintaining stability. Teaching approaches and textbooks that are tied to the JDK need to track changes to that foundation. Several authors of Java textbooks, for example, were forced to pull their books out of production when JDK 1.1 appeared. By using a more stable set of custom library classes, authors and educators can insulate themselves from the rapid pace of software evolution.

After a few months of experimentation, we developed a prototype system that addresses the problems in the **java.awt** model without creating a transition barrier for students who go on to work with the published versions of the JDK. At the implementation level, the foundation of our facility consists of two new classes that extend the basic functionality of the windowing toolkit. The first is the **DBCanvas** class, a subclass of the standard **Canvas** component that automatically offers double buffering to create a persistent bitmap. The second is the **XGraphics** class, which provides a set of operations similar to that provided by **java.awt.Graphics**, but with a variety of extensions that address the problems outlined in section 2.

The constructor for the **DBCanvas** class takes the dimensions of the canvas and allocates the necessary offscreen bitmap. The **paint** method for the class simply copies the offscreen bitmap onto the display. Graphics operations for the **DBCanvas** class are accomplished by making calls on an **XGraphics** object associated with each **DBCanvas** instance.

By default, the coordinate system used in the **XGraphics** class is compatible with the Java standard. The origin is in the upper left corner of the display; coordinate values are measured in pixel units. This design ensures that images displayed using the **XGraphics** model match the Java standard. The coordinate system, however, in fact uses floating-point coordinates that can be scaled, translated, and flipped to obtain a coordinate system that matches the one used by the C-based graphics libraries.

The **XGraphics** class also implements the notion of a current point, but uses overloading to maintain consistency with the method suite supplied by the **Graphics** class. The **drawLine** method, for example, can be invoked with four arguments on an **XGraphics** context **g** so that

```
g.drawLine(100, 100, 200, 100);
```

draws a 100-point horizontal line extending to the right of the coordinates (100, 100), which is precisely what the Java programmer expects. The same effect can be achieved by the statements

```
g.movePen(100, 100);
g.drawLine(100, 0);
```

When called with two arguments, **drawLine** draws a line relative to the current point. By using this design, the **XGraphics** class preserves compatibility with the standard Java model while allowing instructors to use a graphical paradigm that is easier for introductory students to master.

4. THE COLLAGE MODEL

Traditional graphical libraries, including both the **java.awt** package and the **DBCanvas**/**XGraphics** extensions described in section 3, use a drawing metaphor in which users construct a picture by moving a virtual pen on the display surface. Such a model is inherently procedural in its structure. There is, of course, a **Graphics** object passed to the **paint** method of every graphical component that is the target of every method, but the generation of the display nonetheless involves a series of procedure calls to create the individual parts of the complete image. When one is teaching an object-oriented language like Java, it seems retrogressive—particularly when introducing a topic like graphics that is certain to excite student interest—to adopt a model that does not also serve to illustrate the object-oriented paradigm.

As an alternative to the traditional drawing metaphor, we have developed a set of classes in which graphical displays have more the character of a collage. The complete applet—or standalone application, since the model supports both styles—is an object of the class **Collage**, which is analogous to an empty felt board. To generate a display, the student creates new graphical objects called *widgets,* each of which represents an object that can be added to the felt board. The **Widget** class encompasses a variety of predefined subclasses, including **LineWidget**, **ArcWidget**, **BoxWidget**, **OvalWidget**, **PolygonWidget**, **TextWidget**, and **ImageWidget**.

As a simple example of how students would use the collage model to create a graphical application, the following code implements the classic "Hello world" collage:

```
public class HelloCollage extends Collage
{
   public void main()
   {
      add(new TextWidget("Hello world"),
          100, 100);
   }
}
```

which adds a text widget to the collage. Coordinates in the collage use the model from the **XGraphics** class. By default, coordinate value match that used in the **java.awt.Graphics** class. The user, however, can adjust the coordinate system to match any other model. For example, the **collage** package includes a class called **CartesianCollage**, which is identical to **Collage** except that the coordinate system is measured in inches from an origin in the lower left.

Once a widget has been created, the student can manipulate the widget using a set of methods that control the positioning and appearance of the widget. A list of some of the more important methods that apply to widgets is shown in Figure 1.

The following collage code uses the methods in Figure 1 along with the **pause** method from the **Collage** class to animate a red square moving diagonally across the screen from the standard Java origin in the upper left:

moveAbsolute(*x***, *y*);**	**setColor(***color***);**
moveRelative(*dx***, *dy*);**	***color* = getColor();**
moveToFront();	**setFont(***font***);**
moveToBack();	***font* = getFont();**
add(*widget***);**	**setFill(***fill***);**
add(*widget***, *x*, *y*);**	***fill* = getFill();**
remove(*widget***);**	

Figure 1. Selected Widget Methods

```
public class MovingBoxCollage extends Collage
{
   public void main()
   {
      BoxWidget box = new BoxWidget(25, 25);

      box.setColor(Color.red);
      add(box, 0, 0);
      for (int i = 0; i < 100; i++) {
         pause(0.2);
         box.moveRelative(1, 1);
      }
   }
}
```

One of the most important characteristics of the **Widget** class is that it is a container class that allows students to nest widgets inside other widgets. For example, the following widget definition creates a widget whose appearance is a box with a enclosed cross consisting of the two main diagonals:

```
public class CrossedBoxWidget extends Widget
{
   public CrossedBoxWidget(double w, double h)
   {
      add(new BoxWidget(w, h));
      add(new LineWidget(w, h));
      add(new LineWidget(w, -h), 0, h);
   }
}
```

Widgets are rendered by invoking their **draw** method and then calling the **draw** methods of all the widgets they contain. Thus, an instance of the class **CrossedBoxWidget** would execute the default **draw** method for the **Widget** class itself and then display the three interior widgets that compose the class, generating a figure like this:

Programs can manipulate **CrossedBoxWidget** objects as a unit. For example, if you replace the instances of **BoxWidget** in **MovingBoxCollage** with **CrossedBoxWidget**, the resulting applet will animate the motion of the crossed box.

The principal advantages of using the **collage** package are the following:

• *Students can learn object-oriented strategies early*. The collage model with nested widgets is simple enough to be presented at the very beginning of CS1. In doing so, students can master the concepts of inheritance and containment before they are overwhelmed by other programming concepts.

• *Students can use an intuitive coordinate system*. By using the **XGraphics** facility in its implementation, the collage model supports the use of a flexible coordinate system that allows students to use either the pixel-based model used by Java or a Cartesian coordinate system, which is familiar to most students.

• *The need for double-buffering goes away*. Because the **Collage** object keeps track of its internal widgets as part of its data structure, there is no need to maintain a separate screen image for refreshes. Double-buffering may still be used to avoid flicker, particularly when the display contains images.

• *The implementation is surprisingly accessible*. The complete implementation of the **Collage** and **Widget** classes can easily be explained to students while they are still enrolled in CS1, mostly because the model avoids the complexity of double-buffering.

• *Users are insulated from changes in the JDK*. The interface presented by the **Collage** and **Widget** classes does not depend on the details of the **java.awt** package. Courses that are designed on top of this platform are less susceptible to the evolution of the underlying libraries.

5. FUTURE DIRECTIONS

We are currently working to develop a large set of tools predicated on the base provided by the various extended classes described in this paper. In particular, we are currently in the process of implementing a variety of Java-based tools to facilitate the creation of demonstration programs specifically aimed at CS1 and CS2 education [12]. Developing these tools in Java is critical to encourage students to review some of the more conceptually difficult demonstrations and algorithmic animations on their own, from whatever computing environment is available to them.

In addition to using **XGraphics.java** to display graphics, our demonstration tools also provide special facilities to ease conceptual difficulties involved in computer graphics. Specifically, we have created tools that display the graphics library's internal state on screen—particularly the pen location and internal control state—to allow both instructors and students to step through their graphical code and see, step-by-step, the creation of their image.

A web site including the classes described in this paper is available at the URL **http://cse.stanford.edu/java**. We encourage you to experiment with these classes at your own institutions and welcome any feedback you have to offer.

REFERENCES

1. Adobe Systems, Inc. *PostScript Language Reference Manual* Reading, MA: Addison-Wesley, 1985.
2. David Flanagan. *Java in a Nutshell*. Cambridge: O'Reilly, 1997.
3. James Gosling and Henry McGilton. *The Java Language Environment: A White Paper* Sun Microsystems, May 1996. URL: **http://java.sun.com/docs/white/langenv/**. O'Reilly, 1997.
4. Cay Horstmann. *Computing Concepts in Java*. New York: John Wiley and Sons, 1997.
5. Frederick Hosch. Java as a first language: An evaluation. *SIGCSE Bulletin* (September 1996), 45–54.
6. J. N. Patterson Hume and Christine Stephenson. *Programming Concepts in Java*. Toronto: Holt Software Associates, 1998.
7. David Mutchler and Cary Laxer. Using multimedia and GUI programming in CS1. *Integrating Technology into Computer Science Education* (Barcelona, June 1996), 63–65.
8. Seymour Papert. *Mindstorms*. New York: Basic Books, 1980.
9. Eric Roberts. Using C in CS1: Evaluating the Stanford experience. *SIGCSE Bulletin* (March 1993), 117–121.
10. Eric Roberts. A C-Based Graphics Library for CS1. *SIGCSE Bulletin* (March 1995), 163–167.
11. Eric Roberts. *The Art and Science of C*. Reading, MA: Addison-Wesley, 1995.
12. Eric Roberts. Tools for creating portable demonstration programs. *Integrating Technology into Computer Science Education* (Barcelona, June 1996), 78–80.
13. Mark Weiss. Experience teaching data structure with Java. *SIGCSE Bulletin* (March 1997), 164–168.
14. Frank Wester. Visual programming with Java; an alternative approach to introductory programming. *SIGCSE Bulletin* (September 1997), 57–58.
15. Ursula Wolz, Scott Weisgarber, Daniel Domen, and Michael McAuliffe. Teaching introductory programming in the multimedia world. *Integrating Technology into Computer Science Education* (Barcelona, June 1996), 57–59.