

**You Can Try This At Home:
Visual Programming
in Engineering Education**

Peter M. Maurer

**VGAPP Laboratory Technical
Report Number ED-3
Dept. of Computer Sci & Eng
University of South Florida
Tampa, Fl 33620**

You *Can* Try This at Home: Visual Programming in Engineering Education

**Peter M. Maurer
Department of Computer Science & Engineering
University of South Florida
Tampa, FL 33620**

Abstract

A profound change is taking place in the world of applications programming. Visual languages, like Visual Basic and its cousins have become some of the world's leading development languages. In spite of the inherent weaknesses in the underlying language, or more probably, because of them, visual languages have raised programming far above the statement level, and have encouraged code reuse to a degree that was never before possible. This has all happened so quickly, that it is hard for academic curricula to keep up. Even as this is being written, profound changes are taking place in the design and scope of visual languages. New reusable components are being invented, and different uses are being found for them. One of the most interesting developments is the use of reusable components in web-based distributed applications.

Because visual programming is changing every day, it is difficult to construct a useful course in the area. Nevertheless, the impact of visual programming has been so profound that we cannot simply ignore it. This paper describes how visual programming can be integrated into an existing curriculum. It begins by describing visual programming, and gives a tongue-in-cheek history of its development. It introduces custom controls, and describes their usage and impact on visual programming. It ends by describing a series of laboratory exercises that can be used to introduce visual programming. These exercises are designed to be used with an existing course, rather than constituting an entirely new course, but could easily be fleshed out into a complete course in visual programming. In our curriculum, we use these exercises in our Principles of VLSI Design Automation course, because VLSI design automation places heavy demands on the user interface. Visual programming allows the student to design complex user interfaces with a minimum of effort, thereby freeing the student to concentrate on the main aspects of the course. We believe that a far more aggressive approach must eventually be taken toward the teaching of visual programming, but it will take another year or two for the visual programming world to achieve some stability. In the meantime, we believe that the approach we have taken is of great benefit to our undergraduate program, and could be easily adopted by others wishing to reap the same benefits.

1 Introduction

At one time Graphical User Interfaces, or GUIs, were a curiosity in the academic world[1,2,3]. They were toys, and it wasn't necessary to take them seriously. Yes, you had to know something about them to program a MacIntosh, but only a rabid fanatic would do that. And besides, MacIntoshes weren't real computers anyway. When Microsoft windows came along, it didn't change much of anything. PCs were real computers, but so tiny and insignificant that no *real* programmer could take them seriously. Serious programmers did everything on UNIX, or if you were *really* serious, on an IBM Mainframe. But the insidious creep of the GUI didn't stop there. Workstation manufacturers began to provide them for their UNIX systems. At first, one could safely ignore them, because they were only available on the system console, and most of us still used terminals. Even those of us lucky enough to have our own workstations could ignore the manufacturer's GUI, because nothing you wrote for it would run on anything else, even if you recompiled the code.

The world was nearly perfect in those days. We assigned simple programs like reading a list of ten numbers and sorting them. We gave our students text-files with the necessary input (what other kind of file is there?), and they gave us nice blocky-looking printouts. Yes, of course there were other I/O functions besides `getc` and `putc`, but why would anybody use them?

But a snake crept into this paradise. When the price of workstations finally dropped to the point where we could finally afford them, they came with this funny interface called X-Windows. At first, the only thing you could do with it was open a whole bunch of virtual terminal windows. This was OK, because it meant we could do the same old things we always did, even though we were now doing four or five things at once. Well, we did use a real GUI program for E-Mail, but that was just E-Mail after all, not anything important.

It's hard to trace the path of events after this, but somehow we all ended up using PCs with Windows 95. (Well, there are a few UNIX die-hards out there.) Perhaps it was the lovely looking printouts you could get with TrueType fonts, embedded graphics, and laser-printed output. Maybe it was because the latest version of Zork wouldn't run on anything else. Who can say? The only thing that is certain is that we have paid a high price for our lack of vigilance. Now, when we give our students a text file, they ask us how to spell "text." The nice blocky printouts are gone, and nothing will line up right. Nobody wants to count characters any more, and they keep asking us about things like "buttons" and "check boxes." The world has turned.

2 GUI Programming

GUIs seem to be here to stay, and nobody takes those clunky old text-based programs seriously anymore. We think, "Perhaps we ought to try to teach our students something else." So, we go out and buy the eight-shelf-feet of books that describe how to program our favorite GUI. We start with message loops, callback functions, messages and all that stuff (what is this "MakeProcInstance" thing anyway?) and after a few weeks of work what we end up with is nothing much more than a "Hello World" program.

This is a real problem. That clunky old text-based sort program at least accomplished *some* useful work. Most simple GUI programs are nothing more than cute little toys. (To tell the truth, they're not even that cute.) Most of us would prefer to teach our students

something useful. For a while, many of us believed that Object Oriented Programming would save us. Maybe all that “icky” stuff would go away and our students could concentrate on *real* programming. To make a long story short, it didn’t work. Some progress was made, but it seemed like the only stuff that went away was the easy stuff. Stuff that we didn’t really mind doing anyway. “Let’s see ... To handle mouse input I have to capture the Mouse-Down event, record the coordinates, set mouse-capture, then intercept all Mouse-Movement events and track the coordinates, and finally intercept the Mouse-Up event, record the coordinates, and release mouse capture. Now all I need to do is determine what objects on the screen are at these different mouse coordinates, and I can begin my *real* programming.” OOP didn’t change this at all.

Salvation came, but it came from a completely unexpected source. Consider the humble BASIC programming language. Were it not for Microsoft’s obscene obsession with BASIC, it surely would have died a natural death long ago. What can one say about a programming language where line numbers are not only required, but considered *part of the program*. While it is true that BASIC has been cleaned up in many ways, no amount of cleaning could possibly turn it into a good programming language. Nevertheless it is BASIC, and its unique properties that lead to the real solution to the GUI programming problem, specifically the version of BASIC known as Visual Basic.[4]

3 The Visual Basic Programming Language

Visual Basic is a language that is used to program dialog boxes. Dialog boxes are those gray boxes that MS Windows uses to display error messages, like “Unknown Error Program Aborted.” They are also used to set program options like default printer, and current font. They were designed to display small amounts of information for short periods of time. They were intended as an add-on to a real program. Visual Basic took the peculiar approach of elevating the dialog box to the status of a real program. A Visual Basic program begins with a blank dialog box known as a form. Various objects, called controls, are drawn on the form. Short segments of Basic code are added to the controls to implement the functionality of the program. The details of the GUI, such as intercepting mouse messages, and locating items on the screen, are handled automatically behind the scenes.

Visual Basic was distributed with a number of pre-defined controls, some of which are illustrated in Figure 1. An interface was also provided that allowed the user to create custom controls. This idea was hardly new, since most other GUIs provided a similar interface. If the early examples were any guide, these were expected to be useless little trinkets like radio knobs and volume-control sliders.

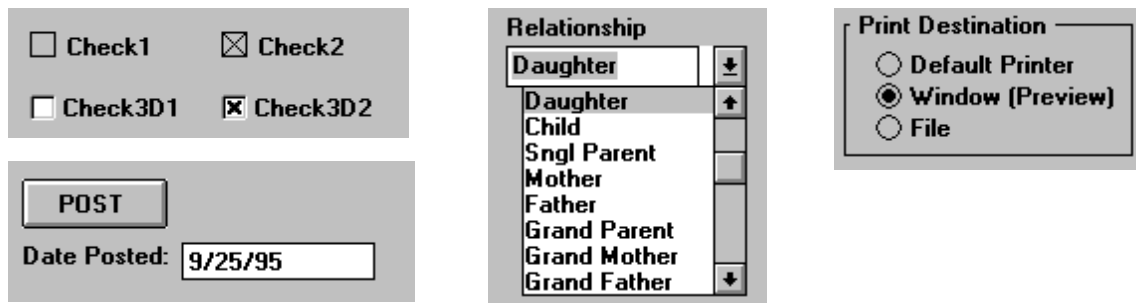


Figure 1. Examples of Controls.

At first Visual Basic was laughed at as a “toy” language by most right-thinking people. (Many people still feel this way.) But somewhere along the line something changed. Visual Basic Version 3.0 included support for database access. This interface was expanded to handle virtually any existing PC database system. The Visual Basic interface was already similar to the interfaces provided by these systems and it was a lot cheaper than any one of them. What’s more, the programs were freely redistributable. You could develop a database interface and distribute it throughout your company without paying anyone an additional dime. It was like getting many database systems for much less than the price of one. This really lifted Visual Basic out of the swamp. The only real problem was that the set of controls that came with Visual Basic was so woefully inadequate that it was almost impossible to do anything very complicated. Furthermore, it was impossible to fill the gap with Basic programming, because *Basic really is a toy language!* This combination of conditions led to one of the most amazing developments in modern programming, namely the explosive growth of the third-party custom control industry.

4 Custom Controls

Let’s review now, we have a toy programming language that can’t be used for anything complicated, a collection of controls that’s so limited we can’t do anything with them, and an interface that was designed for the creation of useless trinkets. Not exactly a textbook formula for success.

The custom control interface provides a set of properties, which are internal variables that can be assigned default values at compile time, and can be read or written at run time. Only simple data types can be used for properties. Structures and pointers are not allowed. Arrays of simple types are permissible, but you can forget about complex linked structures and the other exotic data types so popular with other add-on programming packages. A custom control can also provide a set of events. Each event can be associated with Basic subroutine that is executed whenever the event occurs. Events are things like mouse clicks and keyboard characters. It is generally not necessary to associate a subroutine with each available event.

One thing that was conspicuously missing from the original custom control interface was the ability to define new functions. This omission has since been corrected, but it persisted for long enough that it had a permanent effect on the way custom controls are designed. The restriction to simple data types and a function-less interface meant that the procedures for using a custom control had to be simple. This imposed a severe discipline on the custom control designer, and forced the development of interface procedures that probably wouldn’t have existed otherwise. When you use a custom control, you never have to define a collection of data structures, link them, and call a string of meaningless functions to initialize them. There is no way to do this even if you wanted to, so the control must do it for you.

The inability to define new functions turned out to be a minor problem, because the control has the ability to monitor all usage of its properties. Complex actions can be performed when a new value is assigned to a property, or when the value of a property is accessed. It has become a common practice to provide an “action” property, which can be used to execute operations that are not associated with a property value.

Third party custom controls have advanced from simple data-entry boxes to two-dimensional grids (for display of multiple records), to full-featured spreadsheets, to virtually anything you could imagine. Practically any application that exists as a separate program also exists as a custom control. To give a few examples, there are full-featured word processors, personal information managers, graphical editors, web browsers, communication managers, voice mail systems, as well as a vast array of interface elements like tool-bars and tabbed dialogs. There are several hundred different products available.

5 Inserting Controls into the Curriculum

The development of visual programming has been sudden and surprising, so much so that Visual Basic has become one of the most important tools for application development. We in the academic community probably ought to respond to this, but there are valid reasons for proceeding slowly. The most important is that visual programming is undergoing profound changes almost on a daily basis. It's hard to keep up with the latest developments, and it's even harder to identify any unifying principles that could be used as the basis for a "Visual Languages" course. Nevertheless, there is a strong industry demand for visual programming, and we should at least introduce our students to the fundamentals, as they exist today.

While it would be useful to provide a course in visual programming, this material is easy to learn, and probably insufficient for a full semester course. Instead we recommend adding Visual Basic, or one of its cousins, to an existing course. Our approach was to add several Visual Basic exercises to a highly recommended elective course, which also had the effect of increasing the enrollment of that course.

At a minimum, students should complete the following exercises.

- **Read a list of items from a text file, and display the list in a grid, list-box, or combination-box control.**
- **Perform different manipulations of data in response to clicking on different buttons.**
- **Accept text from a text-box, process it in some way, and display the result in another text-box.**

It is possible to combine these three things into one or two exercises, depending on taste. Two weeks is sufficient time to complete these exercises. Next, we recommend a short exercise in combined C/Visual Basic programming. In this exercise, the students implement several small functions in C, and compile them into a Windows DLL file. (A DLL file is a library of functions that can be loaded at run time. Although DLL programming is simple, it can be simplified further by including several things in an instructor-supplied #include file. Once the DLL is created, it can be tested using an instructor-supplied interface program, or with a program created by the student. The following is an example of such an exercise from a recent course offering[5].

A Dynamic Link Library is a collection of independently compiled functions that can be called by other programs. A DLL can contain two types of functions, Exported Functions, and Internal Functions. Exported Functions must be the target of a FAR call, and must use PASCAL calling conventions.

Exported functions must also have the `_export` keyword. Small DLLs, such as that created in this experiment, usually are coded in a single `.c` file. This `.c` file should begin with the following code.

```
#include <windows.h>

/* The following two functions are required, but do nothing */

/* the following statement is used only with Borland C */
#pragma argsused
int FAR PASCAL _export WEP(int exittype)
{
    return 1;
}

/* the following statement is used only with Borland C */
#pragma argsused
int FAR PASCAL LibMain(HANDLE hInstance, WORD wDataSeg,
                      WORD wHeapSize, LPSTR lpszCmdLine)
{
    return 1 ;
}
```

This code could be supplied in a `#include` file.

Instructions: Create an MS Windows dynamic link library that will perform the following functions.

1. Convert a string of hexadecimal digits into a long integer.
2. Convert a long integer into a string of eight hexadecimal digits.
3. Given an integer `n`, compute the sum of integers from 1 to `n`. Return an integer.
4. Compute the square of an integer, and return an integer.

The first function must be named *HexToLong*, the second must be named *LongToHex*, the third must be named *SumToN*, and the fourth must be named *ISquare*. The name of your library must be *firstone.dll*.

The function declarations must look as follows.

```
long FAR PASCAL _export HexToLong(char *Hex)
void FAR PASCAL _export LongToHex(long Value,char *Hex)
int FAR PASCAL _export SumToN(int N)
int FAR PASCAL _export ISquare(int N)
```

Every Windows DLL must have a `.def` file which is used during the link phase of the compilation. The `.def` file for this project should look as follows.

LIBRARY FIRSTONE

DESCRIPTION 'My First DLL'

EXETYPE WINDOWS

**CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD MOVEABLE SINGLE**

HEAPSIZE 4096

This is a 16-bit example. The 32-bit compilers have changed everything.

An exercise such as that illustrated above should be followed immediately by another that requires students to create their own function headers. Taken together, these exercises can be completed in 2-3 weeks.

As a final step, students should be required to create their own custom control. Instructor-provided #include files are an absolute necessity for such an exercise. Templates for data structures and precompiled resource files are also required. The following is an example of a such an exercise.

Using the supplied templates and files (explicit instructions for using these items can be found on the diskette), create a custom control that has two custom properties *InString* and *OutString*, and one custom event, *BadString*. *InString* and *OutString* must be of type *String* and will normally contain multiple lines of text. When a new value is assigned to *InString*, the control will process the string one line at a time. Any line that contains parentheses must have balanced parentheses. If a line has unbalanced parentheses, fire the *BadString* event. This event must have two parameters, the line number (starting with zero) where the error occurred, and the text of the offending line. Eliminate all bad lines from the output. For a line where the parentheses are OK, change the parentheses to square brackets. In other words, () goes to []. Accumulate the output for all OK lines into the *OutString* property. Test the control using input and output from multi-line text boxes.

Although a substantial amount of “framework” code is necessary to create the custom control, virtually all of it can be supplied to the student. The student can then concentrate on writing the algorithms to perform the required task. This last task should take four weeks to complete.

The more adventurous students may wish to create their own custom controls. There are two ways that this can go, depending on whether the control creates visible output on the screen. The controls that we have created so far are invisible at run time, and do nothing more than provide a programming interface to algorithms written in C. Most students will want to stay with invisible controls, but some may wish to try their hand at screen output. Any control that interacts with the screen must be a full-blown Windows program with a control loop, and complex message handlers. Students who wish to try their hand at this sort of thing should start with simple graphical exercises and work up to more complex screen/mouse/keyboard interaction.

6 Conclusion

In short, we believe that visual programming and the use of custom controls is, or will shortly become, an essential part of all Computer Science and Computer Engineering curricula. It is not yet clear how this will affect the curricula, but we believe that it is necessary to begin taking steps to integrate these topics into existing courses. In the future, we would expect to see curriculum changes that include both courses in visual programming, and courses in the design of custom controls.

At the present time, creating a complex control is beyond the scope of an undergraduate program. In fact, developing such a control could lead to a reasonably strong master's thesis. However, there is continual progress being made in development tools for visual programming. For example, the latest version of Visual Basic allows new controls to be constructed in the Basic language itself. It is not yet clear where this trend will lead, but eventually the process of creating a new control should be simplified to the point where it is feasible to assign a full-featured control as an undergraduate project.

One of the more interesting trends is the development of controls that can be used in many different environments in addition to the Visual Basic development environment. In particular, there are controls that can be used as components of a web page with the Internet Explorer[6]. When used in conjunction with other networking features, this will allow controls to be used in a distributed computing environment with little or no additional effort beyond that normally expended to create the control.

There is also a trend to extend the concept of custom controls to environments beyond the PC. In particular, MacIntosh and UNIX platforms will probably have their own custom controls in the near future. For academia, this is a welcome trend. There is something uncomfortable about teaching a programming style that is applicable only to one type of personal computer. If controls are available on other platforms, we can assure ourselves that we are teaching general principles, not just "Hacking for fun and profit."

There is so much development being done, and so many new things becoming available that it is impossible to predict where future development will lead. Although the future is uncertain, it is certain to be exciting.

7 References

1. *Inside Macintosh: MacIntosh Toolbox Essentials*, Addison-Wesley Publishing Company Inc., New York, 1992.
2. Petzold, C., *Programming Windows 95*, Microsoft Press, Seattle 1996.
3. Mansfield, N., *The Joy of X: An Overview of the X Window System*, Addison-Wesley Publishing Company Inc., New York, 1992.
4. Microsoft Corporation, *Visual Basic 5.0 Professional Edition*, web site, http://www.microsoft.com/products/prodref/195_ov.htm.
5. Maurer, P. *Dr. Maurer's Course Materials*, web site, <http://www.csee.usf.edu/~maurer/courses.html#UGDA>.
6. Microsoft Corporation, *Internet Explorer Home*, web site, <http://www.microsoft.com/ie>.