A Portable Graphics Library for Introductory CS

Eric Roberts
Stanford University
Department of Computer Science
Stanford, CA 94305
+1 650-723-3642
eroberts@cs.stanford.edu

Keith Schwarz
Stanford University
Department of Computer Science
Stanford, CA 94305
+1 650-723-4350
htiek@cs.stanford.edu

ABSTRACT
For several decades, instructors who focus on introductory computer science courses have recognized the value of graphical examples. Supporting a graphics library that is appropriate for beginning students has become more difficult over time. This paper describes a new approach to building a graphics library that allows for multiple source languages and a wide range of target architectures and platforms. The key to this approach is using an interprocess pipe to communicate between a platform-independent client library and a Java-based process to perform the graphical operations specific to each platform.

Categories and Subject Descriptors
K.3.2 [**Computer and Information Science Education**]: computer science education.

General Terms
None.

Keywords
Graphics, libraries, CS1, CS2.

1. INTRODUCTION
Ever since Seymour Papert began teaching students to program using the Project Logo turtle in the 1960s [5], computer science instructors have recognized the importance of using graphics in introductory computer science courses and, more recently, in more general courses that teach computational thinking to a broader audience. The value of a graphics-based approach is reflected in the popularity of programming microworlds such as Alice [3] and Scratch [9] and the success of the media computation model developed by Mark Guzdial and Barbara Ericson [4, 5]. In 2000, Richard Rasala went so far as to describe graphical toolkits as a "pedagogical imperative" for introductory computer science courses [8].

Unfortunately, graphics libraries suitable for introductory courses have become harder to develop and maintain. For one thing, the graphics packages that come packaged with a particular platform or development environment have become so complex that they are difficult for most intermediate students, let alone for novices. In addition, graphics packages tend to evolve so quickly that it is nearly impossible for universities and colleges—most of which lack budgets to maintain such tools—to keep up. Finally, the proliferation of introductory languages and programming environments means that each specific package serves a smaller audience.

This paper describes a new approach to building a graphics library that eliminates most of the maintenance problems by shifting all platform-specific operations to a separate process implemented in Java, which is widely supported on the platforms that students are likely to use. Moreover, by using a text-based interprocess pipe to communicate with the Java process, the client side of the application can use any programming language that supports the creation of processes and pipes. In addition to more traditional systems-programming languages like C and C++, the necessary capabilities are available in such languages as Python, Ruby, and Perl. The new architecture therefore makes it possible to design compatible, easily maintained libraries that work with any of these languages without having to write any of the complex rendering code that leads to the maintenance nightmare.

2. THE QUICKDRAW MODEL
The idea of using an interprocess pipe to drive a Java "back-end" process was originally suggested by Ben Stephenson and Craig Taube-Schock at the University of Calgary, who incorporated that strategy into their QuickDraw system [14, 15]. The QuickDraw library (not to be confused with the original graphics framework for the Apple Macintosh, which has the same name) allows students to create static graphical images by writing a program that composes and sends graphics commands to the QuickDraw application. The program itself can be written in any language; all it needs to do is write the necessary commands to the standard output channel. To see the graphical output, the student uses a command-line interpreter to pipe that output to the Java process, using a command that looks something like this:

```
StudentApplication.exe | java -jar QuickDraw.jar
```

The commands themselves are simple text strings. For example, to draw a circle of radius 50 centered at the point (100, 100), the student application would print the line

```
circle 100 100 50
```

The package described in this paper extends this model by adding a "thin client" layer to the application that transforms high-level graphical operations into the necessary commands. As the later sections of this paper describe, adding this layer makes it possible for students to write highly interactive applications using libraries whose structure matches the style of the language they are using.

ITiCSE'13, July 1–3, 2013, Canterbury, England, UK.

3. GRAPHICS LIBRARIES AT STANFORD

At Stanford, we decided long ago that students should be able to write their programs using the computers they bring to campus. In part, the motivation for this decision is practical: maintaining a dedicated cluster of computers to support the introductory courses would be prohibitively expensive given the number of students involved. Having students develop code using their own machines also increases the likelihood that they will undertake independent projects outside of class, which supports a culture of creativity and experimentation.

The problem with letting students use their own computers is that doing so forces us to support several different platforms in our introductory courses. Some students come to campus with Macintoshes, others bring Windows machines, and a few have Linux workstations. The languages we use in our introductory courses run on all these machines, but the APIs available for implementing graphics differ substantially from platform to platform. And while there are a few industry-standard packages like OpenGL that are supported across multiple platforms, the complexity of those packages puts them out of the range of novices.

When Stanford shifted its introductory course from Pascal to C in the early 1990s, we adopted the strategy shown in Figure 1 to achieve the necessary level of multiplatform support. Applications that needed to use graphics made calls to functions in a common interface that was then separately implemented for each of the supported platforms.

As is typical for software, the implementations for each platform grew over time as new capabilities were incorporated into the interface. The individual implementations, moreover, have needed significant reengineering as the graphics frameworks on which they are based evolve. As a result, the approach shown in Figure 1 has required considerable effort to maintain.

In the early 2000s, we again changed the structure of our introductory courses, shifting the implementation language from C to Java. Although Java provides the desired level of platform independence, we found—as did many other educators who made the same transition—that the graphics capabilities provided by the standard Java APIs were difficult for novices to use. We were also unhappy with the fact that the standard APIs do not use an object-oriented approach despite Java's stated design goal of supporting an object-oriented programming paradigm. Faced with these problems, we experimented with a set of locally designed libraries that proved more effective in the classroom [11]. After refinement over a series of years, these libraries were incorporated into the **`acm.graphics`** package developed by the ACM Java Task Force [1].

Although we have been happy with the **`acm.graphics`** package and the portability it provides, students have not been able to use that model after they move beyond our CS1 course. Our version of CS2 uses C++ as its implementation language, which has heretofore offered limited graphics support using the strategy from Figure 1. When we assign graphical applications in CS2 and more advanced courses, we tend to provide assignment-specific code written by the instructors. Giving students access to a graphics model of the sort they had in CS1 would substantially increase both their capabilities and their excitement level, but the problems of implementing a multiplatform approach have kept us from opening up those tools to student use.

We believe that the new graphics model shown in Figure 2 solves this problem in an exciting way. In essence, the solution combines the interprocess channel from the Calgary QuickDraw approach with the object-oriented graphics model from the **`acm.graphics`** package. The end result offers the portability advantages of QuickDraw along with the flexibility and power of the Java-based environment that has proven to be so successful in our CS1 course.

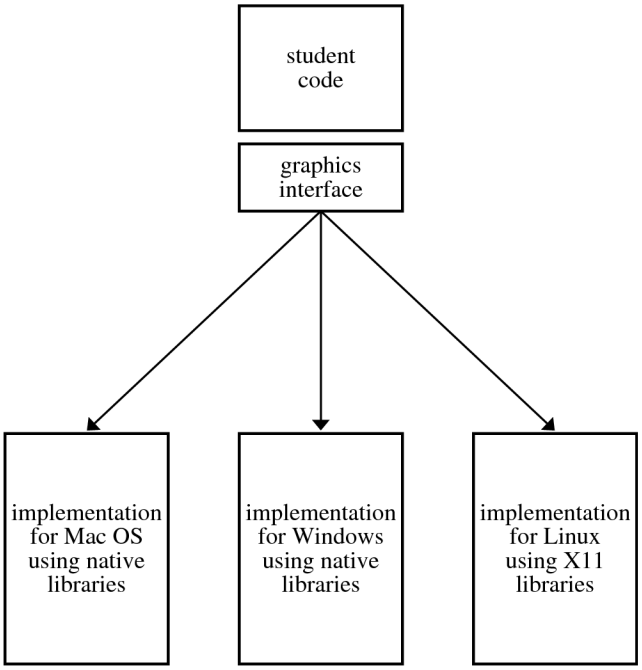

Figure 1. Old multiplatform graphics model

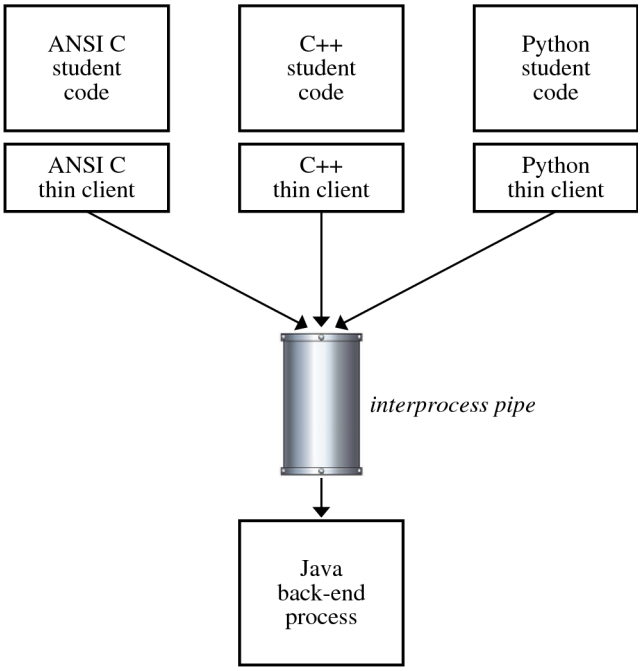

Figure 2. New multilanguage/multiplatform model

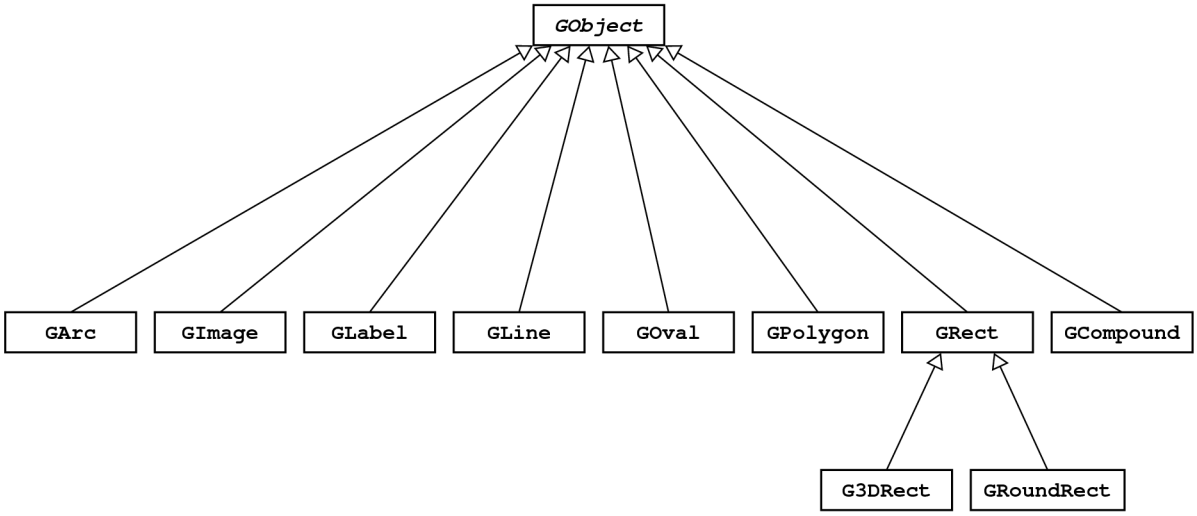

Figure 3. Graphical object hierarchy used in the portable library

4. THE GRAPHICAL OBJECT MODEL

As in the **acm.graphics** package on which it is based, the graphics model used in the portable graphics library defines a set of classes that represent graphical objects. These classes form the hierarchy that appears in Figure 3. The root of the hierarchy is the abstract **GObject** class, which implements the methods that are common to all objects, such as setting the position and color. Each of the concrete subclasses represents a different type of graphical object, each of which inherits the behavior of the **GObject** class while supporting additional methods appropriate to that specific type. The **GLabel** class, for example, exports methods for setting the font and for determining the width of the displayed text. The **GLine** class, by contrast, exports methods for adjusting the endpoints of the line segment. The **G3DRect** and **GRoundRect** classes are subclasses of **GRect** and therefore inherit methods from that class. These classes are used infrequently in practice and are included to illustrate how class hierarchies work.

The methods available for the classes in Figure 3 are described in the online documentation for the package. While the list of available methods is too extensive to describe in detail, the general idea is easy to illustrate by example. Figure 4 shows the code to draw the international symbol for a "DO NOT ENTER" sign, along with the screen image it produces. The code, which for this example appears in C++, begins by creating a small **GWindow** object that serves as the canvas to which the various **GObject** instances are added. For this figure, the only objects are the red circle and the superimposed white bar, which are added at explicit coordinate positions. The calls to the methods **setFilled** and **setColor** indicate that these objects should be filled rather than outlined and give each its appropriate color. The last statements before the program returns add the circle and the rectangular bar to the **GWindow**, at which point they appear on the screen. The graphics library keeps track of the order of the objects, so that the bar appears on top of the circle.

```
/*
 * File: DoNotEnter.cpp
 * --------------------
 * This program draws the international "DO NOT ENTER"
 * symbol in the center of the graphics window.
 */

#include "gobjects.h"
#include "gwindow.h"
using namespace std;

int main() {
   GWindow gw(250, 250);
   GOval *circle = new GOval(50, 50, 150, 150);
   circle->setFilled(true);
   circle->setColor("RED");
   GRect *bar = new GRect(75, 110, 100, 30);
   bar->setFilled(true);
   bar->setColor("WHITE");
   gw.add(circle);
   gw.add(bar);
   return 0;
}
```

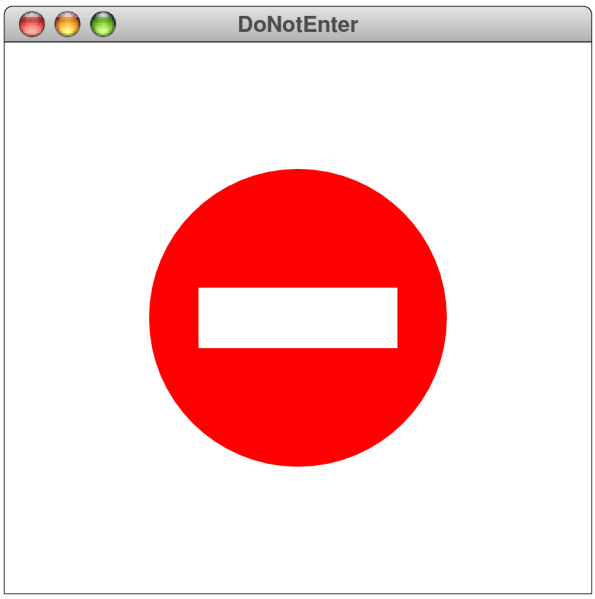

Figure 4. Code for the `DoNotEnter` program and the screen image it produces

5. IMPLEMENTATION STRATEGY

To understand how the portable library uses the interprocess pipe to implement the graphics operation, it helps to consider a slightly more extensive example program, which is interesting in its own right. The screen image to the right in Figure 5 appears to contain a white square framed by the cutouts in the arcs. This square, however, is an optical illusion consisting of a *subjective contour,* as defined by Gaetano Kanizsa in *Scientific American* in 1976 [6].

When compiled using the portable graphics library, the program shown at the left side of Figure 6 automatically launches the back-end process at the beginning of execution. As the program runs and makes graphics calls, it updates its own internal state and sends messages to the back-end to perform the necessary rendering operations. The specific messages this program sends are shown at the right side of Figure 6.

Tracing through the program might make it easier to understand what these commands do. The first line of the program creates a **`GWindow`** whose client area has size 250 pixels by 250 pixels. Creating a **`GWindow`** implicitly creates a **`GCompound`** object, which holds all the objects added to the window.

The seemingly mysterious hexadecimal strings in the transcript are simply unique identifiers for the various objects. In this transcript, for example, the **`GWindow`** object has the identifier **`0x1001d0`** and the top-level **`GCompound`** object has the identifier **`0x1002e0`**. These identifiers can be any strings that uniquely represent the argument. In most cases, they are chosen to be the addresses of the object on the client side.

There are several things to note about the transcript in Figure 6. Most importantly—in contrast to the model in QuickDraw—the student does not ordinarily see these commands, which pass over a pipe allocated by the implementation.

It is also useful to observe that all communication in this example is unidirectional. The front-end makes requests of the back-end, never the other way around. When the front-end needs to determine the size of the window by calling **`gw.getHeight()`**, it retrieves that information from its own cache of information. As a result, no data must be requested and retrieved from the back-end. Using this approach reduces the need for the two processes to exchange information, which turns out to be essential in making the framework efficient enough to use for animation.

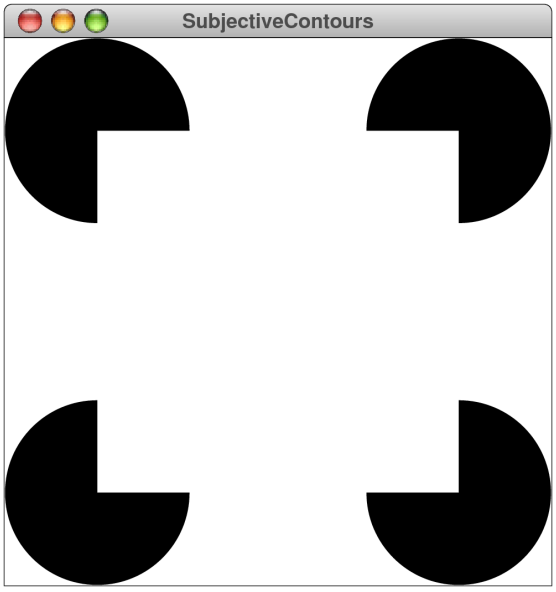

Figure 5. Example illustrating subjective contours

Finally, it is essential to note that *any* program that maintains the same internal state and issues the same commands to the back-end will produce exactly the same output as this program. That program need not have been written in C++; it could just as easily have been written in C, Java, Python, Perl, Haskell, or even assembly language. The only platform-dependent code that must be written in the host programming language is the logic to start the back-end process and maintain the communication channel, which is straightforward to accomplish in any of these languages.

```
#include "gobjects.h"
#include "gwindow.h"
using namespace std;

int main() {
   GWindow gw(250, 250);
   double cSize = 100;
   GArc *nw = new GArc(0, 0, cSize, cSize,
                       0, 270);
   nw->setFilled(true);
   gw.add(nw);
   GArc *ne = new GArc(gw.getWidth() - cSize, 0,
                       cSize, cSize, 270, 270);
   ne->setFilled(true);
   gw.add(ne);
   GArc *se = new GArc(gw.getWidth() - cSize,
                       gw.getHeight() - cSize,
                       cSize, cSize, 180, 270);
   se->setFilled(true);
   gw.add(se);
   GArc *sw = new GArc(0, gw.getHeight() - cSize,
                       cSize, cSize, 90, 270);
   sw->setFilled(true);
   gw.add(sw);
   return 0;
}
```

Transcript of the interprocess channel

```
→ GCompound.create("0x1002e0")
→ GWindow.create("0x1001d0", 250, 250, "0x1002e0")
→ GArc.create("0x100d90", 100, 100, 0, 270)
→ GObject.setLocation("0x100d90", 0, 0)
→ GObject.setFilled("0x100d90", true)
→ GCompound.add("0x1002e0", "0x100d90")
→ GArc.create("0x100e20", 100, 100, 270, 270)
→ GObject.setLocation("0x100e20", 150, 0)
→ GObject.setFilled("0x100e20", true)
→ GCompound.add("0x1002e0", "0x100e20")
→ GArc.create("0x100e70", 100, 100, 180, 270)
→ GObject.setLocation("0x100e70", 150, 150)

→ GObject.setFilled("0x100e70", true)
→ GCompound.add("0x1002e0", "0x100e70")
→ GArc.create("0x100ec0", 100, 100, 90, 270)
→ GObject.setLocation("0x100ec0", 0, 150)
→ GObject.setFilled("0x100ec0", true)
→ GCompound.add("0x1002e0", "0x100ec0")
```

Figure 6. Code for the `SubjectiveContour` program and a trace of the method calls sent through the interprocess pipe

6. IMPLEMENTING INTERACTIVITY

Up to this point, the examples in this paper could just as easily have been implemented using the unidirectional pipe provided by the Calgary QuickDraw system. The real power of the portable graphics library is that the interprocess communication *can* operate in both directions when doing so is necessary to respond to events that happen in the graphics window. The library code implements a rich set of events that includes mouse actions such as clicks and drags, keyboard actions such as key presses and releases, window events such as resizing, and events generated by interval timers that can be used to support animation.

As an example, both versions of the code in Figure 7 implement a simple line-drawing program. Depressing the mouse button generates a `MOUSE_PRESSED` event, which causes the code to create a zero-length line at the current mouse location. Dragging the mouse then generates a series of `MOUSE_DRAGGED` events, each of which updates the end point of the line—but not its starting point—to the new mouse coordinates. The effect is to allow the user to drag the line to its correct position while getting the necessary visual feedback. In computer graphics, this style of line drawing is called *rubber banding* because the mouse appears to be connected to its starting point by an elastic line.

We have used the new graphics library to reimplement all the animated applications we use in our CS1 and CS2 courses, and the performance is entirely acceptable despite the use of text-based interprocess communication that requires parsing and unparsing commands. In particular, the strategy works perfectly for video games like Breakout [13], which we use as the second Java assignment in CS1. The time required to render the graphical display on the Java side easily dominates the cost involved in communicating information across the pipe.

It is, however, important to note that achieving this performance depends on maintaining a mathematical model of the graphical objects on *both* sides of the pipe. When the Breakout application needs to determine whether the ball has hit any of the bricks, it must do so entirely on the client side without requesting that information from the back-end process. Doing so forces the two processes to complete a round-trip exchange consisting of a request and the corresponding response. Requiring that handshake in the exchange of data makes the library far too slow to use in practical applications.

7. MAINTAINING PORTABILITY

Even though the strategy of using the interprocess pipe makes it possible to implement the portable library across a range of languages, the calls that the application makes to the client side of the library must adapt to fit the structure and syntax of the implementation language.

One of the goals of the portable library project is to ensure that the discipline of using the libraries changes as little as possible. The purpose of including both the C and the C++ versions of the line-drawing program in Figure 7 is to illustrate how closely aligned the two versions can be, even when one language supports the object-oriented paradigm and one does not. In the C version, the data structures appear as abstract data types (ADTs), which are implemented in C as pointers to opaque structures. All operations on these structures are implemented as functions that take the ADT pointer as the first argument. The C++ version, by contrast, uses traditional objects and method calls. These differences, however, are decidedly in the *accidental* rather than the *essential* category. The similarity of the graphics model makes the process of translating an application from one language to another almost entirely a mechanical process.

```c
/*
 * File: DrawLines.c
 * -----------------
 * This program allows users to draw lines on the
 * window by clicking and dragging with the mouse.
 */

#include "cslib.h"
#include "gevents.h"
#include "gobjects.h"
#include "gwindow.h"

int main() {
   GWindow gw = newGWindow(600, 400);
   GLine line;
   while (true) {
      GMouseEvent e = waitForEvent(MOUSE_EVENT);
      EventType type = getEventType(e);
      if (type == MOUSE_PRESSED) {
         line = newGLine(getX(e), getY(e),
                         getX(e), getY(e));
         add(gw, line);
      } else if (type == MOUSE_DRAGGED) {
         setEndPoint(line, getX(e), getY(e));
      }
   }
}
```

```c
/*
 * File: DrawLines.cpp
 * -------------------
 * This program allows users to draw lines on the
 * window by clicking and dragging with the mouse.
 */

#include "gevents.h"
#include "gobjects.h"
#include "gwindow.h"
using namespace std;

int main() {
   GWindow gw(600, 400);
   GLine *line;
   while (true) {
      GMouseEvent e = waitForEvent(MOUSE_EVENT);
      EventType type = e.getEventType();
      if (type == MOUSE_PRESSED) {
         line = new GLine(e.getX(), e.getY(),
                          e.getX(), e.getY());
         gw.add(line);
      } else if (type == MOUSE_DRAGGED) {
         line->setEndPoint(e.getX(), e.getY());
      }
   }
}
```

Figure 7. Programs in C and C++ that let the user draw lines on the window by dragging with the mouse

8. CONCLUSIONS

We believe that the portable graphics library has tremendous advantages over our earlier strategies for achieving portability. In particular, this new approach offers the following benefits:

(1) ***Simplified maintenance.*** Responsibility for ensuring that rendering works on all platforms no longer falls on the developers and adopters. As Java's new owner, the Oracle corporation makes sure that Java runs on each new platform and that it keeps abreast of changes in the native APIs.

(2) ***Streamlined migration paths to new languages and platforms.*** Good assignments take a long time to develop. One of the biggest impediments to adopting new languages is the cost of translating existing assignments to a new paradigm. Having a common model implemented for a variety of source languages reduces that barrier considerably.

(3) ***Enhanced opportunities to analyze trace data.*** The text stream that passes between the client code and the back-end process provides useful information that has not before been available in the past. Applying machine learning techniques to these traces may make it possible to provide students with automatic advice as to whether they are on the right track.

(4) ***Substantial possibilities for interesting extensions.*** The strategy of combining a thin client with an interprocess pipe has many applications beyond graphics. In its current implementation, the portable library already includes support for interactors (buttons, sliders, text fields, and the like) and dialog boxes for selecting files. Java implements these features, and it is easy to pass those capabilities along through the interprocess pipe. We are considering further extensions along these same lines including methods for reading network data and for manipulating sound and video.

We hope that other educators consider adopting this model in their own courses and that they join with us to develop its potential. We welcome your comments and suggestions.

9. REFERENCES

[1] ACM Java Task Force 2006. *Project Rationale.* New York, NY: Association for Computing Machinery, August 25, 2006. DOI: http://jtf.acm.org/rationale/rationale.pdf.

[2] Owen Astrachan and Susan Rodger 1998. Animation, visualization, and interaction in CS1 assignments. *Proceedings of the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education* (Atlanta, GA, February 27-March 1, 1998), 317-321. DOI: http://doi.acm.org/10.1145/274790.274321.

[3] Stephen Cooper, Wanda Dann, and Randy Pausch 2000. Alice: A 3-D tool for introductory programming concepts. *Proceedings of the Fifth Annual CCSC Northeastern Conference* (Ramapo, NJ, April 28-29, 2000), 107-116. URL: http://www.stanford.edu/~coopers/alice/ccscne00.pdf.

[4] Mark Guzdial 2003. A media-computation course for non-majors. *Proceedings of the Eighth Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Thessaloniki, Greece, June 30-July 2, 2003), 104-108. DOI: http://doi.acm.org/10.1145/961290.961542.

[5] Mark Guzdial and Barbara Ericson 2012. *Introduction to Computing and Programming in Python.* Upper Saddle River, NJ: Prentice Hall, 2012.

[6] Gaetano Kanizsa 1976. Subjective contours. *Scientific American,* 232, April 1976, 48-52.

[7] Seymour Papert 1980. *Mindstorms: Children, Computers, and Powerful Ideas.* New York, NY: Basic Books, 1980.

[8] Richard Rasala 2000. Toolkits in first-year computer science: a pedagogical imperative. *Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education* (Austin, TX, March 8-12, 2000), 185-191. DOI: http://doi.acm.org/10.1145/330908.331852.

[9] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai 2009. Scratch: Programming for all. *Communications of the ACM,* 52:11, November 2009, 60-67. DOI: http://doi.acm.org/10.1145/1592761.1592779.

[10] Eric Roberts 1995. A C-based graphics library for CS1. *Proceedings of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education* (Nashville, TN, March 2-4, 1995), 185-191. DOI: http://doi.acm.org/10.1145/199688.199767.

[11] Eric Roberts, Antoine Picard, and Maria Fredricsson 1998. Designing a Java graphics library. *Proceedings of the SIGCSE/SIGCUE Joint Conference on Integrating Technology in Computer Science Education* (Dublin, Ireland, August 18-21, 1998), 213-218. DOI: http://doi.acm.org/10.1145/290320.283129.

[12] Eric Roberts 2004. The dream of a common language: The search for simplicity and stability in computer science education. *Proceedings of the Thirty-fifth SIGCSE Technical Symposium on Computer Science Education* (Norfolk, VA, March 3-7, 2004), 115-119. DOI: http://doi.acm.org/10.1145/1028174.971343.

[13] Eric Roberts 2006. Nifty assignments: Breakout. *Proceedings of the Thirty-seventh SIGCSE Technical Symposium on Computer Science Education* (Houston, TX, March 3-5, 2006), 562-563. URL: http://nifty.stanford.edu/2006/roberts-Breakout/.

[14] Ben Stephenson and Craig Taube-Schock. 2009. QuickDraw: Bringing graphics into the first year. *Proceedings of the Fortieth SIGCSE Technical Symposium on Computer Science Education* (Chattanooga, TN, March 4-7, 2009), 211-215. DOI: http://doi.acm.org/10.1145/1539024.1508946.

[15] Ben Stephenson 2009. Using Python and QuickDraw to foster student engagement in CS1. *Proceedings of the Twenty-fourth ACM SIGPLAN conference companion on object-oriented programming systems, languages, and applications* (Orlando, FL, October 25-29, 2009), 675-682. DOI: http://doi.acm.org/10.1145/1639950.1639964