There’s no such thing as a simple GUI

• A simple GUI
• Why do you need GUI’s in the first place?
• Fundamental knowledge
• GUI design strategy
• Example
• Pitfalls

Danny Harvey
President
Boulder Real Time Technologies, Inc.
A Simple GUI

```sh
#!/bin/sh
#
exec wish "$0"

button .button \
    -text "KILL ME!" \
    -background green \
    -foreground red \
    -command exit

pack .button
63 korl%
```

June, 2006
Why do you need GUIs in the first place?

- Think HARD before you go down this road
- GUI programming should be considered as a task that requires advanced programming knowledge and skills
- Do you know why you need a proposed GUI?
- Have you considered less exotic alternatives?
- GOOD reasons for GUIs:
  - To show complex information that lends itself to graphical displays
  - To provide intuitive and highly choreographed user inputs – note that GUIs tend to constrain user interactions
- BAD reasons for GUIs:
  - To alleviate the user from typing (the developer will certainly be typing a lot more)
  - A vain attempt to put a more “sophisticated” or “sexier” front-end on some function that doesn’t really need it – note that a well designed generalized type-based user interface is usually going to provide much more function and flexibility than most GUI interfaces
Fundamental Knowledge

• GUI programming requires lots of knowledge regardless of Antelope; you will not be successful with this unless you do your homework
• The Antelope tools are no replacement for the fundamental knowledge needed by any GUI programmer
• The Antelope tools make it easy for an already experienced GUI programmer to interface with the various Antelope objects
• What “knowledge” does a GUI programmer need?
  – Good knowledge of the underlying programming or scripting language
  – A journeyman’s understanding of the X-windows system; e.g. the X-server, the relationship between clients and the X-server, the event driven nature of X-windows interactions, fonts, colors, images, the various graphics primitives, scaling
  – Good knowledge of the particular widget package that is to be used. In Antelope we mainly use tk-based widgets, either in tcl or perl.
GUI Design Strategy

• Clearly define the problem you are trying to solve – it may surprise you to find that this step may either eliminate the development task entirely or point it in a direction that does not require a GUI
• “storyboard” the GUI – make drawings of what it should look like and exactly how the user would interact (i.e. what particular widgets will be used, how information will be displayed, process flow, etc.)
• Try to dissect the overall problem into three logical parts; 1) user GUI front-end, 2) internal data engine and 3) a graphical display back-end that will show whatever information you need to show
• Don’t try to do the whole problem in one monolithic chunk; it is fine to be running separate programs and scripts that talk to each other in some fashion – this also helps in prototyping and debugging
• Start off with the bare minimum GUI functionality – you can always add more walking menus and dialogs later if you really need them
• Be patient – don’t expect to come up with your “final” solution quickly; GUIs tend to be perpetual and incremental works-in-progress – accept that fact and you will be a happier person
GUI Design Strategy

• Choose your language for the GUI and display parts
  – I will skip over C, C++ and java
  – Antelope contains fairly standard perl/tk extensions; this provides the highest performance scripting approach with the sophistication of the perl language. Downsides are lack of Antelope perl/tk graphical extensions and perl’s hyper-paranoid security limitations on normal tk IPC.
  – Antelope contains fairly standard tcl/tk extensions plus special Antelope graphics extensions, like brrtplot; this provides the highest graphics functionality scripting approach with the simplicity of the tcl language and ease of fully duplex tk-based IPC. Downsides are potential performance problems and limitations in overall complexity due to simplistic nature of tcl language.
GUI Design Strategy

• Figure out how to glue the major pieces together
  – Internal data engine can be Datascope, ORB or standalone analysis programs, like dbwfmeas
  – IPC can be implemented through combinations of database manipulations, external parameter files, command line arguments and the use of tksend to pass messages between processes
  – A good approach is to modularize design by using small stand-alone display libraries and scripts
Example - \texttt{dbnoise}

- Written as a \texttt{tcl/tk} script to provide a GUI to the \texttt{dbwfmeas} program for specifying parameters for noise spectra computations, execute the computations and display the resulting spectra – we consider this to be a simple GUI (420 lines of code)

- \texttt{dbwfmeas} is designed as a graphic-less high performance computation engine that reads data from a database, computes stuff and puts its computations into a database; all of the computational parameters are specified through a normal Antelope parameter file
Example - dbnoise

• Start by going through the process manually of setting up `dbwfmeas` to compute spectra; consult `dbwfmeas` man page, find some example data to work with and do what is needed to compute spectra.

• Determine exactly what parameters need to be specified by the user; from this come up with a front-end “storyboard” for the input GUI:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Widget Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Station code</td>
<td>entry widget</td>
</tr>
<tr>
<td>Channel code</td>
<td>entry widget</td>
</tr>
<tr>
<td>Start/end time</td>
<td>entry widget</td>
</tr>
<tr>
<td>Computation time window</td>
<td>entry widget</td>
</tr>
<tr>
<td>Output spectrum units</td>
<td>radiobutton widgets</td>
</tr>
<tr>
<td>Taper function</td>
<td>radiobutton widgets</td>
</tr>
<tr>
<td>Time slice values</td>
<td>entry widgets</td>
</tr>
<tr>
<td>Execute</td>
<td>button widget</td>
</tr>
</tbody>
</table>
Example - dbnoise

- Write a skeleton **tcl/tk** script that makes the GUI without actually doing anything – iterate until it looks right and includes the right information
Example - \texttt{dbnoise}

```
#!/bin/sh
#
exec $ANTELOPE/bin/awish $0 -- "$@

package require Datascope

# Copyright (c) 1999 Boulder Real Time Technologies, Inc.
# This software module is wholly owned by Boulder Real Time
# Technologies, Inc. Any use of this software module without
# express written permission from Boulder Real Time Technologies,
# Inc. is prohibited.

lappend auto_path $env(ANTELOPE)/data/tcl/pkg2
source $env(ANTELOPE)/data/tcl/pkg2/displaynoise.tcl

set env(SCHEMA_DEFAULT) SDAS1.0 ;# Set default Datascope schema
set pf dbnoise

tk_setPalette \#d9d9ee

option add #Font (helvetica 10 bold)

catch {package require Tclx}
package require Datascope
package require Brrtplot

# define procedures

proc usage {} {
    puts stderr "usage: dbnoise dbname"
}

proc config_window {} {
    global sta
    --More--(8%)
```
Example - dbnoise
Example - \texttt{dbnoise}

- Extend the \texttt{tcl/tk} script to perform the steps that you worked out when manually executing \texttt{dbwfmeas}:
  1. Build up a temporary parameter file in /\texttt{tmp}
  2. Execute \texttt{dbwfmeas} with the proper command line arguments being careful to capture standard and error output
  3. Monitor to see when \texttt{dbwfmeas} is finished and determine if it ran successfully
  4. If \texttt{dbwfmeas} encountered an error, display the error message
  5. If \texttt{dbwfmeas} ran successfully, display its results
  6. Clean up, i.e. get rid of temporary files
Example - dbnoise

Example code:

```python
puts $f [format "measurements $tbl"]
puts $f [format "LArr"]
puts $f [format "channels $tbl"]
puts $f [format ".*.*"]
puts $f [format ""]
puts $f [format "offset $s" $overlap]
if ($rptype == "v") {
    puts $f [format "rptype C"]
} else {
    puts $f [format "rptype $s" $rptype]
}
puts $f [format "taper $s" $taper]
puts $f [format "twin $s" $twin]
puts $f [format "width $s" $width]
puts $f [format "type spec"]
puts $f [format "calib_from_calibration no"]
puts $f [format "]"]
puts $f [format "]"]
close $f

set outfile [format "/tmp/dbout%0d%0d" [pid] $instance]
if ($start == "start") {
    set tm [strftime $mytime]
} else {
    set tm [strftime $mytime]
    set tm [expr $tm - [strftime $twin]]
}
if {$info exists net} {
    set pid [exec dbvmeas -exitonerror -outrecno -p $tmpf $time $sta $chan $tm [strftime $twin] $dbname > $outfile &]
} else {
    set pid [exec dbvmeas -exitonerror -outrecno -net $set -p $tmpf $time $sta $chan $tm [strftime $twin] $dbname > $outfile &]
}
destroy .f.scf.go

set status [format "Computing noise spectrum . . . please be patient"]
frame .f.wait
grid .f.wait -row 3 -column 0 -sticky new
grid rowconfigure .f 3 -weight 1
```
Example - dbnoise

• Display is another standalone tcl/tk script widget, named displaynoise, which will display the noise spectra as it is stored in an Antelope database.
Example - dbnoise