Tcl Europe 2003

Table of Contents

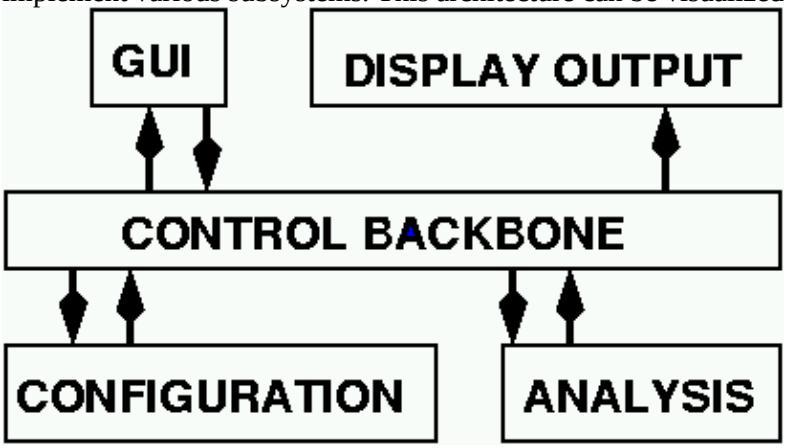
Tcl Europe 2003

Table of Contents

Techniques for developing Tcl and C Applications

Introduction

Many applications are written as a central backbone of code that invokes code from other libraries to implement various subsystems. This architecture can be visualized like this:



Tcl programmers are familiar with using Tcl as the backbone, with compiled extensions like Tk or BLT for the GUI and output, and compiled extensions or standalone applications for the analysis subsystem. This "Tcl as backbone" architecture provides a quick and flexible framework for developing applications.

The Tcl backbone is an excellent choice for many applications, however, it requires the end user to have the proper revisions of all the interpreters and libraries installed on their system. and allows the users to examine, and perhaps modify the application. Exposing the code to the end user is anathema to commercial application developers, who fear competitors reverse−engineering their application, or customers demanding support for modified applications.

Using a compiled language as the backbone provides a solution for all the distribution problems, at the cost of a relatively small amount of development time.

This paper describes several techniques for developing an application with Tcl and using the C API is used to extend the interpreter or embed the interpreter within a C application.

The example application will show how the Tcl/Tk can be used to develop a simple simulation application. I'll describe a pure Tcl solution, a pure C solution, a Tcl script as glue running a C application, Tcl extended with custom C code, and a C mainline using Tcl and Tk for the GUI.

The application models a spaceship landing using a rocket engine. The rocket starts at a given altitude with a given speed and can burn a fixed amount of fuel each second. This is a similar to the old text−oriented **Lunar Lander** game, but a bit simpler. Reduced to pseudo−code, the application looks like this:

```
set initial conditions

while (height > 0) {
    calculate current speed
    calculate current height
    calculate remaining fuel
    report current state
}
```

Pure Tcl solution

A pure Tcl solution is the most portable way to write applications. The same code will run on classic Macintosh systems, OS/X, Windows, X11, and even a Palm Pilot. The downside is that while Tcl has been optimized with a byte−code engine and other performance enhancements, it's still interpreted language, and may not have the speed for all applications.

The limitation of needing to have the appropriate Tcl/Tk interpreter installed on a system in order to run a Tcl/Tk application has been solved by several "wrapper" programs that combine the Tcl/Tk interpreter and application script into a single package. The three best of these are Dennis Labelle's **FreeWrap** (http://freewrap.sourceforge.net/), the **prowrap** (http://tclpro.sourceforge.net/) package originally developed by Scriptics and released as open source (also available as an enhanced version distributed by ActiveState (www.activestate.com), and the **Starkits** (http://trixie.triqs.com/mailman/listinfo/starkit) developed by Jean−Claude Wippler and Steve Landers.

The `prowrap` package provides for code obfuscation by wrapping a byte−code compiled version of the script. FreeWrap and TclKit have hooks that could be used to implement maintaining scripts in an encrypted format, but support for this is not in the primary distributions.

The calculation and simulation loop of a pure Tcl implementation looks like this:

```
#################################################################
# proc calcSpeed {speed mass burn impulse}--
# Return the speed after rocket burn
#    Equation used is:
#  delta_v = g_0 * [delta_t − impulse * ln(m_0/m_1)]
#  g_0:      gravitational acceleration
#  delta_t:  elapsed time (1 second in this calculation)
#  m_0:      initial mass
#  m_1:      mass after burning fuel
# Arguments
#   speed:      Initial speed.
#   mass:       Initial mass of rocket + fuel.
#   burn:       Amount of fuel to burn.
#   impulse:    Amount of thrust generated per fuel unit.
#
# Results
#   No side effects
#
proc calcSpeed {speed mass burn impulse} {
  return [expr {$speed + 9.8 * (1 − $impulse * log($mass/($mass−$burn)))}]
}


proc doSimulation {} {
  global burn

  # Define initial conditions
  set ht    10000.;
  set speed 100.;
  set fuel  10000.;
  set gross 100.;
  set i 0;

  # Use a local variable for burn, so it can be set
  #  to 0 if we run out of fuel.
```

```
  set b $burn
  # Loop until we hit the ground

  while {$ht $gt; 0} {
      set speed  [calcSpeed $speed [expr $gross+$fuel] $b 200]
      set ht [expr $ht − $speed]
      set fuel [expr $fuel − $burn]

      if {$fuel <= 0} {
          set b 0;
          set fuel 0;
      }
      incr i
      showState $i $speed $fuel $ht
  }
}
```

The next step is the GUI to read the burn rate and report results. This application allows the user to set the value for one variable: the amount of fuel to burn per second. This allows the input portion of the GUI to be very simple − a single `scale`widget does the trick.

Syntax: scale *scaleName ?options?*

scaleName The name for this `scale` widget.

?options? There are a many options for this widget. The minimal set is:

−orient orientation
: Whether the scale should be drawn horizontally or vertically: *orientation* may be `horizontal` or `vertical` . The default orientation is vertical.

−length numPixels
: The size of this `scale`. The height for vertical widgets, and the width for horizontal widgets.

−from number
: One end of the range to display. This value will be displayed on the left side (for horizontal `scale` widgets, or top (for vertical `scale` widgets.

−to number
: The other end for the range.

−variable varName
: A variable which will contain the current value of the slider

−resolution number
: The amount to increment the value by when the user clicks on the scale body.

The `−variable` option is supported by all Tk input widgets. The argument to `−variable` is the name of a variable in the global scope which will always contain the current value displayed by the widget. This feature allows an application programmer to divorce analysis code from the GUI. The analytic code only deals with variables, and does not need to know whether a value came from a `scale` widget, a `entry` widget or was read from an html form or file.

Once a user has set the burn value, they need to inform the application that they are ready to run the simulation. This is a classic use for a `button` widget. In this case, the button invokes the `doSimulation` procedure.

Here is the input section of this code:

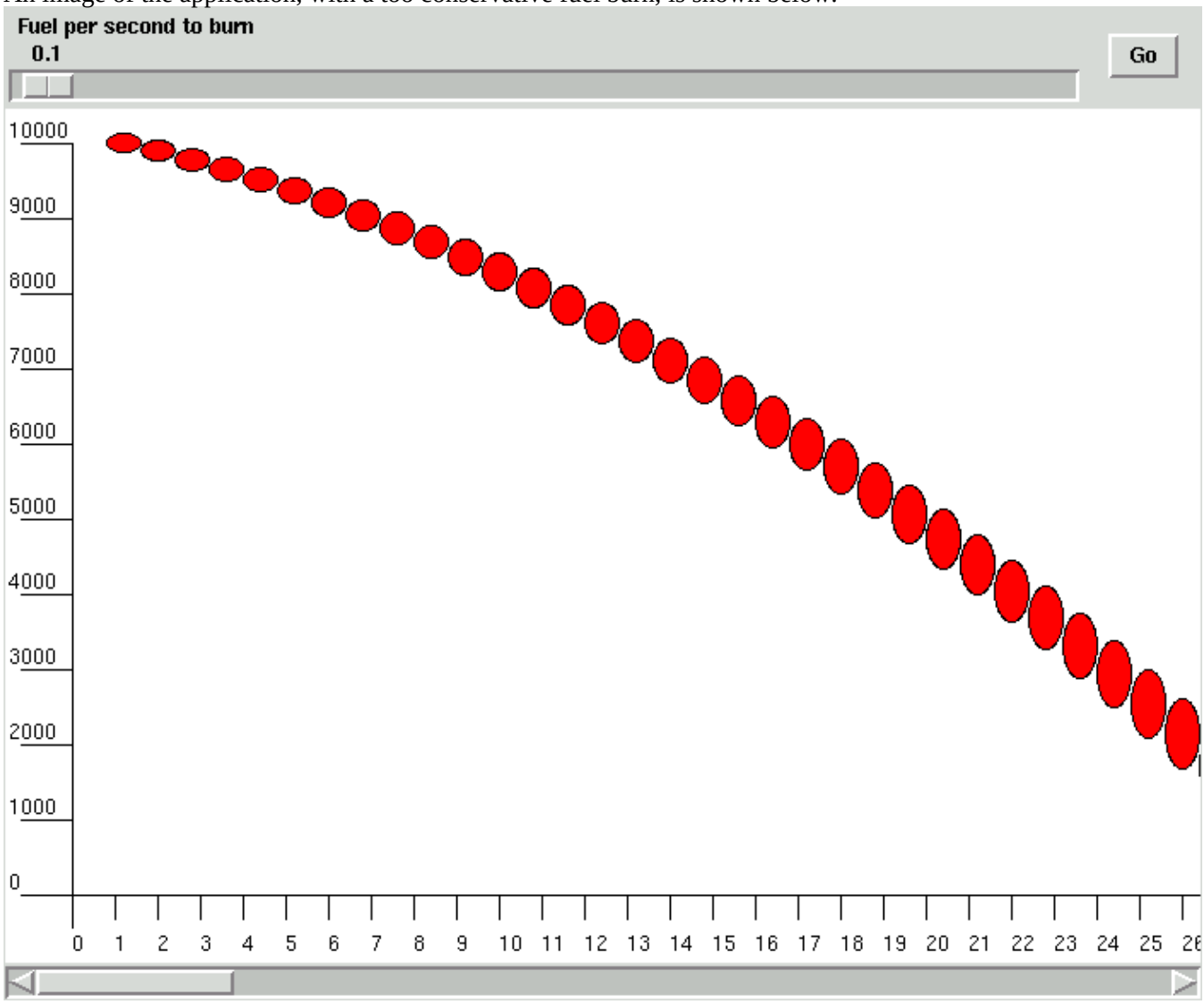
```
scale .s −digits 3 −from 0 −to 10 −label "Fuel per second to burn" \
```

```
    -resolution .1 -showvalue true -length 600 \
    -variable burn -orient horizontal
grid .s -row 1 -column 1 -sticky ew
button .b -text "Go" -command "doSimulation"
grid .b -row 1 -column 2
```

The traditional output for the Lunar Lander program was lines of data displaying the altitude, speed, and remaining fuel for each second of flight time. Lunar Lander was, after all, originally written before the days of graphic monitors.

It's better for modern applications to display information graphically. This application uses the Tk `canvas` widget to draw a simple graph to display the height, speed, fuel and time. The X and Y axis display the time in flight and altitude. The position of the lander is shown with an oval in which the height is proportional to speed, while the width is proportional to the remaining fuel.

An image of the application, with a too conservative fuel burn, is shown below.



The Tk `canvas` widget is a vector oriented drawing surface in which each item displayed is a separate graphic object. These objects have state (size, location, etc) which can be modified by the controlling application. Each object can also have identifiers and actions associated with it.

Syntax: canvas *canvasName ?options?*

canvasName The name for this canvas

?options? Some of the options supported by the canvas widget are:

−background color
: The color to use for the background of this image. The default color is light gray.

−scrollregion boundingBox
: Defines the size of a canvas widget. The bounding box is a list: `left top right bottom` that defines the total area of this canvas, which may be larger than the displayed area.

  These coordinates define the area of a canvas widget that can scrolled into view when the canvas is attached to a scrollbar widget.

  This defaults to `0 0 width height`, the size of the displayed canvas widget.

−height size
: The height of the displayed portion of the canvas. If `−scrollregion` is declared larger than this, and scrollbars are attached to this canvas, this defines the height of the window into a larger canvas.

  The `size` parameter may be in pixels, inches, millimeters, etc.

−width size
: The width of this canvas widget. Again, this may define the size of a window into a larger canvas.

The format of the command to create an object on the canvas is

canvasName create *objectType coordinates ?option value?*

This code snipped below creates the X and Y axis lines and then the ticks and labels

```
# Create the axis lines
.c create line 40 20 40 480
.c create line 40 460 4000 460

# Create and label the ticks on the Y  axis
for {set i 0} {$i <= $height} {set i [expr $i + $height/10]} {
    set y [expr 460 − ($i * .044)]
    .c create text 3 $y −text $i −anchor sw
    .c create line 10 $y 40 $y
}

# Create and label the ticks on the X axis
for {set i 40; set j 0} {$i < 4000 } {incr j; incr i 25} {
    .c create text $i 482 −text $j −anchor nw
    .c create line $i 460 $i 475
}
```

The displayed canvas widget is a window into a much larger area. A script can attach the canvas to a scrollbar, allowing the user to pan around and view sections of the canvas that are not otherwise visible.

Tcl/Tk handles event driven style programing with callback procedures. The `scrollbar` and `canvas` widgets have predefined commands to interact with each other. When a `scrollbar` widget changes state (someone moves the slider), it will evaluate the script that was registered with it. This command will cause the

`canvas` to reconfigure itself to match the scrollbar. When a `canvas` widget changes state, it will evaluate a similar command to cause the `scrollbar` to modify itself.

This code snippet creates the canvas and scrollbar, and links them with the `−command` and `−xscrollcommand` options.

```
    canvas .c −height 500 −width 700 −background white \
        −xscrollcommand {.sb set}
    grid .c −row 2 −column 1 −columnspan 2

    scrollbar .sb −orient horizontal −command {.c xview}
    grid .sb −row 3 −column 1 −columnspan 2 −sticky ew
```

Each displayed item on the canvas is a distinct item with a unique identifier. This identifier is returned when an object is created. Each object can also have a scripts bound to them, to be evaluated when an event occurs over the object. The events include `<Enter>` and `<Leave>` events as well as button events.

This procedure draws a red oval at the appropriate location. The height is proportional to the speed of the lander at that time, while the width is proportional to the amount of fuel. When the user left clicks on an oval, it will invoke the `showDetails` procedure to display the exact speed, fuel and height at that time.

```
proc showState {i speed fuel ht} {
   # Scale values as necessary
   set x [expr $i * 20 + 40]
   set y [expr 460 − ($ht * .044)]
   set y2 [expr $y − ($speed / 10)]

   # Create the oval
   set item [.c create oval $x $y \
       [expr $x + $fuel/50.0] $y2 \
       −fill red]

    # bind a script to show the details to the oval.
   .c bind $item <Button−1> "showDetails [expr $x + 30] $y2 $speed $fuel $ht"
}
```

Objects on a canvas can be accessed by their unique identifier, or they can have one or more tags associated with them. A single tag may be assigned to several objects, and an object can have many tags.

The `showDetails` procedure identifies the object created with the tag `info`. This allows text to be deleted without knowing its unique identifier.

```
proc showDetails {x y speed fuel ht} {
   .c delete info
   set s [format %7.2f $speed]
   set h [format %7.2f $ht ]
   .c create text $x $y −tags info \
       −text "Speed: $s Height: $h Fuel: $fuel" \
       −anchor w
}
```

Pure C solution

A non−GUI C solution looks like this. It accepts the burn amount on the command line, and prints out the speed, height, remaining fuel, etc for each second of flight.

```
#include <stdlib.h>
#include <math.h>

/*******************************************************/
// float calcSpeed(float speed, float mass, float burn, float impulse)
// Return the speed after rocket burn
//    Equation used is:
//  delta_v = g_0 * [delta_t − impulse * ln(m_0/m_1)]
//  g_0:      gravitational accelleration
//  delta_t:  elapsed time (1 second in this calculation)
//  m_0:      initial mass
//  m_1:      mass after burning fuel
// Arguments
//   speed:      Initial speed.
//   mass:       Initial mass of rocket + fuel.
//   burn:       Amount of fuel to burn.
//   impulse:    Amount of thrust generated per fuel unit.
//
// Results
//   No side effects
//

float calcSpeed(float speed, float mass, float burn, float impulse) {
    speed += 9.8 * (1 − impulse * log(mass/(mass−burn)));
    return speed;
}

main(int argc, char *argv[]) {
  // Simulator variables
  float ht, speed, fuel, gross;
  float burn;
  float impulse = 200.0;
  int i;

  // Hardcoded initial values

  ht = 10000.0;
  speed = 100.0;
  fuel = 1000.0;
  gross = 900.0;

  burn = (double) atof(argv[1]);

  // Simulation loop.
  // Calculate speed, remaining mass and height at 1 second intervals

  i = 0;
  while (ht > 0) {
      printf("%d) speed: %8.2f fuel: %8.2f height: %8.2f\n",
          i++, speed, fuel, ht);
      speed = calcSpeed(speed, gross+fuel, burn, impulse);
      fuel = fuel − burn;
      if (fuel <= 0) {
          burn = 0;
          fuel = 0;
      }
      ht = ht − speed;
  }
}
```

A survivable re−entry resembles this:

```
%$gt; lander 8.0332
0> speed:   100.00 fuel:  1000.00 height: 10000.00
1> speed:   101.50 fuel:   991.97 height:  9898.50
2> speed:   102.96 fuel:   983.93 height:  9795.55
...
97> speed:    16.05 fuel:   220.78 height:    20.17
98> speed:    11.76 fuel:   212.75 height:     8.41
99> speed:     7.35 fuel:   204.71 height:     1.06
```

Tcl as Glue

Using Tcl as a glue is a common paradigm for operating systems that support the concept of standard input/output (IE, not classic MacOS). Under Mac OS/X, Windows, or a posix style operating system, the pure C application (`lander`) can be executed, and the output parsed using the `split` command to split the output string into list elements wherever a newline occurs and the `scan` command, which behaves similar to the "C" `sscanf` command.

```
proc doSimulation {} {
    global burn
    set return [exec lander $burn]
    foreach l [split $return \n] {
        scan $l "%d> speed:  %f fuel: %f height: %f" time speed fuel ht
        showState $time $speed $fuel $ht
    }
}
```

Tcl Mainline with a "C" functions

The Tcl interpreter was designed to be extended with compiled functions as well as allowing itself to be embedded into larger applications. As such, the interpreter has a clean API for linking external modules.

There are three primary techniques for developing C code extensions:

- writing an extension by hand.
- using the automated extension generator SWIG (www.swig.org).
- using the CriTcl (www.equi4.com/critcl) package for embedding C code into a Tcl script.

Generating a Tcl extension by hand is the most versatile technique, and not difficult. The Tcl sample extension (available from SourceForge and www.tcl.tk) provides a framework to which custom code can easily be added.

When extending Tcl with existing libraries or code bases, the SWIG package developed by David Beazley makes the process of developing an extension relatively painless.

SWIG uses a modified include file with function and structure definitions to generate a the "C" glue that links a library into a Tcl interpreter. SWIG can be used to generate the interface code for Perl, Python, Tcl/Tk, Guile, MzScheme, Ruby, Java, PHP, or CHICKEN, and runs on several platforms. including Posix, Mac Classic, OS/X and MS Windows.

The easiest way to embed small "C" functions is to use the CriTcl package (developed by Jean−Claude Wippler and Steve Landers). which supports embedding "C" code into a Tcl script. The CriTcl package can generate the appropriate "C" code, and compile an extension on the fly. CriTcl is currently only supported for

the gcc compiler, and can be used on Posix, Mac OS/X and MS−Windows with mingw.

Like most modern packages, the CriTcl commands are encapsulated within a namespace. Following the Tcl convention, the namespace and package use the same name: `critcl`. The minimal set of commands for using CriTcl are:

`critcl::cproc`	Define a C function.
`critcl::clibraries`	Define libraries necessary for the link phase.
`critcl::ccode`	Define ancillary code to be placed within the C source file. This lets you declare include files, and preprocessor macros.

The `critcl::clibraries` and `critcl::ccode` procedures accept a simple list of values for arguments, while the `critcl::cproc` procedure has a somewhat more complex argument list.

Syntax:`critcl::cproc` *`name argList returnType body`*

`name`	The name of this function.
`argList`	The list of arguments to the C function. Format is `{type1 name1 type2 name2...}` in which *`type`* is the type of variable (`int, float,` etc.) and *`name`* is the the name of this variable.
`returnType`	The type of value returned by the function.
`body`	The body of the C function.

This set of code will create a Tcl extension with the `calcSpeed` function.

```
package require critcl
critcl::clibraries −lm
critcl::ccode {#include <math.h>}

critcl::cproc calcSpeed \
    {float speed  float mass  float burn  float impulse} \
    float  \
    {
      speed += 9.8 * (1 − impulse * log(mass/(mass−burn)));
      return speed;
    }
```

However a Tcl extension is generated, the C code will include calls to the Tcl API to create and register new Tcl commands, to set and retrieve Tcl script simple variable values, and perhaps to manipulate Tcl lists and arrays.

The first versions of the Tcl interpreter represented data as an ASCII string internally. This worked, but was slow, particularly for math operations, in which case the string needed to be converted to an integer, a math operation performed, and then the result converted back to a string.

With the 8.0 release, Tcl moved to an internal representation using a `Tcl_Obj` structure that contains both the string representation and a native format representation for the data. Since operations on data tend to come in batches; string style interactions during data input, math operations during analysis, then string operations while generating a final report, this greatly improved Tcl's performance.

Most of the C API now comes in two flavors, original and Tcl_Obj. The older style API is retained for legacy extensions, but the new, `Tcl_Obj` based API is preferred for new code.

The `Tcl_CreateObjCommand` procedure adds a new command to the Tcl interpreter.

Syntax: `int` **`Tcl_CreateObjCommand`** *`(interp, cmdName, func, clientData, deleteFunc)`*

`Tcl_CreateObjCommand`	Registers a new command with the Tcl interpreter. These actions are performed within the Tcl interpreter: • Register *`cmdName`* with the Tcl interpreter. • Define `clientData` data for the command. • Define the function to call when the command is evaluated. • Define the command to call when the command is destroyed.
`Tcl_Interp *`*`interp`*	This is a pointer to the Tcl interpreter. It is required by all commands that need to interact with the interpreter state. Your extensions will probably just pass this pointer to Tcl library functions that require a Tcl interpreter pointer. Your code should not attempt to manipulate any components of the interpreter structure directly.
`char *`*`cmdName`*	The name of the new command, as a NULL terminated string.
`Tcl_ObjCmdProc *`*`func`*	The function to call when *`cmdName`* is encountered in a script.
`ClientData` *`clientData`*	A value that will be passed to *`func`* when the interpreter encounters *`cmdName`* and calls *`func`*. The `ClientData` type is a word, which on most machines is the size of a pointer. You can allocate memory and use this pointer to pass an arbitrarily large data structure to a function.
`Tcl_CmdDeleteProc *`*`deleteFunc`*	A pointer to a function to call when the command is deleted. If the command has some persistent data object associated with it, this function should free that memory. If you have no special processing, set this pointer to NULL, and the Tcl interpreter will not register a command deletion procedure.

The API functions that allow a "C" code to interact with Tcl scripts and variables are discussed in the next section.

"C" Mainline with Tcl script

Programmers accustomed to working with other graphics toolkits are more comfortable with a main "C" procedure that invokes graphics calls as necessary. While a program can be constructed using the Tk library as a graphics library, the more versatile architecture is to evaluate short Tcl scripts from the C mainline.

The flow for this example is:

```
create and initialize interpreter
create GUI
wait for "Go" button to be pressed
run simulation and update output display
wait for exit.
```

The first step in any "C"/Tcl hybrid is to create and initialize the interpreter. A Tcl interpreter is composed of a state structure and the functions to manipulate the structure. Thus, and application can have multiple independent interpreters in operation at any time. The code that manipulates the state structures is all thread safe, allowing multiple interpreters to live in multiple threads.

The state structure is created and initialized by the `Tcl_CreateInterp` function, which allocates memory for the interpreter and returns a pointer to the structure. This code snippet demonstrates creating an interpreter.

```
       #include <tcl.h>
...
       Tcl_Interp *interp;
       interp = Tcl_CreateInterp();
```

The Tcl environment is built from a collection of compiled functions and scripts. Part of creating an interpreter is to load the scripts that will assign values to the global variables and define the non−compiled commands.

After the Tcl interpreter is initialized it can load the Tk initialization scripts to define extra graphics widgets, etc. The two commands which perform the initialization are `Tcl_Init` and `Tk_Init`.

Syntax:`Tcl_Init` *`(interp)`*

Syntax:`Tk_Init` *`(interp)`*

`interp` A pointer to the Tcl interpreter, as returned by Tcl_CreateInterp();
Once the interpreter is initialized, it's ready to accept one or more script to evaluate. The `Tcl_Eval` function passes a script to the interpreter. This can be as simple as a single command, or an arbitrarily long string (within limits of the compiler, etc).

Syntax:`Tcl_Eval` *`(interp, script)`*

`interp` A pointer to the Tcl interpreter, as returned by `Tcl_CreateInterp();`
`script` A NULL terminated string of Tcl commands.
This line will load a Tcl script from the file `config.tcl` , and then invoke the `inputinitialValues` procedure that is defined in that file.

```
  Tcl_Eval(interp, "source config.tcl; inputInitialValues");
```

Like other windowing systems, Tcl uses an event loop to process events. These events include changes in the display (like creating a widget), as well as user events like moving a mouse or clicking a key.

If the program logic were controlled within the Tcl script, the application would start the application and then pass control to the event loop by calling `Tk_Main`.

This example is designed to work in three phases − the GUI controls the application until the user has set the burn rate, then the simulation code runs in a loop until the rocket lands, updating the GUI as necessary, and finally, the code returns to a loop until it is exited.

This requires that the mainline "C" code be able to determine when the user has clicked the "Go" button. A `button` can register a Tcl procedure to be evaluated when it is clicked, but it can not register a pure "C" function. However, the script associated with a button can modify a Tcl variable to be used as a flag by the

"C" mainline code.

One solution to the problem is that the `initializeInputValues` procedure sets a global variable (in this case, named `ready`) to 0, and after the mainline "C" code processes each event, it checks that variable to see if it has been changed.

The `Tcl_DoOneEvent` function will enter the event loop and process one event. If there are no events, the loop will (by default) wait until an event is available to process.

Syntax:`Tcl_DoOneEvent` *`(flags)`*

`flags` A bitmap of flags to control which events will be processed and whether the event loop should wait. Normally, this field will be a 0, to wait until an event is available and process any available event.

These flags include:

TCL_WINDOW_EVENTS
: Do not process window events.

TCL_FILE_EVENTS
: Do not process file events.

TCL_TIMER_EVENTS
: Do not process timer events.

TCL_IDLE_EVENTS
: Do not process idle events.

TCL_DONT_WAIT
: Return immediately.

This snippet will transfer control from a set of "C" to the Tcl event loop, and thus to a running Tcl script.

```
while (1) {
    Tcl_DoOneEvent(0);
}
```

There are two steps to retrieving a value from the Tcl interpreter. The `Tcl_GetVar2Ex` function will return a pointer to the `Tcl_Obj` for this variable, and other functions will extract the native value from the object. If your script needs the string representation, it can use the `Tcl_GetVar2` function instead of `Tcl_GetVar2Ex`.

Syntax:`Tcl_Obj *Tcl_GetVar2Ex` *`(interp, name1, name2, flags )`*

Syntax:`CONST char *Tcl_GetVar2` *`(interp, name1, name2, flags )`*

`interp` A pointer to the Tcl interpreter, as returned by Tcl_CreateInterp();

`name1` The name of a Tcl variable.

`name2` The index if the variable is an associative array, else NULL.

`flags` A bit−mapped set of flags to control the scope in which the variable name will be resolved. Possible values include:

TCL_GLOBAL_ONLY
: Look only in the global scope.

TCL_NAMESPACE_ONLY

Look only in the current namespace.

The functions that extract native format values from a `Tcl_Obj` are `Tcl_GetLongFromObj`, `Tcl_GetIntFromObj` and `Tcl_GetDoubleFromObj`. These functions return `TCL_OK` on success, or `TCL_ERROR` if the conversion fails (perhaps the string representation is not numeric). The `Tcl_GetStringFromObj` function can never fail (all data can be represented as a string) and returns a pointer to the data string.

Syntax:`Tcl_GetLongFromObj` *`(interp, objPtr, longPtr)`*

Syntax:`Tcl_GetIntFromObj` *`(interp, objPtr, intPtr)`*

Syntax:`Tcl_GetDoubleFromObj` *`(interp, objPtr, doublePtr)`*

Syntax:`Tcl_GetStringFromObj` *`(objPtr, intPtr)`*

`interp`	A pointer to the Tcl interpreter, as returned by Tcl_CreateInterp();
`objPtr`	A pointer to a `Tcl_Obj`, as returned by `Tcl_GetVar2Ex`
`intPtr/longPtr/doublePtr`	A pointer to the appropriate "C" variable to receive the data.

For example, this snippet attempts to extract an integer from a Tcl_Object, and reports a problem if it fails.

```
char *str;
int length;
int intValue;
...
if {TCL_ERROR  == Tcl_GetIntFromObj(interp, objPtr, {
  str = Tcl_GetStringFromObj(objPtr,
  printf "Not Integer Value: %s\n", str);
  }
...
```

This code snippet loops until the user clicks on the **Go** button.

```
  Tcl_Obj *readyObj;           /* Tcl_Obj to hold pointer to Tcl var */
  long ready;
...
  readyObj = Tcl_GetVar2Ex(interp, "ready", NULL, TCL_GLOBAL_ONLY);
  Tcl_GetLongFromObj(interp, readyObj,

  while (ready == 0) {
      // Tcl_DoOneEvent takes one event from the stack and processes it.
      Tcl_DoOneEvent(0);
      readyObj = Tcl_GetVar2Ex(interp, "ready", NULL, TCL_GLOBAL_ONLY);
      Tcl_GetLongFromObj(interp, readyObj,
  }
```

This looks like a round−robin loop, but it is not. Within the Tcl_DoOneEvent function, the application waits using the `select` system call, using no CPU time.

Also note that the `Tcl_GetVar2Ex` call is included within the loop, instead of only acquiring the `Tcl_Obj` once and then checking for the value within the `Tcl_Obj`

The Tcl byte code compiler will perform several optimizations with the code it generates. One of these is to reuse variables whenever possible. Within the setup procedure, the command `set ready 0` makes an entry for `ready` in the variable name lookup table, and links that variable to the `Tcl_Obj` that contains a constant

`0` (zero). When the button is pressed, and the command `set ready 1` is evaluated, rather than change the value of the constant zero, Tcl changes the pointer associated with the `ready` entry in the variable table to point to the `Tcl_Obj` that contains the constant `1` . If the value of `ready` were changed with a command like `incr ready` Tcl would create a new `Tcl_Obj` to contain a non−constant value, and would assign the `ready` entry in the variable table to that `Tcl_Obj`.

Once the `ready` variable is non−zero, the "C" mainline code can retrieve the value for `burn`, and start the simulation.

This code snippet includes error checking while retrieving the value for `burn`.

```
  // Get the burnObj from the Tcl script and extract the double value.

  burnObj = (Tcl_Obj *) Tcl_GetVar2Ex(interp, "burn", NULL, TCL_GLOBAL_ONLY);

  if (burnObj == NULL) {
      printf("Tcl global var 'burn' is not defined.\n");
      exit(−1);
  }

  if (TCL_OK != Tcl_GetDoubleFromObj(interp, burnObj, {
      printf("Bad burn value: %s\n", Tcl_GetString(burnObj));
      exit(−1);
  }
```

At this point, the "C" mainline code can do the simulation and display the results. The `Tcl_Eval` command can accept any NULL terminated string, which allows our "C" code to build Tcl commands to be evaluated on the fly with `sprintf`.

The snippet below checks the return from `Tcl_Eval` to confirm that the script was evaluated properly. If `Tcl_Eval` does not return `TCL_OK` the script failed, and an error message will be placed in the global variable `errorInfo`. The error string can be reported with the `Tcl_GetVar(interp, "errorInfo", TCL_GLOBAL_ONLY));` function call, which returns a string.

The `Tcl_DoOneEvent` at the end of the loop will process the window event and display each point as it is calculated. In this application, there is no visible delay in the calculations, but in a more complex simulation, a user might want to see each iteration as it occurs. This also provides a hook for the "C" mainline code to examine the state of various Tcl variables, and interact with the user as necessary.

```
  while (ht > 0) {
      i++;
      // Simulation calculations.
      speed = calcSpeed(speed, gross+fuel, burn, impulse);

      // Generate a string to evaluate as a Tcl command
      sprintf(cmd, "showState %d %f %f %f", i, speed, fuel, ht);
      if (Tcl_Eval(interp, cmd) != TCL_OK) {
          printf("FAILED to run '%s'\n%s", cmd, \
              Tcl_GetVar(interp, "errorInfo", TCL_GLOBAL_ONLY));
          exit(−1);
      }
      Tcl_DoOneEvent(0);
}
```

The final step in this application is to transfer control to the Tcl script, allowing the user to scroll the canvas back and forth. A more complex application would have hooks to re−run the simulation, clear the canvas, etc.

```
# Process event loop until program exits.

while (1) {
    Tcl_DoOneEvent(0);
}
```

Summary

This architecture provides a great deal of versatility for the developer, since the GUI portions are separated from the compiled analysis code. If experience shows that the `scale` widget is not appropriate, another input mechanism can be added. If the output display needs to be modified, again, this can be done without recompiling. The Tcl interpreter provides a versatile platform for developing multi−platform applications using a variety of architectures.

One powerful design pattern is to develop all non−platform specific code in a compiled language like C or C++ (including the backbone), and the platform specific parts (GUI, file system interactions, etc) as Tcl scripts, either embedded within the application as strings, or in ancillary files.

For GUI work, Tcl is best integrated with Tk, which also has multi−platform support. If necessary, Tcl can also be integrated with other graphics packages such as OpenGL(either using SWIG (www.cise.ufl.edu/depot/doc/swig/Examples/OpenGL/Tcl/) or the `GLxwin` extension), Qt (see http://www.staff.city.ac.uk/~sa346/Ktk.html), gtk (see gnocl http://www.dr−baum.net/gnocl/), XVT, and others.

Source Listings

Common GUI: GUI.tcl

```
#####################################################################
# proc buildGUI {height}--
#    Create the application GUI
# Arguments
#   height      : The initial height of the lander
#
# Results
#   Creates scale widget, attached to "burn" variable.
#   Creates button to initiate simulation.
#   Creates a canvas and displays an empty X/Y graph

proc buildGUI {height} {
    global ready
    set ready 0
    scale .s -digits 3 -from 0 -to 10 -label "Fuel per second to burn" \
        -resolution .1 -showvalue true -length 600 \
        -variable burn -orient horizontal
    grid .s -row 1 -column 1 -sticky ew
    button .b -text "Go" -command "doSimulation"
    grid .b -row 1 -column 2

    canvas .c -height 500 -width 700 -background white \
        -xscrollcommand {.sb set}
    grid .c -row 2 -column 1 -columnspan 2

    scrollbar .sb -orient horizontal -command {.c xview}
    grid .sb -row 3 -column 1 -columnspan 2 -sticky ew

    # Create the axis lines

    .c create line 40 20 40 480
    .c create line 40 460 4000 460

    # Create and label the ticks on the Y  axis
    for {set i 0} {$i <= $height} {set i [expr $i + $height/10]} {
        set y [expr 460 - ($i * .044)]
        .c create text 3 $y -text $i -anchor sw
        .c create line 10 $y 40 $y
    }

    # Create and label the ticks on the X axis
    for {set i 40; set j 0} {$i < 4000 } {incr j; incr i 25} {
        .c create text $i 482 -text $j -anchor nw
        .c create line $i 460 $i 475
    }

    .c configure -scrollregion {0 0 4000 500}

}

#####################################################################
# proc showState {i speed fuel ht}--
#    Displays an oval on the canvas scaled for speed and weight and
#    located for height and  time.
# Arguments
#   i           The time in seconds
```

```
#   speed       The speed in meters/second
#   fuel        The mass of fuel remaining
#   ht          The height of the lander in meters
# Results
#   Updates the display.
#   Creates a binding on the new oval to display details.

proc showState {i speed fuel ht} {
   # Scale values as necessary
   set x [expr $i * 20 + 40]
   set y [expr 460 - ($ht * .044)]
   set y2 [expr $y - ($speed / 10)]

   # Create the oval
   set item [.c create oval $x $y \
       [expr $x + $fuel/50.0] $y2 \
       -fill red]

    # bind a script to show the details to the oval.
   .c bind $item <Button-1> "showDetails [expr $x + 30] $y2 $speed $fuel $ht"
}


#####################################################################
# proc showDetails {x y speed fuel ht}--
#    Displays details associated with a point on the graph
# Arguments
#   x, y        The location at which to display the text
#   speed       The speed of the lander in meters/second
#   fuel        The mass of remaining fuel
#   ht          The height of the lander in meters
# Results
#
#
proc showDetails {x y speed fuel ht} {
   .c delete info
   set s [format %7.2f $speed]
   set h [format %7.2f $ht ]
   .c create text $x $y -tags info \
       -text "Speed: $s Height: $h Fuel: $fuel" \
       -anchor w
}
```

Pure Tcl: lander.tcl

```
source GUI.tcl

#####################################################################
# proc calcSpeed {speed mass burn impulse}--
# Return the speed after rocket burn
#    Equation used is:
#  delta_v = g_0 * [delta_t - impulse * ln(m_0/m_1)]
#  g_0:      gravitational acceleration
#  delta_t:  elapsed time (1 second in this calculation)
#  m_0:      initial mass
#  m_1:      mass after burning fuel
# Arguments
#   speed:      Initial speed.
#   mass:       Initial mass of rocket + fuel.
#   burn:       Amount of fuel to burn.
```

```
#   impulse:    Amount of thrust generated per fuel unit.
#
# Results
#   No side effects
#
proc calcSpeed {speed mass burn impulse} {
  return [expr {$speed + 9.8 * (1 − $impulse * log($mass/($mass−$burn)))}]
}

#####################################################################
# proc doSimulation {}−−
#    doSimulation
# Arguments
#   NONE
#
# Results
#   Performs simulation, modifies all state variables,
#   Invokes showState to update display

proc doSimulation {} {
  global burn

  # Define initial conditions
  set ht    10000.;
  set speed 100.;
  set fuel  1000.;
  set gross 900.;
  set i 0;

  # Use a local variable for burn, so it can be set
  #  to 0 if we run out of fuel.

  set b $burn
  # Loop until we hit the ground

  while {$ht > 0} {
      set speed  [calcSpeed $speed [expr $gross+$fuel] $b 200]
      set ht [expr $ht − $speed]
      set fuel [expr $fuel − $burn]

      if {$fuel <= 0} {
          set b 0;
          set fuel 0;
      }
      incr i
      showState $i $speed $fuel $ht
  }
}

buildGUI 10000
```

Pure C: lander.c

```
#include <stdlib.h>
#include <math.h>

/*******************************************************/
// float calcSpeed(float speed, float mass, float burn, float impulse)
// Return the speed after rocket burn
//    Equation used is:
//  delta_v = g_0 * [delta_t − impulse * ln(m_0/m_1)]
```

```
//  g_0:      gravitational acceleration
//  delta_t:  elapsed time (1 second in this calculation)
//  m_0:      initial mass
//  m_1:      mass after burning fuel
// Arguments
//   speed:      Initial speed.
//   mass:       Initial mass of rocket + fuel.
//   burn:       Amount of fuel to burn.
//   impulse:    Amount of thrust generated per fuel unit.
//
// Results
//   No side effects
//

float calcSpeed(float speed, float mass, float burn, float impulse) {
    speed += 9.8 * (1 − impulse * log(mass/(mass−burn)));
    return speed;
}

main(int argc, char *argv[]) {
  // Simulator variables
  float ht, speed, fuel, gross;
  float burn;
  float impulse = 200.0;
  int i;

  // Hardcoded initial values

  ht = 10000.0;
  speed = 100.0;
  fuel = 1000.0;
  gross = 900.0;

  burn = (double) atof(argv[1]);

  // Simulation loop.
  // Calculate speed, remaining mass and height at 1 second intervals

  i = 0;
  while (ht > 0) {
      printf("%d> speed: %8.2f fuel: %8.2f height: %8.2f\n",
          i++, speed, fuel, ht);
      speed = calcSpeed(speed, gross+fuel, burn, impulse);
      fuel = fuel − burn;
      if (fuel <= 0) {
          burn = 0;
          fuel = 0;
      }
      ht = ht − speed;
  }
  exit(0);
}
```

Tcl as glue: landerGlue.tcl

```
source GUI.tcl

proc doSimulation {} {
    global burn
    set return [exec lander $burn]
```

```
    foreach l [split $return \n] {
        scan $l "%d> speed:  %f fuel: %f height: %f" time speed fuel ht
        showState $time $speed $fuel $ht
    }
}

buildGUI 10000
```

Tcl with Embedded C: landerCriTcl.tcl

```
source GUI.tcl

package require critcl
critcl::clibraries −lm
critcl::ccode {#include <math.h>}

critcl::cproc calcSpeed \
    {float speed  float mass  float burn  float impulse} \
    float  \
    {
      speed += 9.8 * (1 − impulse * log(mass/(mass−burn)));
      return speed;
    }
######################################################################
# proc doSimulation {}−−
#    doSimulation
# Arguments
#   NONE
#
# Results
#   Performs simulation, modifies all state variables,
#   Invokes showState to update display

proc doSimulation {} {
  global burn

  # Define initial conditions
  set ht    10000.;
  set speed 100.;
  set fuel  1000.;
  set gross 900.;
  set i 0;

  # Use a local variable for burn, so it can be set
  #  to 0 if we run out of fuel.

  set b $burn
  # Loop until we hit the ground

  while {$ht > 0} {
      set speed  [calcSpeed $speed [expr $gross+$fuel] $b 200]
      set ht [expr $ht − $speed]
      set fuel [expr $fuel − $burn]

      if {$fuel <= 0} {
          set b 0;
          set fuel 0;
      }
      incr i
      showState $i $speed $fuel $ht
  }
```

```
}


buildGUI 10000
```

C with Embedded Tcl: landerEmbed.c

```
#include <stdlib.h>
#include <math.h>
#include <tcl.h>
#include <tk.h>

//  float calcSpeed(float speed, float mass, float burn) {
//    Return the speed of lander after burning rockets for 1 time unit.
//
//  speed   initial speed of lander
//  burn    mass of fuel to burn during this time interval.
//  mass    total mass of the lander plus fuel.
//
//  No State change

float calcSpeed(float speed, float mass, float burn, float impulse) {
    //   delta_v = g_0 * [delta_t − impulse * ln(m_0/m_1)]
    //  time is 1 second

    speed += 9.8 * (1 − impulse * log(mass/(mass−burn)));

    return speed;
}

main(int argc, char *argv[]) {
  // Simulator variables
  float ht, speed, fuel, gross;
  double burn;
  int i, rtn;
  float impulse = 200.0;

  // Tcl 'glue' variables
  Tcl_Interp *interp;         /* Interpreter for application. */
  Tcl_Obj *burnObj;           /* Tcl_Obj to hold pointer to Tcl var */
  Tcl_Obj *readyObj;           /* Tcl_Obj to hold pointer to Tcl var */
  long ready;
  char cmd[128];              /* string to build tcl command in */


  // Create the interp and initialize it.

  Tcl_FindExecutable(argv[0]);
  interp = Tcl_CreateInterp();

  if (Tcl_Init(interp) == TCL_ERROR) {
      return TCL_ERROR;
  }

  if (Tk_Init(interp) == TCL_ERROR) {
      return TCL_ERROR;
  }

// Don't enter Tk_Main − that will go into event loop and not return.
// Use Tcl_DoOneEvent to invoke event loop.
```

```
// Load the Tcl script file, and
// run the "inputInitialValues" proc to input the value for 'burn'

rtn = Tcl_Eval(interp, "source config.tcl; inputInitialValues");
if (rtn != TCL_OK) {
    printf("Failed Tcl_Eval: %d \n%s\n", rtn,
       Tcl_GetVar(interp, "errorInfo", TCL_GLOBAL_ONLY));
    exit(−1);
}


// A Tcl variable 'ready' will be set when the user clicks the 'go'
// button.  It is initialized to 0.

readyObj = Tcl_GetVar2Ex(interp, "ready", NULL, TCL_GLOBAL_ONLY);
Tcl_GetLongFromObj(interp, readyObj,

while (ready == 0) {
    // Tcl_DoOneEvent takes one event from the stack and processes it.
    Tcl_DoOneEvent(0);
    readyObj = Tcl_GetVar2Ex(interp, "ready", NULL, TCL_GLOBAL_ONLY);
    Tcl_GetLongFromObj(interp, readyObj,
}

// Get the burnObj from the Tcl script and extract the double value.

burnObj = (Tcl_Obj *) Tcl_GetVar2Ex(interp, "burn", NULL, TCL_GLOBAL_ONLY);

if (burnObj == NULL) {
    printf("Tcl global var 'burn' is not defined.\n");
    exit(−1);
}

if (TCL_OK != Tcl_GetDoubleFromObj(interp, burnObj, {
    printf("Bad burn value: %s\n", Tcl_GetString(burnObj));
    exit(−1);
}

// Hardcoded initial values
ht = 10000.0;
speed = 100.0;
fuel = 1000.0;
gross = 900.0;

// Simulation loop.
// Calculate speed, remaining mass and height at 1 second intervals

i = 0;
while (ht > 0) {
    i++;
    // Generate a string to evaluate as a Tcl command
    sprintf(cmd, "showState %d %f %f %f", i, speed, fuel, ht);
    if (Tcl_Eval(interp, cmd) != TCL_OK) {
        printf("FAILED to run '%s'\n%s", cmd, \
            Tcl_GetVar(interp, "errorInfo", TCL_GLOBAL_ONLY));
        exit(−1);
    }
    Tcl_DoOneEvent(0);

    // Simulation calculations.
    speed = calcSpeed(speed, gross+fuel, burn, impulse);
    fuel = fuel − burn;
```

```
      if (fuel <= 0) {
          burn = 0;
          fuel = 0;
      }
      ht = ht − speed;
  }

  while (1) {
      Tcl_DoOneEvent(0);
  }

}
```

Embedded Tcl Config File: config.tcl

```
source GUI.tcl

proc inputInitialValues {} {
    global ready
    set ready 0

    buildGUI 10000
}

proc doSimulation {} {
    global ready
    set ready 1
}
```