A C-Based Graphics Library for CS1

Eric S. Roberts
Department of Computer Science
Stanford University

NOTE

This paper has been submitted to the Twenty-sixth SIGCSE Technical
Symposium on Computer Science Education. Copyright is retained by the
author prior to publication release.

ABSTRACT

This paper describes a simple graphics library designed for a CS1 course
using ANSI C as its programming language. The library can be
implemented easily on a variety of hardware platforms, providing a
reasonable level of portability. Implementations currently exist for
compilers on the Apple Macintosh, the IBM PC, and Unix workstations; the
source code for each of these implementations is publicly available by
anonymous FTP from the Roberts.C.CS1 area on host aw.com. In addition,
the public distribution includes a fully standard implementation that
generates a PostScript representation of the graphical image.

1. INTRODUCTION

Getting college students excited about introductory programming is
harder in the 1990s than it was a decade ago. Those who have grown up
with personal computers have strong preconceptions about what
constitutes an "interesting" application. Modern technology has raised
the ante. The simple text-based applications that are the mainstay of
most computer science textbooks offer little excitement to the student
who is used to modern graphical interfaces. A program that sorts a list
of numbers -- as interesting as that problem may be from an algorithmic
perspective -- does not impress students who have played computer games
for years.

Using a graphics library in the introductory course makes an enormous
difference. Even when the homework problems are harder than those
assigned in a more traditional course, students approach them with
considerably more enthusiasm, which in turn enables them to accomplish
more. Captivated by the allure of computer graphics, beginning students
have been able to develop extraordinarily sophisticated programs. I
have had students complete 1000-line programs in the fourth week of the
introductory course.

Beyond fostering excitement, using a graphics library has other
pedagogical advantages. In my experience, requiring students to write
graphical applications early in the term turns out to be the most
effective way to drive home the concepts of parameter passing and
stepwise refinement. In constructing graphical figures, students
immediately recognize the importance of parameters as they ask
themselves, for example, where to put a line or how big to make a
circle. Moreover, creating the program structure necessary to display a
graphical image encourages students to master the technique of
decomposition. Because the graphics library itself provides only a
small set of primitive operations such as drawing a straight line, the
student quickly learns to assemble lines into rectangles and other

composite figures. As these figures are combined to form still larger
structures, the organization of the image suggests a natural
decomposition strategy for the program as a whole.

Although these advantages give computer science instructors a strong
incentive to integrate graphics into the introductory programming
course, doing so is complicated by the following factors:

   o Existing graphics packages are usually designed for a specific
      implementation platform, which includes not only the
      underlying hardware but also the operating system, the
      compiler, and the runtime libraries. An IBM PC program that
      uses, for example, a graphics package designed for Microsoft
      Windows will not run on the same machine under DOS, much less
      on a Macintosh or a Unix system. Because of this high level
      of system dependence, the instructor who wants to use graphics
      must choose a particular platform and limit students to the
      graphical facilities available in that environment.

   o Because there is no single graphical standard that works for
      all machines, textbooks written to attract the widest possible
      audience usually avoid the issue by omitting any discussion of
      graphics. The instructor is therefore forced to develop
      supplementary course materials that explain how to use the
      local graphics facilities.

   o Most graphics libraries are designed for use by experts, not
      introductory students. As a result, those libraries are often
      too complex for novice programmers to comprehend.

Three years ago, when we started to convert Stanford's introductory
course from Pascal to C, we had to face these problems directly. The
THINK Pascal system we had been using included a simple graphics library
that was well designed for use by beginning students. Unfortunately,
the THINK C compiler we chose for the redesigned course offered no
corresponding facility. Because our experience had convinced us that
using graphics strengthens the introductory course, we decided to design
our own C-based graphics library and integrate it with the instructional
materials being developed for the course.

The remainder of this paper provides an overview of the graphics library
and the strategies used to implement it on a diverse set of programming
platforms.

2. INTERFACE DESIGN CRITERIA

The first step in developing the graphics library was to define the
interface. In addition to more traditional principles of good interface
design, we wanted to develop an interface that met the following
criteria:

   1. It must be simple. At Stanford, we introduce the graphics
       library early in the term when students are first learning
       about functions and procedures. Since the students have
       relatively few programming tools at their disposal, the
       graphics interface must not depend on advanced concepts such
       as pointers, records, or even arrays. Moreover, it should not
       export so many functions than the novice programmer cannot
       understand the interface as a whole. Beginning programmers
       can easily be intimidated by a large interface, even if they

are not required to use all of the functions it contains.

2. It must correspond to student intuition. The conceptual model
    that underlies the graphics interface must not be so
    mathematical or so technical that students have trouble
    understanding its operation. They have usually been exposed
    to Cartesian coordinate systems in high school mathematics and
    are now beginning to become familiar with the procedural
    programming paradigm. The design of the graphics interface
    should take advantage of that preexisting knowledge, even if
    the resulting conceptual model is at variance with the
    underlying model supported by the hardware.

3. It must be powerful enough for students to write programs they
    think are fun. The principal advantage of the graphics
    library is that it allows students to undertake programming
    problems that are exciting enough to capture their
    imagination. For this to be possible, the interface to that
    library must export functions powerful enough to generate
    interesting graphical displays.

4. It must be widely implementable. To ensure that our approach
    would be relevant to a variety of institutions and not just to
    Stanford, we believed that it was important to implement the
    graphics package on several different platforms, particularly
    those used most often for introductory-level education. We
    therefore designed the interface so that it did not depend on
    any specific platform.

In many cases, the individual criteria support one another in that
design decisions adopted to satisfy one end up advancing the others.
For example, choosing to leave a feature out of the interface results in
a library that is not only simpler but more portable, because there are
fewer features to implement for each new architecture. In other cases,
however, the design criteria trade off against each other, which
requires the designer to balance the competing criteria and find an
appropriate compromise.

2. EVOLUTION OF THE INTERFACE DESIGN

Fortunately, we have had the opportunity at Stanford to adopt an
evolutionary approach to the graphics library design. In each of the
four quarters during the year, the Computer Science Department teaches
two versions of the introductory programming course: CS106A, for
students with little or no prior programming experience, and CS106X,
which combines the material in the standard CS1/CS2 curriculum into an
intensive one-quarter course for students with more extensive
programming backgrounds. Both courses teach ANSI C using a library-
based approach [Roberts93] and use the graphics library extensively
throughout the course. Thus, we can experiment with a particular
library design one quarter and refine it for the next.

We introduced the first version of the graphics library in the fall of
1992, the first quarter in which ANSI C was used for the entire CS106A
population. At the time, the graphics library was very simple and
included only the following operations:

  o Initialize the library.
  o Move the pen to a specified position on the screen.
  o Draw a line segment with displacements dx and dy.

o Draw a circle with a given center and radius.

Students were assigned several simple programs using the library and
could also to take part in an optional graphics contest. The course
staff evaluated each entry on the basis of both its artistic merit and
its geometrical sophistication. In addition, I invited the students to
suggest new features that would have helped them design an even better
entry. The following are the most commonly cited suggestions, ranked
according to how many times each was cited:

  1. Arcs (as opposed to complete circles)
  2. Shaded regions
  3. Simple animation (by erasing and redrawing a figure)
  4. Color
  5. Text
  6. Mouse input

Having tried to design some graphical displays using the initial version
of the library, I was quite sympathetic to the students' concerns.
Adding the features they requested would certainly make the interface
more powerful and thereby make it more exciting for the better students.
Unfortunately, adding those features would also complicate the
interface, making it harder for other students to understand. Perhaps
more importantly, adding features would make it more difficult to
develop implementations for a variety of platforms and therefore
compromise the portability criterion.

My solution to this dilemma was to separate the graphics library into
two interfaces: a simple graphics.h interface providing a minimal set of
fundamental tools and a more elaborate extgraph.h interface offering the
extended capabilities. The textbook developed along with the course
[Roberts95] discusses only the simple graphics.h interface. Students
who seek greater challenges, however, can use the features offered by
extgraph.h. This design also promotes portability because an
implementation that covers only the functions in graphics.h is
sufficient for all the examples and exercises included in the text. At
Stanford, the standard introductory course requires only the graphics.h
interface, while the accelerated course typically requires the use of
extgraph.h as well.

The contents of the current versions of each interface are summarized in
Figures 1 and 2 at the end of this paper.

4. IMPLEMENTATION STRATEGIES

For the most part, implementing the rendering operations required by the
graphics library is not especially difficult. Existing graphics
libraries invariably include facilities for drawing lines, displaying
text, and filling polygonal regions, which are the only primitives the
library requires. Thus, to implement the procedures that actually
produce graphical output, all that is needed for each implementation is
a small amount of code to translate calls to the graphics.h interface
into the rendering calls required for the system graphics library.

Similarly, it is also usually a simple matter to create a new window on
the screen to use for drawing. Existing graphics libraries usually have
high-level tools for window creation, so the InitGraphics implementation
need only perform the appropriate set of function calls to create and
position a new window.

The hard part is implementing the conceptual model used by graphics.h,
which is fundamentally different from that used in existing graphics
libraries. In the typical graphics library supplied with a compiler or
operating system, applications are coded using what is generally called
the event loop strategy. Each application establishes an initial state
and then enters a loop with the following paradigmatic form:

```
void EventLoop(void)
{
    EventT event;

    while (event = WaitForNextEvent()) {
        RespondToEvent(event);
    }
}
```

Calls to WaitForNextEvent block until a system event occurs, at which
point the operating system delivers the event to the application, which
then initiates some appropriate response. These events include, for
example, requests to update the window, mouse clicks, and keyboard
activity.

While the event-loop paradigm is appropriate for experienced
programmers, it makes little sense to students in an introductory
course. Those students are used to writing a function called main that
executes sequentially, one statement at a time. In the first few weeks
of a CS1 course, it is unwise to introduce a different paradigm. Thus,
students should be able to code graphical applications that start at the
beginning of main and execute sequentially. For example, to draw a
one-inch square centered at the point (1, 1), introductory students
expect the program to look like this:

```
main()
{
    InitGraphics();
    MovePen(0.5, 0.5);
    DrawLine(0, 1);
    DrawLine(1, 0);
    DrawLine(0, -1);
    DrawLine(-1, 0);
}
```

This code contains no event loop and has no place for one. Thus, the
challenge of implementing the graphics library is to find a way to hide
the complexity of the event loop processing inside the library
implementation.

Fortunately, this problem is not usually as hard as it appears because
most C programming environments already include a solution as part of
the standard I/O library implementation. Like the graphics model
assumed by the graphics.h interface, the I/O model assumed by stdio.h is
not directly compatible with the event-loop paradigm. For an
application that uses stdio.h to work, there must be an event loop
somewhere that intercepts the keyboard events as they occur. From the
perspective of a client of stdio.h, however, the event loop is
invisible; the program simply calls an input function, such as getchar
or scanf, which eventually returns with the desired input characters.

To preserve the structural simplicity offered by stdio.h,
microcomputer-based implementations of C typically implement the

standard I/O library so that the required event-loop processing is
embedded in the console input routines.[1] Whenever the program waits
for input from the standard input device, the library implementation
simply enters an event loop, during which it responds to any system
events that occur. Keyboard input is queued until a complete line can
be returned to the application, but the event loop also processes update
requests, so that the program behaves correctly if windows are
repositioned on the screen.

The fact that the system must already perform the required event-loop
processing makes it easier to implement the graphics library. In most
cases, all that is needed is to make the existing event loop used for
console I/O also respond to update events for the graphics window. This
effect can be achieved in a variety of ways, as illustrated by the
following examples:

   o In the Borland C environment for Microsoft Windows, making the
      graphics window a child of the console window was sufficient
      to solve the problem, because the parent window automatically
      delivers events to its children.

   o In the THINK C environment on the Macintosh, it was necessary
      to make a dynamic patch to the console event loop so that it
      also called the appropriate update procedures in the graphics
      library.

   o The X Windows implementation for Unix required a different
      approach. In this implementation, the function InitGraphics
      calls fork to create two parallel threads, connected by a
      communication pipe. One of those threads returns to execute
      the application code. The other enters an event loop waiting
      either for X events that require a response or for commands
      from the application program sent over the communication pipe.

In addition to these platform-specific implementations, the public
distribution site on aw.com also includes a fully standard version that
does not actually display a graphical image but instead writes a
PostScript file suitable for printing. The existence of this
implementation ensures that it is always possible to use the simple
graphics.h interface even on platforms for which no specialized
implementation exists.

5. CONCLUSIONS

Our experience over the last two years at Stanford suggests that using a
graphics library in an introductory course enhances student interest and
helps reinforce several essential programming concepts. We have also
demonstrated that it is possible to design a graphics library that can
be implemented in a relatively portable way.

REFERENCES

[Hilburn93] Thomas B. Hilburn, "A top-down approach to teaching an
introductory computer science course," SIGCSE Bulletin, March 1993.

[House94] Donald House and David Levine, "The art and science of
computer graphics: a very depth-first approach to the non-majors
course," SIGCSE Bulletin, March 1994.

[Papert80] Seymour Papert, Mindstorms, New York: Basic Books, 1980.

[Roberge92] James Roberge, "Creating programming projects with visual impact," SIGCSE Bulletin, March 1992.

[Roberts93] Eric S. Roberts, "Using C in CS1: Evaluating the Stanford experience," SIGCSE Bulletin, March 1993.

[Roberts95] Eric S. Roberts, The Art and Science of C: A Library-Based Approach, Reading, MA: Addison-Wesley, 1995.

FOOTNOTES

[1] In THINK C, for example, this operation is performed as part of the implementation of the console I/O package console.c. In the Borland C/C++ system, the same functionality is provided by the EasyWin package.

Figure 1. Functions in graphics.h

InitGraphics();	This procedure initializes the package and creates the graphics window on the screen. The call to the InitGraphics function is typically the first statement in main.
MovePen(x, y);	This procedure moves the current point to the position (x, y) without drawing a line.
DrawLine(dx, dy);	This procedure draws a line extending from the current point by moving the pen dx and dy inches along the respective coordinate axes.
DrawArc(r, start, sweep);	This procedure draws a circular arc, which always begins at the current point. The arc itself has radius r, and starts at the angle specified by the parameter start, relative to the center of the circle. This angle is measured in degrees counterclockwise from the 3 o'clock position along the x-axis, as in traditional mathematics. The fraction of the circle drawn is specified by the parameter sweep, which is also measured in degrees. If the value of sweep is positive, the arc is drawn counterclockwise from the current point; if sweep is negative, the arc is drawn clockwise. The current point at the end of the DrawArc operation is the final position of the pen along the arc.
width = GetWindowWidth(); height = GetWindowHeight();	These functions return the width and height, in inches, of the drawing area of the graphics window.
cx = GetCurrentX(); cy = GetCurrentY();	These functions return the current x- and y-coordinates of the pen. Like all coordinates in the graphics package, these values are measured in inches relative to the origin.

Figure 2.   Selected functions in extgraph.h

```
DrawEllipticalArc            This procedure is similar to DrawArc in the
  (rx, ry, start, sweep);    basic graphics library, but draws an
                             elliptical rather than a circular arc.

StartFilledRegion(density);  These procedures are used to draw a filled
EndFilledRegion();           region.  The boundaries of the region are
                             defined by a set of calls to the line and
                             arc drawing functions that represent the
                             boundary path.  The parameter density is a
                             floating-point value that represents the
                             pixel density to be used for the fill
                             color.

DrawTextString(s);           This procedure draws the text string s at
                             the current pen position.  The pen position
                             is moved so that it follows the final
                             character in the string.  The interface
                             also contains functions to set the font,
                             size, and style of the text.

SetPenColor(color);
DefineColor(name, r, g, b, );
                             The SetPenColor function sets the graphics
                             package to draw in some color other than
                             the default black.  Colors are identified
                             by name, which makes it easier to save and
                             restore the current color.  By default,
                             only a small set of standard colors are
                             defined, but DefineColor allows clients to
                             define new ones by specifying their RGB
                             values.

x = GetMouseX();             These functions make it possible for the
y = GetMouseY();             client to use the mouse to design graphical
flag = MouseButtonIsDown();  user interfaces.  Given these primitives,
                             it is easy to design interfaces that
                             provide portable implementations of
                             buttons, menus, and other standard
                             interactors.

Pause(seconds);              The Pause function forces the system to
                             redraw the screen and then waits for the
                             specified number of seconds.  This function
                             can be used to animate the displays.
```