Tcl/Tk Extensions for Visualization of Large Data Sets

Neil D. McKay
Enterprise Systems Lab
General Motors Research
30,500 Mound Road
Warren, MI 48090-9055

Abstract

This paper describes some extensions to Tcl and Tk that were used to implement tools for the analysis and visualization of large data sets. Such tasks require efficient manipulation of large numerical arrays, and convenient display of three-dimensional data. These goals were realized by providing Tcl and Tk with five interdependent extensions, which provide:

- tensor objects, for the efficient storage and manipulation of large numerical arrays.
- faceted objects, for implementing display lists
- the `view3d` widget, an extensible 3D viewing widget with OpenGL interpreter
- the ability to render faceted objects in view3d widgets
- commands for converting between tensors and Tk photo images.

These tools were used as part of a toolkit for the analysis of dimensional data in the automobile industry.

1 Introduction

General Motors' vehicle body assembly process is monitored for dimensional accuracy in a variety of ways. During production, every body is measured by optical measurement equipment. In addition, individual bodies occasionally are taken off the assembly line and measured with coordinate measuring machines and/or custom gauges; during plant startup, prior to the beginning of production, the assembly line's entire output may be measured this way. These processes produce numerical data that is analyzed statistically, and the results of the analysis are used to determine when problems are occurring, the nature of those problems, and how to fix them.

The work described here was part of an effort to produce a set of tools for the gathering, storage, statistical analysis, and visualization of assembly line measurement data. This effort, called CDIS, the Common Dimensional Information System, provides tools that can be used by both assembly plant personnel and GM's central manufacturing groups. The author's responsibility was to create an analysis tool that was geared towards people who set up the assembly lines and who troubleshoot the assembly equipment when plant personnel cannot diagnose the problem; these people use a wide variety of computer equipment to manipulate and analyse both small and very large data sets, and need good data visualization tools to guide their analyses. Thus there were three important considerations in creating the foundation on which to build the CDIS analysis tool: portability of the code to multiple platforms, the ability to handle very large data sets, and the production of good 2D and 3D graphics. Tcl/Tk meets the portability requirements very well, since it runs on Windows 95/98/NT and a variety of Unix Workstations. However, the standard Tcl/Tk distribution does not provide adequate data-handling and graphics capabilities; this is where the Tcl `package` and `load` mechanisms came to the rescue.

The Tcl/Tk extensions described in this paper consist of five loadable packages that allow Tk applications to manipulate large numerical arrays and generate 3D graphic displays. The `Tns` (tensor) package is a numerical array extension. This package provides facilities for efficiently manipulating multidimensional arrays of numbers within Tcl. (It is similar in spirit to the BLT package's vector facility.) The `Vtd` package provides a Tk widget, called a view3d widget, in which to display 3D graphic images. This widget's functionality can be extended by adding "renderers", which are programs, written in C, for drawing objects in a view3d widget. The `Fct` (faceted object) package implements a kind of faceted surface model. Faceted objects contain a list of three-dimensional vertices, and lists of lines, triangles, and quads. The line/triangle/quad lists indicate how the vertices are connected together to form surfaces. Although the `Fct` package is intended to provide display lists for the view3d widget, it is written as display-independent code; it can, in fact, be run from `tclsh` without Tk, so that display objects can be built and manipulated offline. Display of faceted objects is performed by the `Fctr` (faceted object rendering) package, which extends the view3d widget with a faceted object renderer. Finally, the `Tnsph` (tensor-to-photo) package facilitates transfer of data between tensors (as provided by the Tns package) and Tk's photo image objects. This allows the user to transfer a native Tk image into a tensor, perform image processing operations, and convert the processed image back to a Tk image.

The remainder of this paper is divided into 6 sections. Sections 2 through 6 provide an overview of the capabilities of each of the five Tk3D extensions; the paper concludes with section 7.

2 Tns: The Tensor package

The tensor package provides Tcl with an efficient way to store and manipulate numerical arrays. The `Tns` package adds two commands to Tcl: `tensor` and `is_tensor`. The `is_tensor` command takes a single argument, and returns 1 or 0, depending on whether the argument is or is not the name of an existing tensor. The `tensor` command creates tensors; its behavior is described below.

2.1 Creating a tensor

The `tensor` command creates a new tensor object. For instance, the command

```
tensor I33 -initial {{1 0 0} {0 1 0} {0 0 1}}
```

will create a tensor named `I33` that contains a 3×3 identity matrix. It also creates the Tcl command `I33`; this is the tensor's *object command*. The tensor `I33` may be accessed and manipulated via the `I33` command, as will be discussed later.

There are several other options that can be used with the `tensor` command. Every `tensor` command must have exactly one of the `-size`, `-initial`, or `-initfile` options; these options determine the tensor's size. The `-size` option must be followed by a list of positive integers that specify the tensor's dimensions. The `-initfile` option loads the contents of a tensor from a file. Finally, there is the `-type` option. By default, tensors will store data as double-precision floating point numbers. The `-type` option will override this, allowing other data types. In addition to `double`, allowed types are `float`, `int`, `short`, `byte`, and `unsigned`.

2.2 Inquiry Functions

Typing a tensor's object command with no arguments will return the contents of the tensor; for instance, typing the `I33` command with no arguments will print out the contents of the `I33` tensor, namely the list

```
{1 0 0} {0 1 0} {0 0 1}
```

It is also possible to print out portions of a tensor, using the `section` directive. For example, the statement

```
I33 section {0 *}
```

will print out the first row of `I33`.

Other arguments to the tensor's object command will return other information about the tensor. In particular, a tensor's type, order (i.e. the number of indices), and dimensions can be obtained via tensor object commands.

2.3 Tensor arithmetic

The most basic arithmetic operation on tensors is assignment. Not surprisingly, assignment is achieved using the "=" command. This gives a fairly natural-looking assignment syntax. Suppose we create two tensors:

```
tensor p34 -size {3 4}; tensor q34 -size {3 4}
```

Setting all elements of a tensor to a single scalar value (in this case, the value 1.0) may be accomplished by the statement

```
q34 = scalar 1.0
```

To place the contents of tensor `q34` into tensor `p34`, we would write

```
p34 = tensor q34
```

It is also possible to add the contents of one tensor to another via the "+=" command. The statement

```
p34 += tensor q34
```

will add the contents of `q34` to `p34`. The "-=" command behaves similarly, except that it will subtract instead of add. The commands "*=" and "/=" multiply and divide, respectively.

Tensor assignment commands may be applied to parts of a tensor via `section` directives. For example, we may assign a value to a single element of `p34` with the command

```
p34 section {0 1} = scalar -1.5
```

Likewise, we could assign a value to a whole column:

```
p34 section {* 3} = scalar -10
```

A `section` directive may also appear in a `tensor` operand, allowing assignment of part of a tensor to part of another tensor:

```
p34 section {0 *} = tensor q34 section {1 *}
```

Several other operand types are also available; these include `multensor` and `outer` for computing tensor products, `file` for reading an operand from a file, and `contraction` for computing tensor contractions. (Contractions are a generalization of the matrix trace operation.)

The tensor package also provides comparison operators, which are similar to the assignment operators except that instead of changing the tensor's contents they return a Boolean value indicating whether the specified condition is met. For instance, suppose we set

```
v3 = array {0 1 2}; v4 = array {0 0 0 0}
```

Then the operations

```
v3 == scalar 0
v4 == scalar 0
```

will return 0 and 1, respectively. The == operator will return 1 if *all* elements of the given tensor are equal to the given operand, 0 otherwise. Other comparison operations work in a similar fashion. The operand of a comparison operator may be any value that is allowed by the assignment operators; in particular, tensor-to-tensor comparisons are possible.

2.4 Miscellaneous Tensor Commands

Tensors have many other commands that won't be discussed here due to space constraints. These include the `put` command, which copies a tensor's contents to a standard Tcl array, the `write` command, which writes the contents of a tensor to a file, and the `minmax` and `listindices` commands, which find tensor elements having specified properties. There are also commands for rearranging a tensor's contents, e.g. exchanging, inserting, and deleting rows and columns.

3 Vtd: a 3D Viewing Widget

The `Vtd` (view3d) package provides a Tk widget in which to draw 3D scenes. It provides access to virtually all the facilities of the OpenGL 3D graphics library[1]. In this section, we will describe the most important features of the view3d widget.

3.1 Creating a view3d widget

Creating a view3d widget is much like creating any other widget; it is done via the `view3d` command, and creates a Tk widget command. The `view3d` command accepts a few standard Tk options, such as `-width`, `-height` and `-cursor`. The `view3d` command also has many widget-specific options for specifying properties that only make sense for view3d widgets. These include `-doublebuffer`, which determines whether graphics will be single- or double-buffered, the `-rgb` option, which specifies whether the view3d widget will use an RGB color scheme or colormapped graphics, and the `-zplanes` option, which specifies the number of bits used for the z-buffer (or depth-buffer).

View3d widgets are destroyed in the usual manner: either with the Tk `destroy` command or by renaming the widget command to the empty string.

3.2 The view3d programming model

Every view3d widget of course has a window associated with it. However, it will also have some other associated structures and properties. Every view3d widget has its own model and view matrices. These are used when rendering objects with the view3d widget's `render` command, and may be manipulated via the `model`, `model_offset`, and `view` widget commands. The `model` and `view` commands may use Euler angles or roll, pitch, and yaw angles to set the rotation matrices in the corresponding transforms.

There are also some tables or "maps" associated with each view3d widget: a color map, and front and back material maps. The color map is, as its name implies, an array containing color information. In the case of colorindex (colormapped) windows, this is just the window's own colormap. The view3d widget's `colormap` command provides a way of reading, writing, and modifying the colormap. For RGB windows, the color map is really just a convenience; it can be accessed by object renderers for various purposes, e.g. displaying different classes of polygons in different colors. The front and back material maps behave like the color map, except they store "material" information: material color, diffuse and specular reflectivity, and several other properties. These maps, like color maps, are available for use by object renderers.

3.3 Interpreted OpenGL commands

Most of the commands implemented by view3d widgets are handled by an OpenGL interpreter. The interpreter maps view3d widget commands onto equivalent OpenGL library calls on a one-to-one basis; the first argument after the view3d widget's name specifies the OpenGL function to be called, and subsequent arguments are translated into arguments for the OpenGL function. The procedure for converting OpenGL function calls to their view3d widget equivalents is straightforward: strip off the "gl" prefix, strip the OpenGL suffixes that indicate the number and types of arguments, and translate upper case to lower case. For instance, the OpenGL call `glColor3f` becomes just `color`. Likewise, `glBegin` becomes `begin`, and `glEnd` becomes `end`. The commands for specifying polygon vertices, e.g. `glVertex3i`, `glVertex4f`, etc., become the single command `vertex`. There is also a (slightly more complicated) set of rules for translating OpenGL argument lists to Tcl arguments; these rules won't be described here, but they generally result in statements that are easily read by anyone who is familiar with OpenGL. For instance, the commands

```
.v clearcolor red
.v clear {color depth}
```

set the window's clear color to red, and clear both the window itself and the window's depth buffer. Likewise, the code

```
tensor p0 -size 3 -type float
p0 = array {-0.5 -0.5 0}
.v begin line_strip
.v vertex p0
.v vertex {0.5 -0.5 0}
.v vertex {0.5 0.5}
.v vertex {-0.5 0.5}
.v end
```

will draw a polyline with the given vertices. (Note that the `vertex` command can accept coordinates as either a Tcl list or as a tensor name.)

3.4 Object Rendering Commands

The `render` and `render_props` view3d widget commands provide hooks for calling custom C code that draws in a view3d widget. In general, the `render` and `render_props` commands must be followed by the name of a renderer. (The name is set by the renderer's C code.) All following arguments are passed on to the rendering code itself (in the case of `render`) or to the renderer's property-setting code (in the case of `render_props`). How the arguments are parsed is entirely up to the rendering code.

The view3d code comes with one object renderer pre-installed: the `gl_graphics` renderer. The `render_props` command for this renderer doesn't do anything at all; the `render gl_graphics` command accepts a single argument, namely a Tcl list of interpreted OpenGL instructions. The previous polyline example could be written as follows:

```
.v render gl_graphics {
    {begin line_strip}
        {vertex p0}
        {vertex {0.5 -0.5 0}}
        {vertex {0.5 0.5}}
        {vertex {-0.5 0.5}}
```

```
        {end}
    }
```

The elements of the list are interpreted by the widget's OpenGL interpreter, almost as if individual widget commands had been executed. Although the `gl_graphics` renderer appears to be redundant, it has some useful properties. First, it is efficient; the `gl_graphics` code will be executed as a single Tcl command, so there's less execution overhead. The view3d widget also only needs to set the OpenGL graphics context once to execute multiple OpenGL commands. The second useful property of the `gl_graphics` rendering command is that the `render gl_graphics` command will set the widget's perspective transformations using the widget's model and view matrices. This means that all drawing commands executed in the object renderer can be performed in a fixed coordinate system, without concern for where the graphics appear in the rendered scene, or where the scene will be viewed from.

3.5 Event bindings

The OpenGL library is a "render-only" graphics library. This means that it only contains functions for drawing pictures; input, e.g. handling mouse clicks or other interaction, is left up to the operating system that OpenGL is running under. Fortunately, Tk abstracts much of this interaction in an operating-system-independent way. Thus interaction in OpenGL can be handled using standard Tk commands, such as `bind`. For example, when another window overlaps a view3d widget, and then is moved or destroyed, the view3d widget's contents must be redrawn. If the procedure `redraw` redraws the contents of view3d widget `.v`, then to handle exposure events we need only add the code

```
bind .v <Expose> {redraw}
```

so that the window will always be redrawn when an `Expose` event occurs. Other familiar Tk programming techniques can be used for handling keyboard input and mouse-related events.

4 Fct: Faceted Objects

The `Fct` (faceted object) package provides Tcl with a way of creating and manipulating faceted surface models, i.e. sets of points, lines, triangles, and quads. This is accomplished by adding two new Tcl commands: `facetobj` and `is_facetobj`. These commands are similar to the `tensor` and `is_tensor` commands; `facetobj` creates a faceted object, which is accessed via an associated command, and `is_facetobj` returns a Boolean value indicating whether its argument is or is not the name of a faceted object. The rest of this section discusses the creation, destruction, and manipulation of faceted objects.

4.1 Creating and destroying faceted objects

Faceted objects are created with the `facetobj` command. For instance, the command

```
facetobj f
```

creates a faceted object named `f`. The `facetobj` command above also creates an object command `f`, through which the faceted object can be manipulated. A faceted object can be destroyed by renaming its object command to the empty string.

4.2 The faceted object's programming model

In essence, a faceted object can be thought of as a list of vertices, and some lists of polygons built using those vertices. The vertices themselves are three-dimensional points, represented using homogeneous coordinates. For instance, suppose we create a faceted object called `f1`, and put some points in its vertex list:

```
facetobj f1
f1 append points {{-1 -1 0} {1 -1 0} {1 1 0} {-1 1 0}}
```

The vertex list of `f1` would look like this:

Index	X	Y	Z	W
0	-1	-1	0	1
1	1	-1	0	1
2	1	1	0	1
3	-1	1	0	1

Note that the fourth coordinate, W, has the value 1.

Polygons (or *point tuples*) are defined in terms of the indices of the vertex list. The command

```
f1 append triangles {{0 1 2} {2 1 3}}
```

creates a pair of triangles; the first triangle has the vertices $(-1,-1,0)$, $(1,-1,0)$, and $(1,1,0)$, while the second triangle has vertices $(1,1,0)$, $(1,-1,0)$, and $(-1,1,0)$. Schematically, the triangle list looks like:

Index	V1	V2	V3
0	0	1	2
1	2	1	3

Of course there are other point tuple lists as well, namely `lines`, which have two vertices per tuple, `quads` which have four vertices, and `pointrefs` (point references), which have only one vertex.

Although having vertex and point tuple lists is useful for representing polyhedral objects, their display often requires that other information be associated with individual vertices and polygons. To this end, each vertex and each point tuple have three associated quantities: a flag word, a label, and a normal vector. The flag word is a small integer (generally an unsigned byte) that can be used arbitrarily by the programmer. The label is an arbitrary text string associated with the point or tuple; this could be used, for example, to classify triangles or quads as belonging to a particular surface patch. There is one normal vector for each point or point tuple; these may be used arbitrarily, but their intended purpose is for holding surface normal information.

4.3 Querying vertex and point tuple lists

There are a number of ways to obtain the contents of the vertex and point tuple lists. The simplest is to use the `get` command. For the faceted object `f1` defined above, the command

```
f1 get points
```

will return the point list, in the form of a Tcl list of four-dimensional vectors. For the vertex list currently stored in `f1`, the value returned would be

```
{-1 -1 0 1} {1 -1 0 1} {1 1 0 1} {-1 1 0 1}
```

We may obtain vertex lists in the same manner; the command

```
f1 get triangles
```

returns

```
{0 1 2} {2 1 3}
```

It is also possible to copy the contents of the vertex and tuple lists into tensors using the `put` command, e.g. the command

```
f1 put points into v
```

will create a 4×4 floating-point tensor named `v`, containing `f1`'s vertices. The `number` command may be used to obtain the number of items in the vertex and tuple lists.

4.4 Modifying the vertex and point tuple lists

As previously shown, it is possible to append new points onto a faceted object's vertex list with the `append points` command; `append` also works for point tuple lists. The commands

```
f1 append points {{2 -1 -0.5} {2 1 -0.5}}
f1 append triangles {{1 4 5} {1 5 2}}
```

will append two more points to the vertex list and two more triangles to the triangle list. It is also possible to replace and delete vertices and tuples. The command

```
f1 replace points starting at 0 with {{-1.1 -1.1 0} {1.1 -1 0.1}}
```

will replace the first two points in the vertex list with $(-1.1, -1.1, 0)$ and $(1.1, -1, 0.1)$. Point tuples and vertices may be deleted with the `delete` command, e.g.

```
f1 delete triangles range {1 2}
```

will delete triangles 1 and 2. Note that the list will be renumbered; what used to be triangle 3 becomes triangle 1, and old triangles 1 and 2 disappear.

Deleting vertices is complicated by the fact that the deleted points may be used to define lines, triangles, or quads. Faceted objects keep their vertex and point tuple lists consistent by deleting any point tuples that reference the deleted points.

So far we have discussed relatively simple forms of the commands for modifying the point and tuple lists; however, many of these commands may have conditional clauses attached to them. For instance, we may delete vertices that are not referenced by any point tuples with the command

```
f1 delete points where ref_count == 0
```

Of the commands discussed here, the `delete`, `number`, `get`, and `put` commands allow a conditional clause. Many other conditions are allowed in conditional clauses; a few of these will be discussed in the next section.

4.5 Setting and testing flags

Conditional clauses are most often used to test and manipulate the lists of flags associated with vertices and point tuples. To get the flags associated with a faceted object, use the `get` command:

```
f1 get point flags
```

Flags usually are set using a variant of the `set` command. For example:

```
f1 set point flags 0x01 to 0x01 if ref_count == 1
```

will set the low-order bit of the flags word for every vertex that is referenced by exactly one point tuple.

The conditional clause in a `set flags` can take on many other forms. The most powerful conditions involve spatial relationships. For instance, it is possible to set flags for all entities that lie inside a given box or other convex set. The command

```
f1 set point flags 0x01 to 0x01 \
        if inside box {{-1.5 1.5} {-1.5 1.5} {-1.5 1.5}}
```

will set the point flags to 1 for every point that is inside a box that is 3 units on a side, centered on the origin. A large variety of other tests are also supported, including tests based on distances between polyhedra, and tests based on flag settings and label patterns.

4.6 Miscellaneous commands

Faceted objects support a wide variety of other commands. These commands can merge faceted objects, compute and manipulate normal vectors, read and write faceted objects to and from external storage, generate outlines of polygons, search for points and polygons that have specified properties, and substitute one point for another.

5 Fctr: Viewing Faceted Objects

The `Fctr` (faceted object rendering) package provides a means for displaying faceted objects, provided by the `Fct` package, in view3d widgets provided by the `Vtd` package. Unlike the other packages discussed so far, the `Fctr` package does not add any new commands to Tcl/Tk; it adds a new renderer, the `facetobj` renderer, to the `view3d` widget.

5.1 Rendering a faceted object

Rendering of faceted objects is best illustrated by an example. Suppose we want to render a faceted object called `cube`, containing eight vertices and six quads, in a view3d widget named `.v`. First, we need to set up a projection matrix. The commands

```
.v matrixmode projection; .v loadidentity
.v ortho -4 4 -4 4 -4 4
```

will set up orthographic projection. The commands

```
.v matrixmode modelview; .v loadidentity
```

set the modelview matrix. We also need to enable depth buffering and clear the window:

```
.v enable depth_test; .v clear {color depth}
```

With this setup done, we may now render the cube. However, we must tell the faceted object renderer what parts of the faceted object to display. The command

```
.v render_props facetobj render quads
```

tells the renderer to display only quads when rendering faceted objects. Now the single command

```
.v render facetobj cube
```

will display the cube; with the default rendering properties and the default initial position and orientation, this will look like a white square in the middle of a black background. The `view3d` widget's `model` and `view` commands change the position and orientation of the cube; other aspects of its appearance are controlled by supplying other arguments to the `view3d` widget's `render_props` command.

5.2 Setting rendering properties

The simple code above shows one characteristic that can be controlled by the `render_props` command: what vertices or point tuples to display. However, there are many other display properties that are controlled via the `render_props facetobj` command. For example, suppose we set `cube`'s quad flags to {`0 1 2 3 4 5`} and set the view3d widget's colormap as follows:

```
.v colormap set entries {0 5} to map {red orange yellow green blue purple}
```

Then the code

```
.v render_props facetobj for quads set color from flags
.v clear {color depth}; .v render facetobj cube
```

will draw the cube with a different color on each face.

The color may also be taken from the point flags. This is useful if you want to do smooth (Gouraud) shading. The point flags may be set with the command

```
cube replace point flags starting at 0 with {0 1 2 3 1 2 3 4}
```

The command `.v shademodel smooth` will turn on smooth shading, and the commands

```
.v render_props facetobj for quads set color from point_flags
.v clear {color depth}; .v render facetobj cube
```

will draw the cube with smooth-shaded faces. The color will be set at each vertex, and the faces will be drawn with color that varies smoothly from vertex to vertex.

It is also possible to display faceted objects in more sophisticated ways; for example, the faces of the cube can be material-mapped and lighted, or they may be texture mapped.

6 Tnsph: Tensor to Photo Image Conversion

The `Tnsph` package is a Tk extension for converting between tensors and Tk photo images. It does this by providing two commands: `tensor2photo` and `photo2tensor`. These commands are described below.

6.1 Converting between photo images and tensors

The `photo2tensor` command converts Tk photo images to tensors. For instance, if we have a 256×256 color photo image named `img1`, we may convert this image to a tensor `img1Tensor` with the command

```
photo2tensor img1 img1Tensor
```

This will create a tensor named `img1Tensor`, of type `byte` and dimension $256 \times 256 \times 3$. (The "3" indicates that it is a color image, having red, green, and blue components.)

The image data in `img1Tensor` may be manipulated using any tensor command. For instance, we could eliminate the green components by setting them to zero:

```
img1Tensor section {* * 1} = scalar 0
```

To display the manipulated image, we first create a new image, e.g. with the command

```
image create photo img2
```

We may copy the processed image contained in the tensor into this new photo image using the `tensor2photo` command:

```
tensor2photo img1Tensor img2
```

This image should look like the original, except it will look purple, since it has only red and blue color components.

7 Conclusions

The Tk3D suite of Tcl packages provides a number of Tcl/Tk extensions that are useful for building 3D-enabled applications and for manipulating large data sets. Although the suite currently is used only in a data analysis tool, its constituent extensions are designed to be generic and flexible, and should be useful for many other purposes as well. In particular, the tensor package would be useful for any application that performs vector and matrix operations, and the `Tnsph` package could serve as the basis for an image-processing tool.

The ability of the `Vtd` package to accomodate rendering code written in C also enhances the package's flexibility. It allows for reasonably efficient 3D graphics while still providing the flexibility of a scripting language, and allows the use of multiple data representations simultaneously.

References

[1] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison-Wesley, Reading, Massachusettes, 1993.