Experiences with GraphApp: A Tool for Teaching User Interface Design

Lachlan J. Patrick

Technical Report 515
10 September 1997

BASSER DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY OF SYDNEY
Experiences with GraphApp: A Tool for Teaching User Interface Design

Lachlan J. Patrick
Basser Department of Computer Science
University of Sydney, Australia
loki@cs.su.oz.au

Abstract

GraphApp is a cross-platform toolkit for the construction of graphical interfaces to applications. It is designed with speed of learning as a prime concern, and was used in a course teaching user interface design. This paper describes some of the implementation decisions which make this toolkit different from other similar tools. Some comparisons are made with other tools in terms of programming style and ease of use. The interface design course which made use of this tool is described in detail and some student programs and opinions are summarised to evaluate the usefulness of this tool.

1 Introduction

There are few general-purpose graphics toolkits designed specifically with learning in mind. This paper presents a new tool, GraphApp [1], and describes how it is designed and how it was used in a course.

Learners have a number of requirements which are different from experts. Beginners need to see easy-to-understand examples which they can modify, and have access to good tutorial documentation. Experts are more likely to write programs from scratch and use reference manuals for occasional help.

A toolkit imposes its own world view on a learner, using terminology and function names which may be quite foreign. To help a learner, it is a good idea to make this world view as similar to the learner’s view as possible, using terminology which is familiar.

To understand what the learner’s world view is, it is necessary to describe the intent of the student. In the context of the course GraphApp was used in, the intent of students was to create a graphical user interface to a program which would allow searching and updating of a small bibliographic database. So their focus was on using buttons and text fields to provide input and output to a small database.

Since this course was teaching user interface design principles, and the tools which can be used to build such interfaces, a number of different tools were presented to students, including HTML, Tk [2] and Visual Basic.

The context of this course places other requirements on a tool. Because there was little time to teach all of the tools, the toolkit should be quick to learn. It should work on a variety of operating systems, so students can take assignments home if they wish.

Additionally, the use of a toolkit should clearly demonstrate how to build user-driven modeless interfaces. This is the current, quite successful, model of user interaction underpinning such concepts as multi-tasking and multi window programs: the user is in control.

GraphApp was chosen as part of this course because it is a high-level graphics toolkit which is easy to learn and provides source code portability between X-Windows and Microsoft Windows [3] platforms. Its design has taken into account the needs of learners as well as experts.

This paper will describe the key design decisions involved in the development of GraphApp, before outlining the user interface design course and analysing some student responses to the tools taught. GraphApp will then be compared with other, similar tools.

2 The Design of GraphApp

GraphApp is an easy to learn, cross-platform C language graphics toolkit with a vocabulary optimised for user interface construction.

This section describes some design goals of the GraphApp toolkit and explains what implementation choices have arisen as a result.
2.1 Easy to Learn

What makes a graphics toolkit easy to learn? It depends what you try to do with it. A system such as OpenGL [4] is useful for rendering three dimensional scenes, so it could be said this is 'easy to learn' for people trying to perform that task.

GraphApp is optimised for creating user interfaces. It gives access to a variety of common widgets, such as buttons, text fields, windows, menus and so on, and it uses language appropriate to the task of creating user interfaces.

How easy a toolkit is to learn also depends on the choice of programming language and operating system. Some systems, such as Visual Basic, define their own language, while others like Tk can be bound to a number of languages, even though they began life matched to only one language.

GraphApp is a library build in the C language [5], which can also be used from the Python language [6]. The C language was chosen for a number of reasons. One was the target audience: the students who would use the library. Another reason was to do with availability of compilers for C.

The students who used GraphApp as part of a third year computer science course had a good working knowledge of the C programming language. Prior surveys had shown that the majority of students owned PC compatible computers at home. A general familiarity with graphical user interfaces, such as those used in Microsoft Windows, was assumed.

The computers used by the students in the practical part of the course work were X-Windows terminals linked to a central file server.

The C language matched students' prior knowledge well, and in practice it was found that many students owned C compilers for their home computers, which made taking source code home a real possibility.

In style, GraphApp's closest neighbours are the Tk toolkit, the SUIT toolkit, described by Pausch et al [7]. It is as fast to learn as each of these systems, if not faster, due to an optimised vocabulary for interface construction. This aspect of the library will be explored later.

Additionally, GraphApp was designed to work with X-Windows using Motif or Athena, and also with Microsoft Windows. This allows learners to transport code between home and work in many cases.

The toolkit can be used to introduce the central concepts of other event-driven environments such as X-Windows. GraphApp has a clean, event-driven mechanism which allows graphical objects to call user provided functions, known as 'call-backs'. This is a common model of programming well worth exploiting.

In the following sections, some example programs will be given, some features of the toolkit will be discussed, and the vocabulary design and the cross-platform nature of GraphApp will be explored in more detail.

2.2 Example GraphApp Programs

Consider these two programs. The first, shown in Figure 1, is the GraphApp "Hello World" program:

```
#include <graphapp.h>

void main(void)
{
    printf("Hello, world");
}
```

Figure 1

This program shows a window on the screen with the words, "Hello, world" written on it. The window stays there until the user closes it.

Notice that the library arranges for the creation of a visible window, and that events are handled automatically after the main program has ended. These are two features which can help beginners a lot, because they can learn how to achieve these tasks after some simple programs have been written.

The second program (Figure 2) creates a window with a button in it, which asks the user a question when clicked:

```
#include <graphapp.h>

void ask(button b)
{
    askyesno("Is this easy?");
}

void main(void)
{
    rect r = rect(10,10,80,30);
    newbutton("Ask me", r, ask);
}
```

Figure 2

The creation of the button with 'newbutton' automatically creates a window, and automatic event handling ensures the window stays visible until the user explicitly closes it. The question appears in a separate dialog box window.

The 'ask' function is a call-back which is supplied to the new button at creation time, to be called when the user clicks on the button. The rectangle 'r' defines the location of the new button, and the 'askyesno' function controls all of the dialog box interaction.

The program does nothing with the results of the dialog question, but it would be easy to examine the result returned from 'askyesno' and perform an action.

It can be seen from these two examples that GraphApp is a high-level toolkit with a simple vocabulary designed to aid beginners in performing what would be quite complex tasks in other toolkits.
2.2 Vocabulary Choices

Speed of learning has been an important factor in the design of GraphApp's vocabulary. The term 'vocabulary' refers to the names of function names, constants and object types, as well as the 'style' of programs which use the library.

A set of standard graphical widgets is available. Buttons, checkboxes, radio buttons, windows, list boxes, text boxes and labels all have their own constructors. These constructors have names such as 'newbutton', 'newwindow' and so on, in keeping with the tradition in C++ [8] and Java [9] to use the word 'new' as a constructing function.

In general, function names are formed from appropriate verb phrases which describe the operation. For instance, there are functions named 'hide', 'resize', 'invisible', 'getname', which operate on widgets or return information about them.

The style of programming supported by the library is in-keeping with the C procedural style of function calling. That is, functions take parameters, the first of which is the object being modified, if any, and the other parameters specifying what that modification is.

Thus, in GraphApp you might write show(w) to render a window named 'w' visible, while in C++ or Java you would write w.show(). A C++ binding for GraphApp could be achieved fairly easily using this transformation, but this was not the focus of the work.

By making function names from verb phrases, learning should improve, because the student can make sensible assumptions about what certain functions do, provided these verb phrases are well-matched to their expectations.

Some toolkits are property-based, as the following Tk/Python example of how to create a quit button shows:

```python
self.quitFrame.quit = Button(self,
('text': 'Quit',
'command':self.do_quit,
'bg': 'white'))
```

Note that the constructing function 'Button' is a noun, with several properties such as 'text' and 'command' are being added to the object. These properties can be thought of as adjectives describing a noun.

GraphApp uses a different approach to creating objects. Constructors have parameters describing the required properties of an object, and other functions exist to add other, less commonly needed, properties. For instance, the equivalent code to the above would be:

```python
quit=newbutton('Quit', r, do_quit); setbackground(quit, White);
```

Here, 'r' is a bounding rectangle while 'do.quit' is a call-back function. The constructor accepts just enough information to get by with.

The verb oriented approach eliminates the notion of adjectives by replacing it with operations on objects. Rather than having to think of a button as a collection of attributes, a student starts with the end-user's idea of a button being something you can click on, and adds to this notion functions which modify its behaviour or appearance.

This approach relies on using sensible defaults for the less common properties of objects, an approach to programming which may be seen as an example of a 'training wheels' approach to learning [10].

GraphApp employs a noun/verb programming model, where the nouns are familiar user interface elements and the verbs are functions. Each object's data, the adjectives, are hidden behind this function call interface.

This approach should improve the readability and intuitive appeal of the language, because instead of writing code such as this:

```python
setproperty(quit, "bg", White);
```

there is a more direct way of achieving the effect:

```python
setbackground(quit, White);
```

The number of function names is comparatively larger than similar toolkits, partly due to this policy but also because additional functions have been created instead of using extra parameters.

For example, hiding and showing widgets is achieved through the use of two separate functions. Some systems have a 'setvisible' function which works thus:

```python
// w invisible
setvisible(w, True);
// w visible
```

GraphApp instead uses two functions, 'show' and 'hide', to improve the comprehension of code, while eliminating a parameter. Separate functions make use of English verbs in a sensible way.

The cost of this approach is an increase in the size of the library's symbol table, but this is an acceptable trade-off if learning speed is considered important.

Parameter lists have been kept short through three methods: the above mentioned policy of using more function names instead of parameters, passing structures to some functions, and also by the use of a small number of global variables.

Some global variables have been used as an efficiency in drawing and also to reduce the length of parameter lists where it was sensible to do so. Their use is limited to drawing operations and widget creation. Even though they are global, each object has its own local private copy of them, used when needed. A function-call interface hides these variables from the user-interface designer.

These different approaches have resulted in 149 of the 177 functions defined by the library's header file using two or fewer parameters. Only one function has five parameters. Keeping the parameter lists short
helps students to remember the function calls and thus speeds learning. Code is also more readable with fewer long argument lists, which is important if marking student code by hand or trying to understand someone else's code.

2.3 Sensible Defaults

The above examples showed some use of the colour mechanisms in GraphApp. The ‘setbackground’ and ‘setcolour’ function accepted as parameters some predefined colours, such as 'White' or 'Red'.

By default, drawing occurs using black pixels on a white background. For many tasks, only widgets need be used, so drawing isn't even important. For beginners, the mechanisms for setting and using colours is probably not as important as producing a working interface.

GraphApp uses sensible defaults to allow students to learn concepts as needed, rather than all at once at the beginning of learning. If you don't know about colour, you can concentrate on drawing operations or widgets first and address that aspect of the toolkit later.

This provides a less steep learning curve than some systems where colour is in integral part of defining objects or drawing operations. This idea is not limited to colour, but also applies to the creation of windows, the initialisation of the library and even to event handling.

The example in Figure 2 shows that if a programmer begins a program by creating a button, the library automatically creates a window first, since buttons can only occur within an output window. The library assumes the common case by providing a generic window.

The programmer need not explicitly handle events either. Since most behaviour is controlled by adding call-back functions to widgets, the library arranges to call these functions when needed. If the programmer doesn't specify call-backs, nothing happens when the user clicks on that button or menu.

It is difficult to make GraphApp programs crash by misuse of the library. This promotes experimentation and learning in small steps, as well as facilitating incremental development of programs from prototypes. A number of pre-defined fonts, colours and cursors also makes a beginner's life easy.

2.4 Cross-platform

A cross-platform toolkit is one which can operate on different operating systems in the same way, and provide the user with the ability to move source code between computers.

In some environments, such as Java Applets hosted on the World Wide Web, this involves automatic compilation which is mostly transparent to the user. In the case of GraphApp, portability of source code is achieved, but the user must recompile for each platform.

GraphApp provides a thin-layer between the user programmer and the operating system's native graphical programming interface, or API. Unlike Java and Tk, GraphApp does not attempt to implement its widgets from scratch; instead it relies on the operating system.

This has a number of advantages. Firstly, programs compiled for each platform look like they belong on that platform. X-Windows widgets behave as expected in that environment, and in Microsoft Windows the buttons look correct, and the standard keyboard shortcuts work correctly.

Additionally, this approach allows the code for GraphApp to be quite small, around 4000 lines of code on each platform, which makes it easy to download and install on a home system.

There are some disadvantages in linking to each operating system in this way. One is that widgets do not behave in the same way across different platforms. For the most part they do, but there are occasional differences, particularly with appearance, which may be annoying.

These are mostly cosmetic problems. A more major problem lies in attempting to place the widgets supported by the library into a class hierarchy, as some systems do.

There is no real class-hierarchy in GraphApp because the widgets are outside the library, they reside in the operating system. So there is no way to guarantee the same behaviour of a class of objects on several platforms.

Class hierarchies can be good because inherited behaviours allow user interface designers to quickly produce new widgets based on old ones. However, the target audience of GraphApp is not widget designers, but end-users, and as such this is not a concern.

GraphApp supports a large number of common graphical objects which work in the same way on several platforms, but does not provide access to highly platform-dependent widgets. This allows compatibility across platforms and at the same time simplifies the implementation of the library.

The down-side of this approach is that the library is not as powerful as it could be if it only existed on one platform. This is acceptable because the benefits of being able to easily port programs are so significant.

Some benefits of portability are: programs rely on one header file for graphics, so they compile very quickly; working on a home system becomes very attractive, without having to install Linux; learning can therefore occur by experimenting outside the work environment; programs can be easily written for the home computing environment, improving the life of a Windows user; the source code for a program is maintained in one place for all platforms; and later improvements to GraphApp can give existing programs access to the Macintosh or other operating systems.

On the X-Windows system, GraphApp can be dynamically linked to programs, which is a boon for a university course where many students have limited disk space and RAM usage. The small compilation time of GraphApp programs also helps to reduce CPU load on student computers.
3 GraphApp in the Classroom

GraphApp was used as part of a one semester course teaching students in the third year of a Computer Science course about human-computer interaction. It was one of many tools selected to illustrate to students the choices available in programming tools to support construction of good user interfaces.

This section discusses the nature of the course’s programming component and assesses whether GraphApp was an appropriate tool for use in this course.

3.1 Course Outline

The fourteen week course was designed to teach user interface design issues using a problem-based curriculum. Students were given a small exercise to solve and this problem was solved four times using different tools.

To solve each of these small exercises, students were given an example program, which implemented a trivial pizza-ordering program, and told to modify this program to keep track of a small number of bibliographic references.

The tools students were exposed to were: HTML using the CGI interface; Tk/Python; GraphApp; and Visual Basic. A small search language was also implemented using yacc and lex [11], however students did not use this tool to create a user interface.

The time spent on learning each of these tools was not great. Students had two one-hour lectures and four hours of workshop time to implement a solution to each exercise, so each tool had to be learned quickly.

In between lectures on the various tools, issues which the students came across in their practical work were used as a springboard for lectures on user interface design principles.

For a major assessment task, students were asked to form into groups of about four people and then take one solution of the small interface construction task and develop it into a more complete and robust solution, with a user interface that improved on the prior prototypes.

These student groups had approximately four weeks in which to improve the design and implement a robust bibliographic database to allow book and journal titles to be recorded and retrieved by searching.

Figure 1 shows one student group’s solution to this major task, and in the foreground the example program they were given to start with.

Notice that the major project has copied the style of the original example, but added menus, database search functions and a series of buttons for changing the window between data entry and searching functions.

This example was one of the best solutions, showing mastery of how to remove widgets from a window and create new ones in their place, as well as understanding the use of menus.

Many other solutions used two windows, one for data entry and another for searching the database. Most copied the style of the original pizza-ordering example slavishly, but some were quite original. The example code given to students clearly influences their coding practices greatly, as this phenomenon of copying the pizza example occurred with Tk too.

3.2 Course Requirements

The problem-based nature of the course meant that the tools used had to be simple and quick to learn, powerful enough to implement real-world solutions to interface problems, and each should illustrate a different approach to the question of how to write an interface program.

The Hypertext Mark-up Language used in the World Wide Web illustrates a particular approach to building interfaces. HTML is easy to learn, relevant to real-world problems and demonstrative of specialised interface-design languages.

Tk/Python illustrated the use of an interpreted scripting language with a graphical user interface toolkit to allow rapid development of interfaces. Interpreted languages can be developed interactively, an important attribute in the world of user interface development.

GraphApp represented the traditional compiled language approach to event-driven programming using callbacks to achieve functionality. GraphApp programs are implemented in the C language, which offered the advantage of familiarity to students who had been programming with C for over a year.

Visual Basic was chosen as the representative of GUI-builder systems, which integrate a scripting language with direct visual manipulation of graphical objects on the screen to design an interface to a program.

The course ran for fourteen weeks, and each student had two hours of practical time each week to learn the above-mentioned tool, which they typically spent
every other week doing. The other weeks were used for teaching screen design aspects, usability, user-centred design and testing.

Having less than ten weeks to introduce students to five different tools meant that it was critical that each of the tools could be mastered by students having only two hours of exposure through lectures and four hours of exposure through hands-on exercises.

This was a tough constraint to place on the tools, but each of them satisfied this constraint.

Because the same task was reimplemented using each new tool, the mastery of a new tool was the focus of each practical session. Example code which solved a similar but simpler problem was given for each of the tools, so students could compare source code from previous implementations and re-use code by adaptation.

This approach was successful, and few students had difficulty in completing solutions using all of the tools. It fact it would be fair to say that from the marks recorded almost all students were able to produce solutions to the exercises tasks for each tool.

3.3 Student Usage of GraphApp

For a major design assessment task, students worked in groups of about four people to expand on an earlier solution to the bibliographic interface problem. They demonstrated their final working solution to their class in a group presentation at the end of the semester.

The usage of each tool for the major assessment task is summarised in Figure 4:

<table>
<thead>
<tr>
<th>Tool Used</th>
<th>Number of Groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML/CGI</td>
<td>12</td>
</tr>
<tr>
<td>GraphApp/C-language</td>
<td>10</td>
</tr>
<tr>
<td>Tk/Python</td>
<td>3</td>
</tr>
<tr>
<td>Visual Basic</td>
<td>2</td>
</tr>
<tr>
<td>Total Number</td>
<td>27</td>
</tr>
</tbody>
</table>

**Figure 4**

As can be seen from this table, GraphApp was used by a similar number of groups as used HTML. There were two groups which specifically requested to use Visual Basic, and this was allowed although it was not intended to be used for the assessment. The Tk graphical interface toolkit, which is comparable to GraphApp in many ways, was chosen by less than half as many groups as chose GraphApp.

This fact is probably attributable to four factors:

- GraphApp programs can be built using the C language, which is more familiar than Python to our students;
- GraphApp’s pixel-based widget layout technique is simpler to understand than the Tk packer method of specifying widget positions relative to each other;
- GraphApp programs are fully portable to the Microsoft Windows platform, as well as Linux, which allowed students to take source code home and tinker with it much more easily than Tk/Python code; and documentation for GraphApp was more readily available than for Tk/Python.

What is the justification for assuming these were the reasons for students choosing GraphApp over Tk? There is some direct evidence and some circumstantial evidence. The direct evidence comes from a small number of respondents to a web-based survey which was run at the end of the semester in which the course was run and at the beginning of the next semester.

The survey indicated that most students knew C better than Python, that they preferred GraphApp to Tk, and that it was important that GraphApp worked on multiple platforms, particularly Windows. This survey is not statistically significant however, since only around 7% of students responded to it.

The circumstantial evidence comes from the number of students who did download the Windows version of the library, a number which was obtained from the web-logs for the software download page. There were 33 downloads from this page from a course containing about 104 students, and although it is possible one person may have downloaded the software more than once, this is a rough indicator that taking assignments home was an attractive option.

Anecdotal evidence supports the theory that Tk packer confused a lot of people, since the author was a tutor in this course and was dealing with several questions. At the end of the semester, a course questionnaire was filled in and returned by about 70% of the students in the course. Perceived relevance of the various tools used in the course were rated on a scale of one to five, five being highly relevant. The following ratings were recorded:

<table>
<thead>
<tr>
<th>Component Relevance</th>
<th>Ave. Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML</td>
<td>4.50</td>
</tr>
<tr>
<td>GraphApp</td>
<td>3.55</td>
</tr>
<tr>
<td>Tk/Python</td>
<td>3.34</td>
</tr>
<tr>
<td>Visual Basic</td>
<td>3.13</td>
</tr>
<tr>
<td>yacc/lex</td>
<td>2.53</td>
</tr>
</tbody>
</table>

**Figure 5**

These results mirror the usage of GraphApp in the assessment task. This survey covered many other aspects of the course, and did not attempt to discover why a particular tool was favoured over another.

GraphApp will be used again in this course, and future work will attempt to establish why students choose the tools they do for their assessment projects.

4 Comparisons

Full comparisons between GraphApp and other similar systems would be beyond the scope of this paper, but some general comparisons can be made.

In terms of expressive power, GraphApp is in the same league as Tk, Java’s AWT, and a few other toolkits, such as LibSX. LibSX is a small X-Windows library which hides much of the low-level details of X,
and so has found some success in teaching computer graphics.

To demonstrate that GraphApp has a similar expressive power to these other tools, the author has converted a student's LibSX program to use GraphApp instead, and timed the performance of both programs. Output from the program (a ray-tracer) is shown in Figure 6.

![Figure 6](image)

The timing tests have shown that GraphApp is at least as fast as LibSX, which is itself very fast, being little more than a slight abstraction of the Xt toolkit.

In style, GraphApp is less object-oriented than either Tk or Java, but this is due more to language choice than inherent design.

In code size, GraphApp programs are slightly smaller than either Tk or Java programs. Object-oriented languages such as Java tend to add many keywords to the language, which is one reason for this, but a significant factor is also the optimised nature of GraphApp.

Compare the following Java "Hello World" program with the GraphApp version given in Figure 1:

```java
import java.awt.*
public class Hello {
    public static void main(String[] argv) {
        Frame f = new Frame("Hi");
        f.add(new Button("Hello"));
        f.pack();
        f.show();
    }
}
```

![Figure 7](image)

The Python/Tk version of this program would be similar. Notice the need to explicitly create a window (or frame), explicitly add a button to this window, pack the button to give it a certain size, and show the window.

GraphApp can perform most of these operations for the beginner, while an expert can do these things explicitly to have greater control.

The one thing GraphApp lacks at present time is access to an automatic widget layout system, such as the pack method in Java or Tk. However, using rectangles to specify locations of widgets can be easier to understand for beginners, and also gives experts better control of widget placement.

Consider this last example of a program which creates a window with some text fields and a button for ordering a pizza:

```c
#include <graphapp.h>

field name, phone;

void place_order(button b) {
    printf("Name: %s\n", gettext(name));
    printf("Ph: %s\n", gettext(phone));
    exitapp(1);
}

void main(void) {
    window w;
    rect r;
    w = newwindow("Pizza",
                   rect(0,0,400,230),
                   StandardWindow);
    r = rect(10,10,80,30);
    newlabel("Name: ", r, AlignRight);
    r.y += 35;
    newlabel("Phone: ", r, AlignRight);
    r.y += 35;
    r = rect(100,10,250,30);
    name = newfield("Your name? ", r);
    r.y += 35;
    phone = newfield("Your phone? ", r);
    r.y += 35;
    newbutton("Order Pizza",
               rect(50,100,100,30),
               place_order);
    show(w);
}
```

![Figure 8](image)

The example shows the use of a rectangle 'r' in placing objects exactly, the creation of a window, a button and some text labels and fields. It also demonstrates the use of a call-back function, 'place_order', to do something when the user clicks on the "Order Pizza" button.

GraphApp does not have access to the Java libraries nor the Tk package nor many other facilities. But it is high-level and appropriate to the task of creating user interfaces and performing drawing operations.

It is useful for learning about graphics and interface design, but is also a general-purpose programming tool with wide application.
5 Conclusions

GraphApp is a graphics toolkit with a vocabulary optimised for the needs of user interface construction and speed of learning. It is compact, efficient and works with more than one operating system.

It has been used in a course teaching user interface construction tools and methods. The students of that course have used this tool and reported being satisfied with its design, even using it more often than another similar tool. It is appropriate for the task of evolutionary prototyping and teaching user-interface construction.

6 References


ISBN: 186451 324 1

Basser Department of Computer Science
University of Sydney
Sydney NSW 2006
Australia