Experience with Tcl/Tk for Scientific and Engineering Visualization

Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974
bwk@research.att.com

ABSTRACT

This paper relates experience gained while building a graphical user interface for a system that helps design and implement indoor wireless communications facilities. In this paper, I will describe this interface briefly, and draw some conclusions about what works well and what does not. I will also compare Tcl/Tk with Visual Basic for the same application.

1. Overview

For the past 18 months we have been developing a system for design and optimization of indoor wireless communications systems. The input is a description of the coordinates and composition of the walls of a building, and myriad parameters for power, frequency, antenna types, signal to noise ratios, etc. A family of programs computes radio energy throughout the building to predict the behavior of the system, optimize the locations of base-station transceivers to minimize system cost, and analyze coverage, sensitivity, etc. The overall system is described in [F95].

The user interface is a Tcl/Tk program that coordinates the activities of these programs. It displays plan, elevation, and perspective views of a building, and shows in living color the power received at each location for given base-station positions. The picture may be scaled and panned over; station locations and parameters may be set interactively.

Figure 1 shows the plan and elevation views of one floor of the Hynes Convention Center in Boston as it appears through this interface. A base station (the small blue circle) has been placed at the center of the circular area near the bottom left, three meters above the floor. The coverage map in the plan view shows received signal strength at a height of one meter throughout the building, using the color scale shown near the upper right corner. Areas below a selectable threshold (here –78 dBm) are shown in gray. (Normally this appears in brilliant color; you may be seeing a gray scale copy, which is not nearly as nice.)

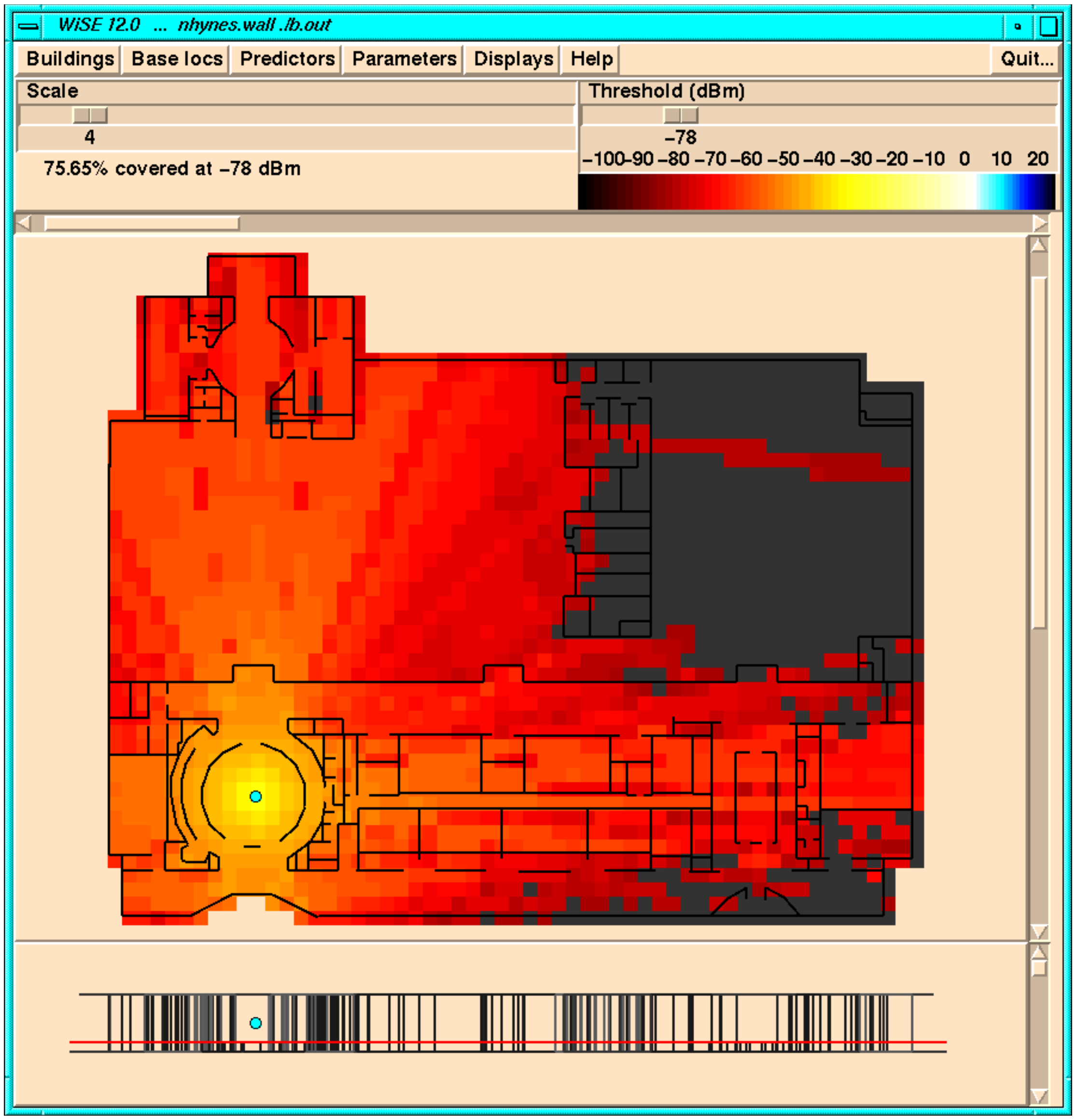
Each menu button across the top of the display raises a sub-menu of further choices, some of which in turn raise other top-level windows. For example, the Parameters menu provides submenus for setting parameters of prediction algorithms, radio properties, grid spacing, optimization, and color map. It also provides for saving and restoring current parameter settings and base locations, and for restoring all parameters to their default state. Figure 2 shows the Predictor and Radio parameter sub-menus to illustrate the general style.

The most important component of the system is a C++ program that predicts radio coverage throughout a building, given parameter values and building geometry and composition. This program (dubbed ''bounce'' because it works by tracing rays as they bounce around the building) reads ASCII input files of parameters and wall data, and produces ASCII output files giving, for example, coverage at each point of a grid that covers the building. Figure 3 shows a small sample of input and output files.

A second component is an optimizer that attempts to improve coverage by moving base stations; it calls bounce as a subroutine. There are also a handful of supporting programs. All of these are controlled by the user interface.

The components of this system have been in continuous evolution since the fall of 1993; in particular, there have been many versions of the user interface. Quite early, I decided not to use any extended version of Tcl or Tk, nor to write C code to be bound into the same executable as Wish; all communications would be with standalone programs, through files or pipes. This has simplified portability to our user population, who are not comfortable with importing any public domain software, let alone an amalgam from multiple sources. At the same time, it does not seem to have cost much in development time or efficiency.

In any case, the system structure clearly reflects this decision. The Tcl/Tk interface handles drawing and user interaction, and is currently about 3000 lines long.



75.65% covered at -78 dBm

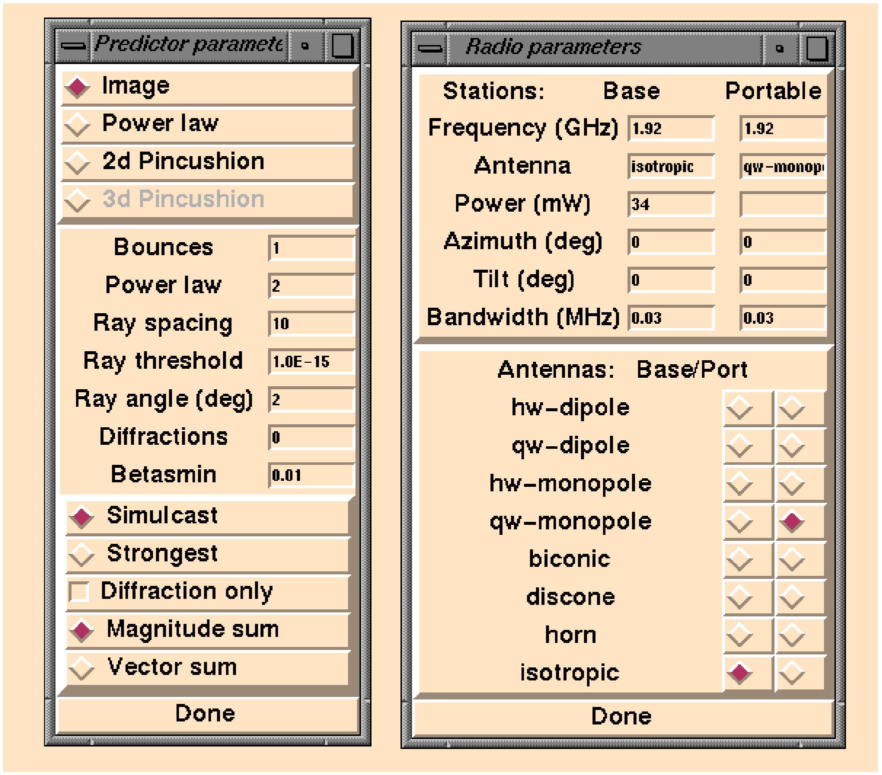
Figure 1: Interface view of Hynes Convention Center, Boston

A separate program, originally in AWK and now in C++, manages data structures for wall and receiver grid coordinates. This program is about 1400 lines long. These programs are described further in the next section.

The split into two components was encouraged and ultimately forced by two issues. First, Tcl's notation for arithmetic expressions is clumsy; it is easier to write most expressions in C. More important, however, Tcl is not very good at data structures, since it provides only associative arrays and strings (upon which a veneer of linked lists can be applied). Again, C or C++ is more expressive and the code is much easier to maintain. These operations also run significantly faster in C/C++; this more than compensates for any extra file traffic.

2. Canvases

The user interface makes extensive use of the canvas widget. One can draw a variety of graphical objects on canvases, including lines, rectangles, circles, ellipses, text, and arbitrary polygons, in any color, with outlining, shading, etc. Each object can be tagged with any number of strings so that groups of objects can be treated as a unit. Any group of objects can be scaled or moved independently of others.



[✓ Image]
Power law
2d Pincushion
3d Pincushion

Bounces	1
Power law	2
Ray spacing	10
Ray threshold	1.0E-15
Ray angle (deg)	2
Diffractions	0
Betasmin	0.01

[✓ Simulcast]
Strongest
Diffraction only
[✓ Magnitude sum]
Vector sum

Stations:	Base	Portable
Frequency (GHz)	1.92	1.92
Antenna	isotropic	qw-monop
Power (mW)	34	
Azimuth (deg)	0	0
Tilt (deg)	0	0
Bandwidth (MHz)	0.03	0.03

Antennas:	Base	Port
hw-dipole		
qw-dipole		
hw-monopole		
qw-monopole		✓
biconic		
discone		
horn		
isotropic	✓	

Figure 2: Menus for predictor and radio parameters

The plan and elevation views of a building are drawn with a modest number of lines; coverage maps are drawn with lots of colored rectangles, one for each grid point. For example, the Hynes convention center is about 140 by 175 meters and has 313 walls (on the one floor shown); bounce automatically generates a grid of 2329 rectangles of 3 meters on a side to cover this area. AT&T's facility in Middletown, NJ, has 4320 walls on five floors; this is the largest building for which we have done serious experiments.

Tags

The tag mechanism provided by canvases is indispensable. Each wall is tagged with the name `walls` so that all walls can be manipulated at once. Each wall also has a unique tag and an encoded composition so that its identity and properties can be displayed when the mouse is clicked on it. Each colored rectangle is tagged with its power and delay values so these can be displayed when the mouse is pressed on any point in the building. Other objects are tagged with type names and serial numbers.

Tags are also used in conjunction with Tk's `raise` and `lower` commands to control visibility. For example, we can hide the walls behind the coverage or raise them above by a single command. We also use tags to make sure that base stations are always displayed above walls, which are normally above coverage data. Rulers may be added or removed; again, tags make it easy to treat a group of related objects as a single unit for moving or deleting.

Specific events are bound to specific tags; for example, base stations and portable transceivers can be moved in plan or elevation view and the corresponding object tracks in the other view. Walls may be moved only if the capability is explicitly enabled; users are disturbed by walls that move unexpectedly when touched with the mouse. Most other objects can not be moved under any circumstances. Touching any point on the canvas displays its coordinates in building space.

Finally, tags are the only way to keep track of objects that appear in more than one canvas. For example, a base station appears in both the plan and elevation views, and when it is moved (by the user) in one view, the other view has to change as well. The code that is bound to button 1 on a base or portable identifies the object in the original view, finds the object in the other

```
filetype bounce
output coverage
prediction image
nbounce 1
ndiffract 0
antenna isotropic qw-monopole
power 34
freq 1.92
threshold -70
nxmit 1
xmit 1 19.7 7.3 3 power 34 azimuth 0 tilt 0 antenna isotropic
...
```

(a) Input parameter file

```
filetype wall
title law office
wall        1     0.914   8.839   0.000   5.486   8.839   3.657   2   3
wall        2     5.486  10.058   0.000   7.010  10.058   3.657   2   3
wall        3     7.010   8.534   0.000  11.277   8.534   3.657   2   3
wall        4    11.277   7.315   0.000  16.763   7.315   3.657   2   3
...
```

(b) Wall description file

```
filetype rcvr
grid    0.9295 1.034 1  0.5 0.5 0       43 31 1
...
nrcvr 729
nxmit 1
xmit 1 19.5 6.3 3 antenna isotropic power 34 azimuth 0 tilt 0
rcvr 1 0.9295 9.035 1 -53.8572 1.77397 0
rcvr 2 0.9295 9.535 1 -63.7057 1.78578 0
rcvr 3 0.9295 10.034 1 -61.7194 1.60315 0
...
```

(c) Output file

Figure 3: Input and output file excerpts.

view that has the same name tag, then moves them in unison.

Almost all interactions take place on button 1, with cursor changes to indicate special states, like preparing to add or delete a base or portable, or measuring a path. Users much preferred a single button model to the more conventional (on Unix) use of three buttons.

Coordinate Systems

There are too many coordinate systems. Distances are measured in meters or feet, and users want to view either, so conversion functions are ubiquitous. In the plan view, real life has the Y coordinate going up, while Tk has it going down; we finesse this issue by converting building coordinates once and for all. The same problem arises with Z coordinates, but it is harder to avoid, since there is a clear meaning to up and down in this dimension. More conversion functions. We also stretch the Z coordinate in the elevation view, normally by a factor of two, by another conversion function. On-screen building displays can be scaled, which adds another conversion; furthermore, for really large buildings, another scaling is needed to shrink them so the range of the slider that controls scaling isn't too big. Selecting the largest or smallest value on the scale slider scales the whole picture by a factor of 40, but the slider itself always shows a range from 0 to 40.

Of course it is not sufficient merely to scale every item on a canvas. The small colored circles that represent base and portable stations have to remain a constant size as the building is scaled. Fortunately, this one is easy: scale everything, then use tags to rescale the bases to cancel the original scaling. Another conversion function is needed.

There are also some potential problems of lost precision in the coordinates of bases and portables. They are stored in most places in terms of their centers, but Tk stores them in terms of their bounding box. Another set of conversion functions is needed.

Coordinate transformations have been a continuing irritation and a fertile source of bugs. Some of the problems could have been obviated by better design ahead of time, but foresight has been difficult in a program that has gone through nearly 30 releases.

Canvas Limitations

Although canvases are remarkably flexible, there are some limitations. Canvases are not bitmaps in Tk 3.6, so it is difficult to do arbitrary drawing on them. In particular, since the most effective perspective algorithms are based on raster-filling in the proper Z order, we use a separate program to draw perspective views of buildings. The perspective viewer is invoked and controlled from the Tk interface, but it is implemented (with code from Tom Duff) as a standalone Xlib program of about 1400 lines of C++.

One of the features of the interface is a mechanism for collecting multiple displays onto a single canvas, where they can be arranged and annotated, then saved in Postscript. Unfortunately, it is not possible to make a direct copy of an item from a canvas. Instead one must interrogate the type and properties of each item, then instantiate a new item with the same type and properties. This is excruciatingly slow. Even a small building requires several minutes of computing time to copy the main canvas onto the clone window.

The X color map that underlies Tk limits us to 256 colors on the screen at one time. The color scale in the interface has been restricted to about 150 colors so that there is little danger of running out, but if some other program is also using a lot of colors, Tk will go into monochrome mode.

3. Interprocess Communication

As mentioned above, the interface is split into two main pieces: a Tcl/Tk portion that handles interaction with the user, and a C++ component that manages wall and receiver-grid information. The two halves of the interface communicate through files and a two-way pipe. The C++ program uses normal C structures to handle wall and receiver coordinates, and converts the output of prediction programs to Tk commands. For example, a wall-file line like

```
wall 1 138 61 0 138 53 8 1 2
```

is converted into Tk commands to draw it on two canvases (plan `.c` and elevation `.h`):

```
.c create line 690 305 690 265
   -fill black -width 2 -tag {walls #1 w2}
.h create line 690 80 690 0
   -fill #444444 -width 2 -tag {walls #1 w2}
```

The multiple tags name the wall and encode its composition; a gray scale in the elevation view gives some illusion of depth. The commands are written to a temporary file, which is read into Tk with a `source` command. The scaling is done by the C++ program; there is another set of coordinate transformation functions here too.

The two-way pipe requires synchronous communication. In a typical interaction, the interface sends a one-line command (an ASCII string) to the C++ program, with a flush to force the output. The C++ program performs the requested operation, then replies with a single line that either contains the entire answer, or points to a file that the Tcl/Tk program reads with a `source` command; again a flush is needed. For example, if the user selects Predict received power from the Predictors menu, the interface creates an input file for bounce, runs it, then sends a command to the C++ program to load the data computed by bounce. That data is converted by the C++ program into Tk commands to be loaded with `source`.

The prediction (bounce) and optimization programs are completely independent of the interface. They are called by Tcl `exec` commands and all communication is by command-line arguments, files, and status returns. Since some of these operations can take a very long time, the commands are set running asynchronously (using `exec &`). To keep track of their status, every few seconds the interface runs `ps|grep` and displays its output; when the process goes away, the results can be converted to Tk commands and displayed. The process id can be used to kill the job early if necessary; a Kill button is displayed as long as the process is running. This is somewhat tricky to set up and the output of the `ps` command is system dependent, but overall the mechanism works well.

The separation of the interface into Tcl/Tk and C++ halves has also been successful; each half concentrates on what it does well, and there is less need to force a language into a task that it wasn't meant for. At the same time, there are some definite problems. The most serious issue is that an uncomfortable amount of information is stored in both halves, and must be kept consistent.

For instance, the scale is needed by the interface for display and by the C++ program so that already-scaled Tk commands can be generated. Keeping the scale purely in the interface would work, although every time data was loaded from the C++ program it would have to be scaled (and then the bases and portables would have to be unscaled). Scaling is fast enough that this would likely be feasible.

A more serious problem is keeping track of bases and portables in both halves. New bases and portables are usually generated by the user with the mouse, and then passed to the C++ program; coordinates have to be unscaled and converted from bounding box to center. But bases and portables can also come from data files, so they may be seen first by the C++ program. Furthermore, bases and portables have a display size that is independent of scaling, and color attributes that have to be dealt with by both halves. Keeping these possibilities straight has been an ongoing nuisance.

A final issue is how to handle wall editing. Users want to be able to add, delete, and move walls with the mouse, but the information about walls is kept in the C++ program. This requires further coordination between the two halves, and more coordinate transformations.

4. Visual Basic

It was clear from the outset that our intended user population really wanted the whole set of programs, including the interface, to run on a PC under Microsoft Windows.

After we had enough experience to believe that the interface was somewhat stable, I undertook a subset implementation for Windows 3.1. The prediction and optimization C++ code needed almost no changes; those programs are now identical on Unix and Windows. The real issue is the interface. There are a couple of preliminary versions of Tcl/Tk available for Microsoft Windows, but these were not yet robust enough to be seriously considered. It was also unclear whether they would provide an interface that had the right look and feel for Windows users. And of course our users would be even less happy depending on these untried and unsupported packages.

The obvious alternative is Microsoft Visual Basic, a popular user interface builder for Windows. VB provides a set of about 20 ''controls'' that are analogous to Tk's dozen widgets; for example, there is a ListBox control that looks much like Tk's `list` widget, and a pair of ScrollBars that match Tk's `scale` widget. Each control comes with a set of properties directly equivalent to Tk's configuration parameters; these may be set statically or at run time. Each control also comes with a set of methods — the operations that can be performed on it (like inserting a line into a list box) — and a set of events that it responds to. Again, though details vary quite a bit, the analogy with Tk is strong, and familiarity with one system makes it easy to understand the other.

To program in VB, one interactively selects controls from a menu, places them on a Form (a control rather like Tk's `frame`), and sets initial properties from another menu. Still another set of menus gives access to templates for the code to be executed for events; this is equivalent to the code one writes for the `-command` operation of a widget or for the `bind` command, but VB is more structured in how this is expressed. For example, the subroutine for the event that occurs when a button is pushed is always called *button-name*`_Click`.

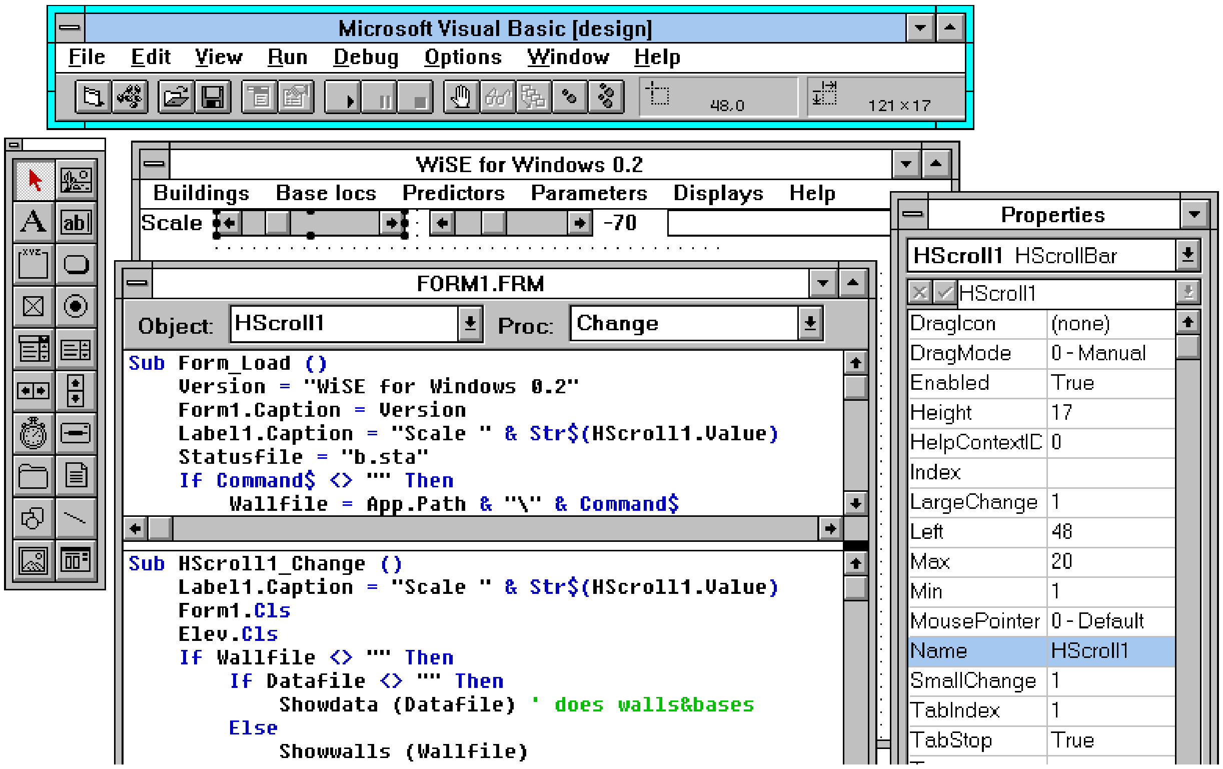
Syntax is checked as code is entered so trivial errors are caught instantly; global errors like missing declarations or misspelled names are caught when execution begins. If the program contains no errors, all of the code and control windows are hidden, so the screen looks as it will when the application is real use. The cycle of editing and testing is very convenient in VB, though the facilities for viewing and editing text are so primitive that one yearns for a powerful editor like `ed`.

Figure 4 shows a set of VB windows during design of the interface. The top window is VB's main control, the left window is the tool bar for the available controls, and the right window is the property list for the horizontal scrollbar `HScroll1`, which is the currently-selected control. The window labeled ''WiSE for Windows 0.2'' is the interface itself (undergoing construction) and ''FORM1.FRM'' is the code window, showing parts of two subroutines. The top, `Form_Load`, is executed when the main window is first loaded. The bottom, `HScroll1_Change`, is executed when the scaling scrollbar (to the right of the word ''Scale'') is changed.

Most operations that can be done at form-creation time can also be done during execution, but VB is not as dynamic as Tcl/Tk; in particular, there is no equivalent of the `source` command. There is a clear-cut separation between ''compilation'' and execution, so it is not possible to create code on the fly as one can do in Tcl.

Figure 5 shows a similar prediction to the one seen in Figure 1, using the Visual Basic interface on Windows.

The version of the wireless interface with VB is incomplete, as are any conclusions that might be drawn from the experience. Nevertheless, it is clear that VB is an effective way to create interfaces for Windows programs. Here are some specific observations.


Microsoft Visual Basic [design]
File Edit View Run Debug Options Window Help
48.0
121 x 17
WiSE for Windows 0.2
Buildings Base locs Predictors Parameters Displays Help
Scale
-70
FORM1.FRM
Object: HScroll1
Proc: Change
Properties
HScroll1 HScrollBar
HScroll1

```
Sub Form_Load ()
    Version = "WiSE for Windows 0.2"
    Form1.Caption = Version
    Label1.Caption = "Scale " & Str$(HScroll1.Value)
    Statusfile = "b.sta"
    If Command$ <> "" Then
        Wallfile = App.Path & "\" & Command$
```

```
Sub HScroll1_Change ()
    Label1.Caption = "Scale " & Str$(HScroll1.Value)
    Form1.Cls
    Elev.Cls
    If Wallfile <> "" Then
        If Datafile <> "" Then
            Showdata (Datafile) ' does walls&bases
        Else
            Showwalls (Wallfile)
```

DragIcon	(none)
DragMode	0 - Manual
Enabled	True
Height	17
HelpContextID	0
Index	
LargeChange	1
Left	48
Max	20
Min	1
MousePointer	0 - Default
Name	HScroll1
SmallChange	1
TabIndex	1
TabStop	True

Figure 4: Visual Basic Design Screens

VB provides an interactive facility for creating the initial geometry of an interface (something that Tk does not, though packages like XF do). This is convenient for getting started, much easier than Tk's `pack` command. In effect it is an interactive version of Tk's `place` command. But Tk's packer wins hands down when one wants to create windows that change size. Its dynamic adjustment of sub-window sizes and positions is completely automatic; in VB, one has to write code for all this tedious geometry (as one would have to do with `place`).

Tk provides a richer set of events (inherited from X) than VB does, and they can be combined and used much more freely. VB does provide built-in Drag, DragOver and DragDrop methods for some controls, which are convenient if one can use them, but otherwise inflexible and non-extensible.

VB provides a common dialog control for file system browsing and selection; it is the standard Microsoft file browser, and is easy to use. Tk provides no such standard, unfortunately, so each user has to cobble one together and each one looks and acts different. More generally, a major strength of VB is that applications written with it share a common and familiar look and feel; this is an important selling point for any piece of software.

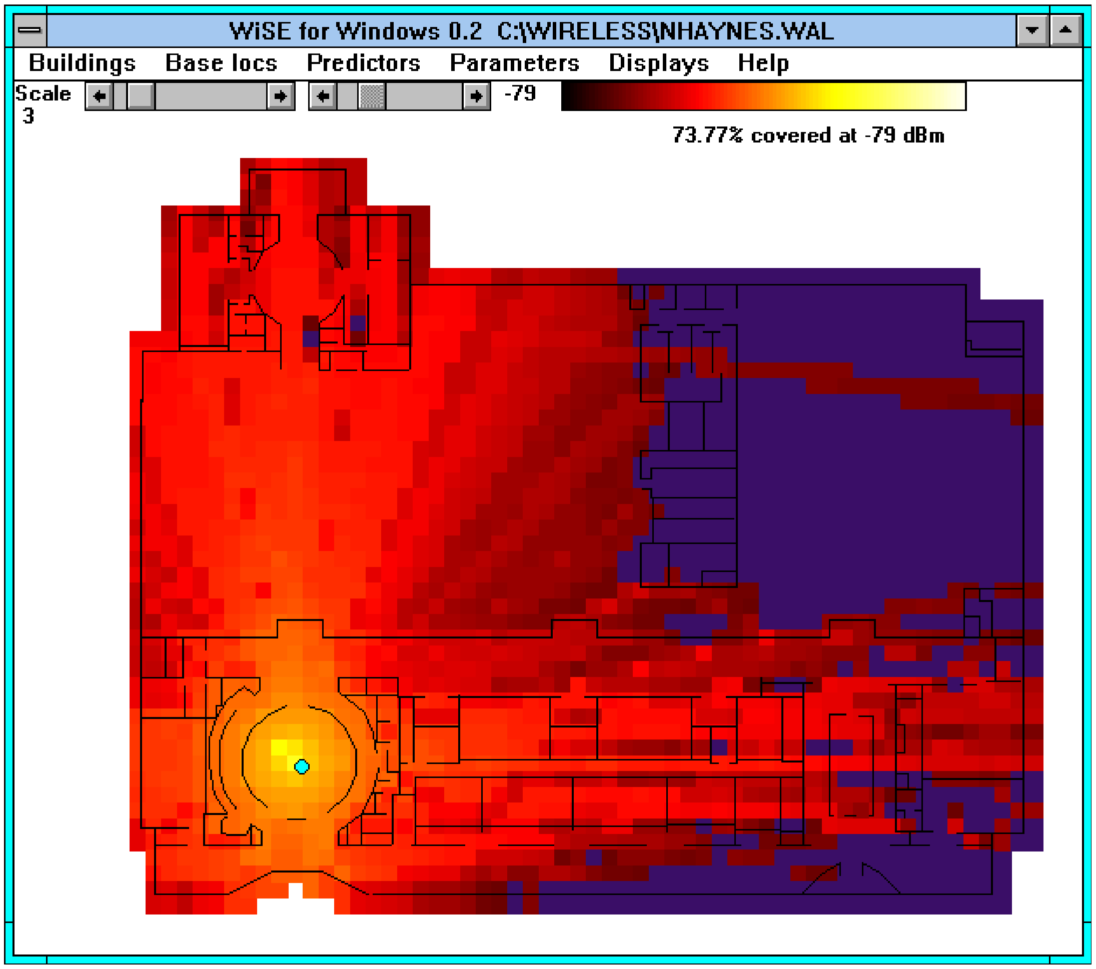

WiSE for Windows 0.2 C:\WIRELESS\NHAYNES.WAL
Buildings
Base locs
Predictors
Parameters
Displays
Help
Scale
3
-79
73.77% covered at -79 dBm

Figure 5: Visual Basic interface on Windows

VB provides a helpful ‘‘setup wizard’’ that gathers all the components needed for an export package, and compresses them onto a minimal set of diskettes. The package includes a standard installation program so the recipient can install the program on another machine, using the conventional `A:\SETUP` mechanism. The recipient need not have a copy of VB; enough of its functionality is automatically included in the export package. And again, the installation process looks conventional and familiar to the recipient.

VB’s extension language, a dialect of Basic, is clumsy but a vast improvement on the Basic of years ago. It provides integers, single- and double-precision floating-point numbers, strings, arrays, and structures. It has control flow, functions, subroutines, and real scope rules. There is an excellent on-line help system. As a purely personal reaction, it took me perhaps a day to internalize VB’s conventional syntax and semantics, where it took me months to become comfortable with Tcl. Figure 6 shows a straightforward VB function that parses a string into blank or tab-separated fields that it stores in a global array `FF`. VB itself provides nothing that does this parsing operation.

Figure 7 shows a more graphical piece of code, an excerpt from the subroutine that reads received power information from a file, parses each line with `Getfields`, and draws colored rectangles in the plan view.

One of the most important properties of VB is that it is easy to connect VB code with dynamic link libraries (.DLL’s), which are the standard way in which components are packaged in Windows. These libraries can be one’s own code or code from others. This is analogous to the `tclAppInit` mechanism, but simpler and cleaner. (VB can only link to 16-bit .DLL’s, perpetuating the arcane problems of the original PC architecture.) There is also a substantial third-party industry creating

```
Function Getfields (s As String)
    Dim nf As Integer, inword As Integer, i As Integer
    Dim c As String

    nf = 0
    inword = 0
    For i = 1 To 20
        FF(i) = ""
    Next i
    For i = 1 To Len(s)
        c = Mid(s, i, 1)
        If c = " " Or c = Chr$(9) Then   ' tab
            inword = 0
        ElseIf inword = 0 Then
            nf = nf + 1
            FF(nf) = c
            inword = 1
        Else
            FF(nf) = FF(nf) & c
        End If
    Next i
    Getfields = nf
End Function
```

Figure 6: Input Field Splitting in Visual Basic

```
Open fname For Input As fn
Do Until EOF(fn)
    ' w, n, x0, y0, z0, pwr, dly
    Line Input #fn, s
    n = Getfields(s)
    If FF(1) = "rcvr" Then
        x0 = HScroll1.Value * CDbl(FF(3))
        y0 = HScroll1.Value * CDbl(FF(4))
        pwr = CDbl(FF(6))
        ntot = ntot + 1
        If pwr >= hscrThresh.Value Then
            nin = nin + 1
            Line (x0 - dx, y0 - dx + Dy)-Step(2 * dx, 2 * dx),
                 Colormap(pwr + 100), BF
        Else
            Line (x0 - dx, y0 - dx + Dy)-Step(2 * dx, 2 * dx),
                 RGB(100, 100, 100), BF
        End If
    ...
```

Figure 7: Received Power Display in Visual Basic

new VB controls to be included in arbitrary programs, and adventurous users can do the same. This is equivalent to writing new Tk widgets and appears to be of similar complexity.

By far the most serious limitation for the wireless interface application is that VB provides nothing remotely approaching the canvas widget in capabilities, and the thought of programming equivalent behavior in Basic is daunting. At the bottom, the VB Form control provides only the graphics primitives of lines, circles, and rectangles. There are no tags, so one has to keep track of objects for oneself. Graphic objects appear to be handled strictly as bitmaps, with only a minimal notion of Z order, so moving an object (for example a base) can leave a hole behind. Much painful code would have to be written; the best approach would probably be to create a Canvas control.

VB also suffers from having to run in the awful Windows environment, where the least provocation can cause one's entire machine to go catatonic. Interprocess communication in the Unix sense is difficult in Windows; mechanisms like DDE and OLE are complicated beyond description, and a simple notion like the pipe is non-existent. Newer systems like Windows NT and Windows 95 appear to be incompatible, but they are still complicated. Compilers, though blessed with elaborate user interfaces, are bug-ridden and shaky.

As a rough summary, for the specific purposes I have used it for, Visual Basic is significantly better than Tcl/Tk for the purely visual part (creating the interface on the screen and having it look somewhat like a commercial product), worse for programming, and extremely unsatisfactory for interprocess communication. Each is the easiest interface-building tool in its native environment.

5. Observations and Conclusions

I began building interfaces with conventional tools, with toolkits like Xt and widget sets like Motif. These were incredibly frustrating: it was necessary to study hundreds or even thousands of pages of manuals and write voluminous code to achieve even the simplest effects. By this standard, Tcl/Tk is wonderfully productive; in a few hours one can accomplish what might well take days or even weeks with C-based tools.

For the wireless application, this has been especially important: since I am implementing for an application area in which I am not an expert, it is vital to adapt quickly to the needs of the real users. I have done a great number of experiments to refine the interface; this amount of evolution and refinement simply would not have been possible with another tool.

Tk is efficient enough for most purposes; some components, like text widgets, are so fast that one is left wondering how they could work so well. Other aspects, such as parsing input, seem surprisingly slow, and canvas copying is unusable. It would be helpful to have a cost model for various Tcl and Tk constructs, to help predict what will be fast and what will not.

Fortunately, however, it has proven straightforward to partition tasks between Tcl and C++ to match each to what it does best. The Tcl mechanisms for interprocess communication are reliable and fast, so this works well. Extending the system by writing code to be loaded directly with the Tcl library, as Ousterhout's original design intended, gives better efficiency and simpler program structure at a modest cost in portability.

I am not convinced that Tcl would survive by itself; it has many competitors. But coupled with Tk, there is nothing else in the Unix world that comes even close for building interfaces. The package is extremely robust, very well documented, and has an active and cooperative group of users. The source code is freely available and of exceptionally high quality. It is clearly possible to build production-quality user interfaces with Tcl/Tk, and to do so far faster than with competing tools.

Acknowledgements

I am deeply indebted to the other members of the group that has worked together on the wireless system: Steve Fortune, David Gay, Orlando Landron, Reinaldo Valenzuela, and Margaret Wright. Their endless patience and helpful suggestions have contributed greatly to the interface and to my enjoyment in building it. I am also grateful to John Ousterhout for creating and disseminating a tool of such value.

References

[F95] S. J. Fortune, D. M. Gay, B. W. Kernighan, O. Landron, R. Valenzuela, M. H. Wright, WISE Design of Indoor Wireless Systems. *IEEE Computational Science and Engineering*, **2**, 1, pp. 58–68, March 1995.