

Cross-platform programming:
Tcl/Tk for Mac Classic, OS X, Windows and Unix
Patterns for Applications
Clif Flynt
clif@cflynt.com

Abstract

To gain wide acceptance, an application must run under Windows, Macintosh and X-11/Posix style operating systems. The largest obstacle to this is the very different GUI styles and libraries used on the 3 platforms. Tcl/Tk provides a platform independent GUI and program flow models that work well with a number of design patterns.

Introduction

In 1960, Connie Francis advised girls to go *Where the Boys Are*. This advice is still appropriate, especially for software developers - in order to gain wide acceptance, an application has to run on the most popular platform, as well as the best platforms. The problem for the application developer is to determine which tools and techniques will best allow them to develop on their favorite platform, and then painlessly port their application to other platforms.

This paper examines some architecture and tool decisions that affect how easily an application can be ported to other architectures and shows examples of how to construct a simple GUI driven simulation program using different design paradigms.

A modern application can be considered to have two components - the User Interface, and the analysis code (ie, everything else). These design paradigms can be ordered by how tightly coupled these two components are. This leads to several architectural styles that can be used to construct applications, including:

- 1 Object Oriented design with analysis code as methods of UI objects.

This architecture is encouraged by the Microsoft Visual Studio IDE

The class structure is easy to develop from story-board use cases, but can be hard to extend and modify. The analysis code, which should be the most reusable component, becomes integrated with the GUI, the least portable component.

- 1 OO with separate object trees for analysis and UI.

This architecture allows the analysis objects to be easily reused.

This model is encouraged by the wxWindows package by using XML resources to define the GUI.

The class structure can be designed using multiple inheritance, in which the main application class inherits from the top level GUI and analysis class, or with aggregation, in which the main application class contains instantiations of the GUI and analysis objects.

- 1 A single flow 'main' with GUI and analysis in separate libraries.

The mainline code makes calls to the event loop to receive events, and processes them in a loop.

This architecture is common in applications using toolkits that are essentially widget drawing tools, rather than GUI environments, Applications built around the X11 Xt and Andrew toolkits use this paradigm, as do applications built with XVT

- 1 An event handler with GUI and Analysis in separate modules.

The mainline code sets up a collection of callback functions to handle events, and then passes control to

the event handler to route keystrokes and mouse movements.

The Gtk toolkit uses this model.

- 1 A compiled mainline code that invokes interpreter to handle UI.

The application is a hybrid of compiled and interpreted code in which the main control loop is compiled. The interpreter acquires user input and displays results.

This architecture is commonly used by compute intensive application like electronic design timing simulators.

- 1 An interpreted mainline that invokes library routines for UI/analysis.

The main program logic is written in an interpreted language and compute intensive portions are performed by compiled functions that are linked to the interpreter. This paradigm is common for extensible interpreters like Tcl, Python, Perl or Visual Basic.

- 1 A glue mainline/UI that invokes separate tasks for analysis.

The main program is interpreted and special purpose functionality is performed by separate compiled executables. This paradigm is common for shell languages like Tcl, sh, perl and python.

The decision of what architecture to use can be tightly coupled with the choice of tools. Very few graphics libraries support all of these architectures. VC++ and wxWindows support the object oriented approaches, but not others. GUI libraries like the Xt, and XVT support styles with a C/C++ mainline, but not scripted or hybrid architectures, while the Qt toolkit supports C/C++ mainline and scripted applications, but not hybrids.

Of the various tools available, Tcl/Tk is the most versatile. The Tk graphics package has been ported to more platforms than other toolkits, and the extensible/embeddable interpreter library supports all of the architectures.

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

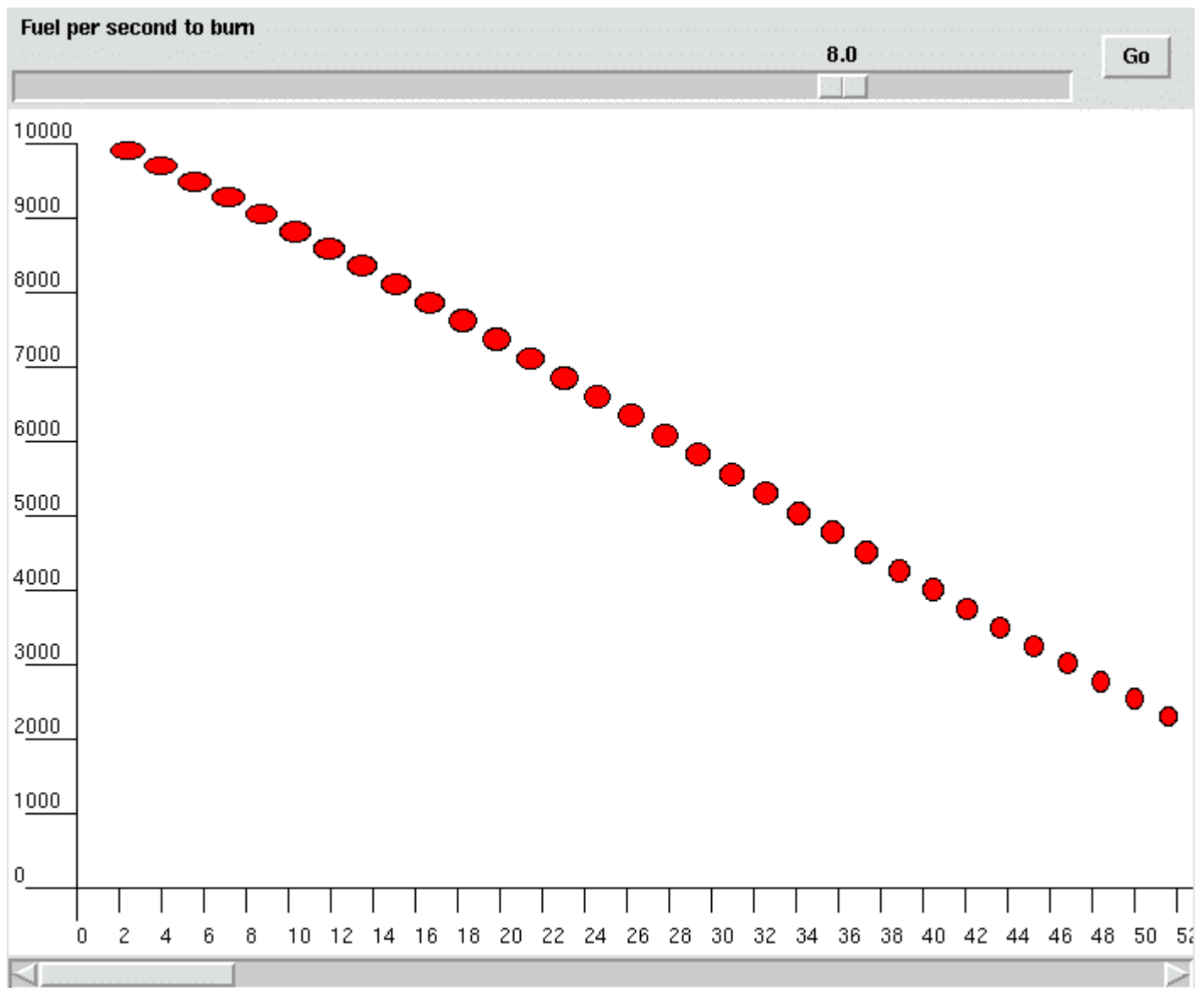
```
set initial conditions

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

The user defines the fuel burn rate using the Tk scale widget, and initiates the simulation by clicking the Go button.

The traditional output for an application like this 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.

A graphic display can convey information more easily. 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.



The application looks like this after a user has completed the simulation

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 handhelds like the Palm Pilot and Windows/CE machines.

The downside is that while Tcl has been optimized with a byte-code engine and other performance enhancements, it's still interpreted language, and a pure Tcl solution may not be fast enough for compute heavy 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://www.equi4.com/starkit>) developed by Jean-Claude Wippler and Steve Landers.

Writing an application in pure Tcl/Tk is a fast way to develop GUI oriented programs. Depending on the complexity of the application, you may write a fully functional version of the application, or stub out some analysis functions with dummy procedures that return precalculated values.

Introduction to Tcl

The Tcl/Tk package is divided into two sections. Tcl is the core interpreter which supports variables, control flow, procedures, namespaces and libraries. Tk is the graphics toolkit which supports many widgets including buttons, labels, scrollbars, and drawing surfaces.

The Tcl syntax is very simple:

The first word of a command line is a command name.

The words in a command line are separated by one or more spaces.

Words can be grouped with double quotes or curly braces.

Commands are terminated with a newline or semi-colon.

A word starting with a dollar sign (\$) must be a variable name. This string will be replaced by the value of the variable.

Words enclosed within square brackets must be a legal Tcl command. This string will be replaced by the results of evaluating the command.

The Tcl/Tk language is very rich and full featured, but for this application we only need a few of the commands.

The `set` command assigns a value to a variable.

Syntax: `set varName ?value?`

varName The name of the variable to define.

value The data to assign to the variable.

```
set txt "Hello, World"
set counter 1
```

Math operations are performed with the `expr` command.

Syntax: `expr mathExpression`

Tcl supports all the math operations in the standard math library, including trig and log functions.

```
set speed [expr {$speed + $acceleration}]
set counter [expr {$counter + 1}]
```

As a shorthand for simple math operations, Tcl includes an `incr` command that is the equivalent to the "C" language `++` and `+=` commands.

```
# counter <= counter + 1
incr counter

# height <= height + velocity
incr height $velocity
```

Tcl supports several looping constructs, the numeric `for` loop, a list-oriented `foreach` loop, and a `while` loop.

Syntax: `while test body`

`while` Loop until a condition becomes false

`test` A statement that tests an end condition.

This statement must be in a format acceptable to `expr`.

`body` The body of code to evaluate on each pass through the loop

```
while {$height < 0} {  
    # height  <= height + velocity  
    incr height $velocity  
}
```

Tcl procedures are defined with the `proc` command.

Syntax: `proc name args body`

`name` The name of the procedure being defined.
`args` The argument list for this procedure.
`body` The body of the procedure being defined.

```
#####
# proc calcSpeed {speed mass burn impulse}--
# Return the speed after rocket burn
# Equation used is:
#  $\Delta v = g_0 * [\Delta t - \text{impulse} * \ln(m_0/m_1)]$ 
# g_0:      gravitational acceleration
# delta_t:  elapsed time (Hardcoded to 1 second)
# 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 pre fuel unit.
#
# Results
# No side effects
#
proc calcSpeed {speed mass burn impulse} {
    return [expr {$speed + 9.8 * (1 - $impulse * log($mass/($mass-$burn))}]
}

```

With these commands, we can write the main simulation loop like this:

```
proc doSimulation {} {
    global burn

    # Define initial conditions
    set ht 10000.0
    set speed 100.0
    set fuel 10000.0
    set gross 100.0
    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
    }
}

```

The next step is the GUI to read the burn rate and report results.

Tk widgets are constructed using this pattern:

widgetType windowPath -option value

The *widgetType* will be the type of widget being created, a canvas, button, label, etc.

The *windowPath* is the full name for this widget in X11 format. This is a tree structured description of the window and its parents using a dot as the separator, and a single dot as the topmost window. Thus, a button on the main window might have a name like `.button1`, or `.exit`, while a button that is part of a frame might have a name like `.controlFrame.update`.

The options and values for widgets vary depending on the behavior of the widget, but always includes the standard options for foreground color, background color, height, width, font and other options commonly included in a graphic context.

Compared to other widget kits like gtk, Qt and wxWindows, Tk is a higher level of abstraction with a lower level of implementation. The high abstraction level insulates the application developer from the underlying windowing system, while the low level of implementation keeps the Tk kernel relatively small.

Tk's higher level of makes it easier to write platform independent code, since there is no temptation (or hooks) to use a feature that only exists on a given platform. This abstraction layer also makes it easier for the maintainers to port Tk to other platforms, since only the functionality of different platforms must be duplicated, without the need to duplicate one framework for defining graphic objects in another.

The base Tk toolkit provides 15 primitive widgets but does not include things like html browser widgets with support for Javascript. While not supporting the complexity of an HTML widget, the `canvas` and `text` widgets are very high level abstractions with many unique features like the ability to bind actions to elements (as small as a single letter or pixel or as large as the entire display), and embedding graphic objects like raster graphics or other Tk windows.

The Tk toolkit does include simple compound widgets like dialog boxes and file selectors. Tk has been extended with hundreds of add-on widgets including the html widget developed by D. Richard Hipps, the BWidget Toolkit maintained at SourceForge as part of the Tcllib project, and the graph and bar chart tools provided by the BLT extension.

For the sake of simplicity, this set of examples uses only basic Tk widgets.

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.

The scale widget

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.

The widget creation commands like `scale` and `canvas` create a widget, but do not display it. The display is handled by the geometry mapping commands `place`, `pack` and `grid`.

The `place` and `pack` commands are useful for asymmetric GUIs, but it can take some work to get the exact display you want. For many applications (like this example) the `grid` command is the best choice.

The `grid` command treats the main window like a spreadsheet and places widgets in the requested row and column, expanding the rows and columns as necessary to hold the largest widget that needs to be displayed.

Syntax: `grid widgetName ?option value?`

widgetName

The widget to be displayed.

?option value?

The `grid` has many options. This is a minimal set:

`-column columnNumber`

The column position for this widget

`-row rowNumber`

The row for this widget.

`-columnspan columnsToUse`

How many columns to use for this widget. Defaults to 1.

`-sticky side`

Which edge of the cell this widget should 'stick' to. Values may be `n`, `s`, `e`, `w`, `ns`, `ew`, or `nsew`.

Once a user has set the burn value, they need to inform the application that they are ready to run the simulation. Or, in the words of Professor Fate "Press the button, Max".

The Tk `button` command creates a button and binds a script to the button. The script will be evaluated whenever a user clicks on the button.

Syntax: `button buttonName ?option value? ...`

buttonName

The name to for this button.

?options?

Valid options for `button` include:

`-command script`

A script to evaluate when the button is clicked.

`-text displayText`

The text that will appear in this button. A newline character (`\n`) can be embedded in this text to create multi-line buttons.

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 Tk `canvas` widget is the generic drawing surface. This is a vector oriented drawing surface in which each item displayed is a separate graphic entity. Each entity has its own state (size, location, etc) which can be modified by the controlling application. Each entity 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 be 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.

Whenever Tk creates a graphic widget, it also creates a new command to use to interact with that widget. This new command is used to do things like modify a widget's configuration options or create new objects on a canvas screen. The canvas can hold many types of objects including graphic primitives like lines, ovals, text and rectangles, as well as more complex objects like raster images and other Tk windows.

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

```
canvasName create objectType coordinates ?option value?
```

This code snippet 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
}
```

A canvas is actually a window into a much larger area. If a script attaches the canvas to a scrollbar, the user can pan around and view sections of the canvas that are not initially visible.

Tcl/Tk handles event-driven style programming 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.

Syntax: `scrollbar scrollbarName ?options?`

`scrollbar`

scrollbarName

The name for this scrollbar

options

This widget supports several options. The `-command` option is required.

`-command "procName ?args?"`

This defines the command to invoke when the state of the scrollbar changes. Arguments that define the changed state will be appended to the arguments defined in this option.

`-orient direction`

Defines the orientation for the scrollbar. The *direction* may horizontal or vertical. Defaults to vertical.

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 script bound to them, to be evaluated when an event occurs while the pointer is 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)}]
    set x2 [expr {$x + 30}]

    # 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
# next to the oval.

.c bind $item <Button-1> \
  "showDetails $x2 $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 uses the `format` command (similar to `sprintf`) to format the display values, and 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 \
    -anchor w -text \
    "Speed: $s Height: $h Fuel: $fuel" \
}

```

Pure C solution

A non-GUI C solution would look like this. It would accept the burn amount on the command line, and would print out each

```

#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:
//  $\Delta v = g_0 * [\Delta t - \text{impulse} * \ln(m_0/m_1)]$ 
// g_0:      gravitational acceleration
// delta_t:  elapsed time (Hardcoded as 1 second)
// 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 pre 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;
    }
}

```

The program output resembles this:

```
%> lander 3.0
0> speed: 100.00 fuel: 1000.00 height: 10000.00
1> speed: 106.70 fuel: 997.00 height: 9893.30
2> speed: 113.40 fuel: 994.00 height: 9779.90
3> speed: 120.09 fuel: 991.00 height: 9659.80
4> speed: 126.78 fuel: 988.00 height: 9533.02
5> speed: 133.46 fuel: 985.00 height: 9399.56
6> speed: 140.14 fuel: 982.00 height: 9259.41
...
```

Tcl as Glue

Like other shell type languages such as perl and bash, a Tcl script can fork/exec an executable as a child task. The exec command will execute another program. It returns whatever output the child task would have sent to the standard output device.

Note that the exec command is not supported under a Classic Mac OS, where a textual standard output device doesn't exist.

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 defined character (newline) occurs and the scan command, which behaves similar to the "C" sscanf command.

```
proc doSimulation {} {
    global burn

    # Run the child app, collect output
    set return [exec lander $burn]

    # Split into lines
    foreach l [split $return \n] {

        # Scan line for values
        scan $l \
            "%d> speed: %f fuel: %f height: %f" \
            time speed fuel ht

        # Display results
        showState $time $speed $fuel $ht
    }
}
```

Tcl Mainline with "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 linking sets of C code into the Tcl interpreter. Either writing an extension by hand, using the automated extension generator SWIG (www.swig.org), or with 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 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. 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::libraries`

Define libraries necessary for the link phase.

`critcl::ccode`

Define ancilliary code to be placed within the C source file. This lets you declare include files, and preprocessor macros.

The `critcl::libraries` 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 `typeint`, `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.

Most of the Tcl API commands come in two flavors, original and object. Initially, the internal format for Tcl variable data was a string. 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 of Tcl several years ago, 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.

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 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.

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

`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 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, an 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.

The `Tcl_FindExecutable()` function initializes the Tcl variables that describe the full path for the executable (script or embedded application). These are used internally by the Tcl interpreter to find and load required libraries, and will be returned by the `info nameofexecutable` command.

```
#include <tcl.h>
```

```

...
main(int argc, char *argv[]) {
    Tcl_Interp *interp;
    ...
    Tcl_FindExecutable(argv[0]);
    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 open for business, and ready to accept one or more script to evaluate. The `Tcl_Eval` function is one of a family of functions that pass a script to the interpreter. The script can be as simple as a single command, or arbitrarily long.

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.

These lines 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");
Tcl_Eval(interp, "inputInitialValues");

```

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

In applications where the program logic is controlled within the Tcl script, the application starts the script and then passes control to the event loop.

This example is architected 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 enters the event loop and processes 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: `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 bitmapped 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`.

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

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

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

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.

These functions return `TCL_OK` on success, or `TCL_ERROR` if the conversion fails (perhaps the string representation is not numeric).

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

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

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, &ready);
}
```

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, &burn)) {
    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 retrieved from the interpreter with the `Tcl_GetVar(interp, "errorInfo", TCL_GLOBAL_ONLY);` function call.

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.

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.

Summary

The Tcl interpreter provides a versatile platform for developing multi-platform applications using a variety of architectures.

The GUI and output display can be created either in scripts that are maintained externally to the application, or within the application mainline code.

The multi platform Tk package is the best integrated GUI package for Tcl. However, 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/](http://www.dr-baum.net/gnocl/)), XVT, and others.

Bibliography

[Price] Nat Price, Patterns for Scripted Applications
<http://www.doc.ic.ac.uk/~np2/patterns/scripting/index.html>

[Ferrieux98] Alexandre Ferrieux, Concepts of Architectural Design for Tcl Applications,
<http://mini.net/tcl/297>, 1998

Common GUI procedures

```
#####
# 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 Solution

```
source GUI.tcl
```

```

#####
# proc calcSpeed {speed mass burn impulse}--
# Return the speed after rocket burn
# Equation used is:
#  $\Delta v = g_0 * [\Delta t - \text{impulse} * \ln(m_0/m_1)]$ 
# 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 pre 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” Solution

```

#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:
//  $\Delta v = g_0 * [\Delta t - \text{impulse} * \ln(m_0/m_1)]$ 
// 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 pre 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 and "C" using Critcl

```
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” Code with embedded Tcl Interpreter

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

```

// 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, &ready);

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, &ready);
}

// 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, &burn)) {
    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);  
  }  
}
```