Doing 3D with Tcl

Paul Obermeier
obermeier@poSoft.de

Abstract

This paper presents an approach called **tclogl**, which offers the 3D functionality of OpenGL at the Tcl scripting level. Tclogl is an improved and enhanced OpenGL binding based on the work done with Frustum by Roger E Critchlow.
The paper starts with an overview of existing 3D libraries with a Tcl sripting interface. Different solution approaches are discussed and compared against the given requirements. The choosen implementation, which relies heavily on SWIG, is explained in detail in the main section of this paper. Common pitfalls when programming OpenGL in Tcl, as well as open issues of this approach are shown. Finally the results of a range of test programs and some demonstration applications are shown.

1 Overview

Hardware accelerated 3D capabilities are available nowadays on nearly every PC. There is also a broad range of programming libraries for doing 3D visualization, coming from different application domains, like simulation, gaming, visualization or animation.
These libraries differ in availability on computer architectures and operating systems, complexity and richness of supplied functionality, as well as the supported language bindings.

There are two low-level (light-weight) graphic APIs in common use today: OpenGL and DirectX. While DirectX from Microsoft is available only on machines running the Windows operating system, OpenGL is running on PC's as well as on workstations. OpenGL also has a software-only implementation called "Mesa", so you can run OpenGL based programs even in virtual machines or over a network. OpenGL libraries are part of all major operating systems distributions.
DirectX and OpenGL both offer a C based programming interface.

Based on one of these 2 low-level APIs lots of heavy-weight libraries exist, available as OpenSource implementations as well as commercial versions, adding features like scene-graphs, image handling, animation, advanced lighting models, etc.
Most of these libraries offer a C/C++ language binding, but only a few of them enable the user to "script" a 3D application.

Some examples of 3D libraries offering a Tcl language binding are listed in the following overview. The libraries are divided into the above mentioned categories heavy-weight and light-weight. Only non-commercial libraries are taken into account.
You may also take a look at the OpenGL related Tcl'ers Wiki page ([5]).

Name	Platforms	Source	Reference URL
Nebula	X11/Win/MacX	Yes	http://www.nebuladevice.org
Fltk	X11/Win/MacX	Yes	http://www.fltk.org
VRS	X11/Win	Yes	http://www.vrs3d.org
VTK	X11/Win/MacX	Yes	http://public.kitware.com/VTK
tk3d	X11/Win	Yes	http://www.gm.com/company/careers/career_paths/rnd/lab_manuf_sw.html

Table 1: List of heavy-weight 3D libraries with Tcl binding

Name	Platforms	Source	Reference URL
Glut/Tk	X11/Win	Yes	http://zing.ncsl.nist.gov/gluttk
Tkogl	X11/Win	Yes	http://hct.ece.ubc.ca/research/tkogl/tkogl
togl	X11/Win/MacX	Yes	http://togl.sourceforge.net
Frustum	X11/Win	Yes	http://www.elf.org/pub/frustum01.zip
XBit	Win	No	http://www.geocities.com/~chengye/opengl.html
tom	X11/Win	Yes	http://sourceforge.net/projects/om2t

Table 2: List of light-weight 3D libraries with Tcl binding

The following short excerpts from the libraries' home pages should act as a brief introduction and overview of their capabilities.

Nebula Device is an open source realtime 3D game/visualization engine, written in C++. Version 2 is a modern rendering engine making full use of shaders. It is scriptable through TCL/Tk and Lua, with support for Python, Java, and the full suite of .NET-capable languages pending. It currently supports DirectX 9, with support for OpenGL in the works. It runs on Windows, with ports being done to Linux and Mac OS X.

FLTK (pronounced "fulltick") is a cross-platform C++ GUI toolkit for UNIX®/Linux® (X11), Microsoft® Windows®, and MacOS® X. FLTK provides modern GUI functionality without the bloat and supports 3D graphics via OpenGL® and its built-in GLUT emulation.

The Virtual Rendering System is a computer graphics software library for constructing interactive 3D applications. It provides a large collection of 3D rendering components which facilitate implementing 3D graphics applications and experimenting with 3D graphics and imaging algorithms. VRS is implemented as a C++ library. Applications can incorporate VRS as C++ library based on the C++ API. In addition, we provide a complete Tcl/Tk binding of the C++ API, called iVRS.

The Visualization ToolKit (VTK) is an open source, freely available software system for 3D computer graphics, image processing, and visualization used by thousands of researchers and developers around the world. VTK consists of a C++ class library, and several interpreted interface layers including Tcl/Tk, Java, and Python.

Tk3D is a collection of extensions to Tcl/Tk that allow Tcl/Tk applications to manipulate large numerical arrays and generate 3D graphic displays. The Tk3D suite contains five packages, named Tns, Vtd, Fct, Fctr, and Tnsph. The "Tns" (tensor) package is a numerical array extension. It provides facilities for efficiently manipulating multidimensional arrays of numbers within Tcl. 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.

GLUT/Tk is a "light-weight" system that seeks to leverage GLUT and Tcl/Tk by tying them together in a stylistically consistent way with the addition of only a few commands to each. The basic implementation strategy is to enable a GLUT process to launch an independent Tk script. Thus, the built-in event loops of these two systems can operate as usual and the resulting programming style (registering callbacks for given events) is unchanged.

TkOGL is a package extension to the Tcl scripting language that enables a user to utilize OpenGL, a multi-platform API for interactive 2D and 3D graphics applications. TkOGL makes it possible for the user to display OpenGL graphics on the Tk canvas along with other Tk widgets.

Togl is a Tk widget for OpenGL rendering. Togl allows one to create and manage a special Tk/OpenGL widget with Tcl and render into it with a C program. That is, a typical Togl program will have Tcl code for managing the user interface and a C program for computations and OpenGL rendering.

Frustum implements a specialization of the Togl widget and a Swig generated Tcl binding for the opengl and glu libraries to allow 3d modelling to be done entirely from Tcl.

XBit has implemented a Tcl shell for OpenGL primatives at Windows platforms. The implementation focuses on scriptive programing in OpenGL rendering with an emphasis on code reusability and GUI. It provides an OpenGL rendering engine whose states can be changed with a greater flexibility during execution.

Tom is an OpenGL wrapper for Tcl/Tk. It provides Tcl procs very close to OpenGL C functions.

2 Wish and reality

2.1 Requirements

As has been shown in the previous chapter, a number of 3D libraries with Tcl bindings are currently available. But none of them fulfilled my personal wish list for a Tcl enabled 3D library: It should give me the ability to integrate small- to medium- sized 3D content into my Tcl/Tk based graphical user interfaces.

The prefered candidate should be an OpenGL based light-weight package, because OpenGL is available on nearly every platform. 3D functionality should be scriptable with Tcl commands and it should be possible to extend the functionality with C code. Graphical output should be displayed in a Tk widget.

The following table summarizes the requirements of my favourite Tcl-3D library.

#	2.1.1 Requirement	Comment
1	Light-weight	Small code size, Tcl package.
2	License	Source code availability under BSD license.
3	High automation	No need to write lots of wrapper/glue code. Easy upgrade to newer versions of the 3D library.
4	Portable	Availability on many platforms.
5	C and Tcl IF	Ability to program the library in both C and Tcl. Easy interchange between Tcl and C code.
6	Up to date	Buildable with actual tools and operating systems.

Table 3: Requirements for the Tcl 3D-library

2.2 Discussion of available solutions

Glut/Tk uses the GLUT library. Although GLUT is available on different platforms, it has not been actively supported for quite some time. GLUT contains lot of operating system dependent code covering features like event handling or simple menus, features that are already handled by Tk.

Tom has not been updated for a while and consists of a hand-crafted interface to OpenGL.

Togl allows programming OpenGL in C only.

X-Bit is not available as source code.

Out of the 6 possible solutions listed in Table 2 the following packages left over for a more detailled inspection:

Tkogl, currently maintained by the University of British Columbia in Vancouver and **Frustum** by Roger E Critchlow Jr, which is not maintained by the author anymore.

Both have a very similar approach: Wrap the OpenGL core libraries GL and GLU with SWIG ([6]), and display the contents in a Tk widget.

The next table lists the features which didn't fit my requirements:

	TkOgl	**Frustum**
Use of old SWIG version 1.1	Yes	Yes
OpenGL header files modified	Yes	Yes
Handcrafted tables for mapping GLenums	Ys	No

SWIG 1.1 is not supported any more and may be not available on newer versions of operating systems. The current SWIG version is 1.3.24 and this version offers lots of new features.

Edited OpenGL header files need manual changes when compiling on platforms with a newer OpenGL version, otherwise the additional commands are not available. Changes in the API have to be done by hand, too.

OpenGL declares a bunch of enumerations, as can be seen in the following table. These differ from platform to platform and keeping them up-to-date manually for the different platforms and versions would not be reasonable.

GL_VENDOR	SGI	Microsoft Corporation
GL_VERSION	1.1 Irix 6.5	1.1.0
GLU_VERSION	1.2 Irix 6.5	1.2.2.0 Microsoft Corporation
Number of gl commands	485	352
Number of glu commands	68	67
Number of gl enums	1041	588
Number of glu enums	138	116

So the final decision was to follow the Frustum approach, which only needed two parts to be cleaned up and extended.

3 Implementation

3.1 SWIG-based OpenGL wapper

The first task was to create a language binding for the OpenGL core libraries GL and GLU with the help of SWIG ([6]). As stated earlier, it should work with an actual version of SWIG and the OpenGL header files should not be touched.

Due to the new version of SWIG and it's extended typemap features it was possible to generate a consistent mapping between C functions and equivalent Tcl commands without changing the OpenGL header files gl.h and glu.h.

The following tables show, how parameters and return values of the C based OpenGL functions are mapped to Tcl command parameters and return values. Every type of parameter is explained with a typical example.

Note:

- The notation `TYPE` stands for any scalar value `(GLboolean, GLbyte, GLubyte, GLshort, GLushort, GLint, GLuint, GLfloat, GLdouble)`. It is not used for type `void`.
- The notation `STRUCT` stands for any C struct.

Input parameter	GLenum
C declaration	`void glEnable (GLenum cap);`
C example	`glEnable (GL_BLEND);`
Tcl example	`glEnable GL_BLEND` `glEnable $::GL_BLEND`

`GLenum` as an OpenGL function input parameter can be supplied as numerical value or as name.

Input parameter	GLbitfield
C declaration	`void glClear (GLbitfield mask);`
C example	`glClear (GL_COLOR_BUFFER_BIT);`
Tcl example	`glClear GL_COLOR_BUFFER_BIT` `glClear $::GL_COLOR_BUFFER_BIT`

`GLbitfield` as an OpenGL function input parameter can be supplied as numerical value or as name.

Note:

- A combination of bit masks has to be specified as a numerical value like this:
  `glClear [expr $::GL_COLOR_BUFFER_BIT | $::GL_DEPTH_BUFFER_BIT]`

Input parameter	GLboolean
C declaration	`void glEdgeFlag (GLboolean flag);`
C example	`glEdgeFlag (GL_TRUE);`
Tcl example	`glEdgeFlag GL_TRUE` `glEdgeFlag $::GL_TRUE`

`GLboolean` as an OpenGL function input parameter can be supplied as numerical value or as name.

The mapping of the types `GLenum`, `GLbitfield` and `GLboolean` is handled in file ***consthash.i***.

Input parameter	TYPE
C declaration	`void glTranslatef (GLfloat x, GLfloat y, GLfloat z);`
C example	`glTranslatef (1.0, 2.0, 3.0);` `glTranslatef (x, y, z);`
Tcl example	`glTranslatef 1.0 2.0 3.0` `glTranslatef $x $y $z`

Scalar types as an OpenGL function input parameter must be supplied as numerical value.

The mapping of scalar types is handled by the SWIG standard typemaps.

Input parameter	const TYPE[SIZE], const TYPE *
C declaration	`void glMaterialfv (GLenum face, GLenum pname, const GLfloat *params);`
C example	`GLfloat mat_diffuse = { 0.7, 0.7, 0.7, 1.0 };` `glMaterialfv (GL_FRONT, GL_DIFFUSE, mat_diffuse) ;`
Tcl example	`set mat_diffuse { 0.7 0.7 0.7 1.0 }` `glMaterialfv GL_FRONT GL_DIFFUSE $mat_diffuse`

Constant pointers as an OpenGL function input parameter must be supplied as a Tcl list.

The mapping of const TYPE pointers is handled in file ***autoarray.i***.

Note:

- This type of parameter is typically used to specify small vectors (2D, 3D and 4D) as well as control points for NURBS.
- Unlike in the C version, specifying data with the scalar version of a function (ex. glVertex3f) is faster than the vector version (ex. glVertex3fv) in Tcl.
- Note, that Tcl lists given as parameters to an OpenGL function have to be flat, i.e. they are not allowed to contain sublists. When working with lists of lists, you have to flatten the list, before supplying it as an input parameter to an OpenGL function. One way to do this is shown in the example below.

```
set ctrlpoints {
      {-4.0 -4.0 0.0} {-2.0 4.0 0.0}
      { 2.0 -4.0 0.0} { 4.0 4.0 0.0}
}
glMap1f GL_MAP1_VERTEX_3 0.0 1.0 3 4 [join $::ctrlpoints]
```

Input parameter	const GLvoid *
C declaration	`void glVertexPointer (GLint size, GLenum type, GLsizei stride, const GLvoid *ptr);`

C example	`static GLint vertices[] =`
	`{ 25, 25, 100, 325, 175, 25,`
	`175, 325, 250, 25, 325, 325};`
	`glVertexPointer (2, GL_INT, 0, vertices);`

Tcl example	`set vertices [VectorFromArgs GLint \`
	`25 25 100 325 175 25 \`
	`175 325 250 25 325 325]`
	`glVertexPointer 2 GL_INT 0 $::vertices`

Constant void pointers as an OpenGL function parameter must be given as a pointer to a contiguous piece of memory of appropriate size.

The mapping of const void pointers is handled by the SWIG standard typemaps.

Note:

- The allocation of useable memory can be accomplished with the use of the **`Vector`** command, which is described later in this chapter.
- This type of parameter is typically used to supply image data or vertex arrays. See also the description of the Tk photo mapping later in this chapter.

Output parameter	TYPE *, GLvoid *

C declaration	`void glGetFloatv (GLenum pname, GLfloat *params);`
	
	`void glReadPixels (GLint x, GLint y, GLsizei width,`
	`GLsizei height, GLenum format,`
	`GLenum type, GLvoid *pixels);`

C example	`GLfloat values[2];`
	`glGetFloatv (GL_LINE_WIDTH_GRANULARITY, values);`
	
	`GLubyte *vec = malloc (w * h * 3);`
	`glReadPixels (0, 0, w, h, GL_RGB, GL_UNSIGNED_BYTE, vec);`

Tcl example	`set values [Vector GLfloat 2]`
	`glGetFloatv GL_LINE_WIDTH_GRANULARITY $values`
	
	`set vec [Vector GLubyte [expr $w * $h * 3]]`
	`glReadPixels 0 0 $w $h GL_RGB GL_UNSIGNED_BYTE $vec`

Non-constant pointers as an OpenGL function parameter must be given as a pointer to a contiguous piece of memory of appropriate size.

The mapping of non-constant pointers is handled by the SWIG standard typemaps.

Function return	TYPE, STRUCT *

C declaration	`GLuint glGenLists (GLsizei range);`
	
	`GLUnurbs* gluNewNurbsRenderer (void);`

C example	`GLuint sphereList = glGenLists(1);`
	
	`GLUnurbsObj *theNurb = gluNewNurbsRenderer();`
	`gluNurbsProperty (theNurb, GLU_SAMPLING_TOLERANCE, 25.0);`

Tcl example	`set sphereList [glGenLists 1]`
	
	`set theNurb [gluNewNurbsRenderer]`
	`gluNurbsProperty $theNurb GLU_SAMPLING_TOLERANCE 25.0`

Scalar return values are returned as the numerical value.
Pointer to structs are returned with the standard SWIG mechanism of encoding the pointer in an ASCII string.

The mapping of return values is handled by the SWIG standard typemaps.

Note:

- The next lines show an example of SWIG's pointer encoding:

```
% set theNurb [gluNewNurbsRenderer]
% puts $theNurb
_10fa1500_p_GLUnurbs
```

The returned name can only be used in functions expecting a pointer to the appropriate struct.

Exceptions from the standard rules

The GLU library as specified in header file ***glu.h*** does not provide an API, that is as consistent as the GL core library. So one class of function parameters (`TYPE *`) is handled differently with GLU functions. Arguments of type `TYPE*` are used both as input and output parameters in the C version. In GLU 1.2, which is the current version, most functions use this type as input parameter. Only two functions use this type as an output parameter.
So for GLU functions there is the exception, that `TYPE*` is considered an input parameter and therefore is wrapped as a Tcl list.

Input parameter	**TYPE * (GLU only)**

C declaration	`void gluNurbsCurve (GLUnurbs *nobj, GLint nknots,`
	`  **GLfloat *knot**, GLint stride,`
	`  **GLfloat *ctlarray**, GLint order,`
	`  GLenum type);`

C example	`GLfloat curvePt[4][2] = {{0.25, 0.5}, {0.25, 0.75},`
	`  {0.75, 0.75}, {0.75, 0.5}};`
	`GLfloat curveKnots[8] = {0.0, 0.0, 0.0, 0.0,`
	`  1.0, 1.0, 1.0, 1.0};`
	`gluNurbsCurve (theNurb, 8, curveKnots, 2,`
	`  &curvePt[0][0], 4, GLU_MAP1_TRIM_2);`

Tcl example	`set curvePt {0.25 0.5  0.25 0.75  0.75 0.75  0.75 0.5}`
	`set curveKnots {0.0 0.0 0.0 0.0 1.0 1.0 1.0 1.0}`
	`gluNurbsCurve $theNurb 8 $curveKnots 2 $curvePt 4`
	`  GLU_MAP1_TRIM_2`

The two aforementioned functions, which provide output parameters with `TYPE*` are `gluProject` and `gluUnProject`. These are handled as a special case in the SWIG interface file glu.i. The 3 output parameters are given the keyword OUTPUT, so SWIG handles them in a special way: SWIG builds a list consisting of the normal function return value, and all parameters marked with that keyword. This list will be the return value of the corresponding Tcl command.

Definition in glu.h	**Redefinition in SWIG interface file glu.i**

`extern GLint gluUnProject (`
`  GLdouble winX, GLdouble winY,`
`  GLdouble winZ,`
`  const GLdouble *model,`
`  const GLdouble *proj,`
`  const GLint *view,`
`  **GLdouble* objX,**`
`  **GLdouble* objY,**`
`  **GLdouble* objZ**);`
	`GLint gluUnProject (`
	`  GLdouble winX, GLdouble winY,`
	`  GLdouble winZ,`
	`  const GLdouble *model,`
	`  const GLdouble *proj,`
	`  const GLint *view,`
	`  **GLdouble* OUTPUT,**`
	`  **GLdouble* OUTPUT,**`
	`  **GLdouble* OUTPUT**);`

Example usage (see Redbook ([1]) example ***unproject.tcl*** for complete code):

```
glGetIntegerv GL_VIEWPORT $viewport
glGetDoublev  GL_MODELVIEW_MATRIX  $mvmatrix
glGetDoublev  GL_PROJECTION_MATRIX $projmatrix
set viewList [VectorToList $viewport 4]
set mvList   [VectorToList $mvmatrix 16]
set projList [VectorToList $projmatrix 16]

set realy [expr [$viewport get 3] - $y - 1]
set winList [gluUnProject $x $realy 0.0 $mvList $projList $viewList]
puts "gluUnProject return value: [lindex $winList 0]"
puts [format "World coords at z=0.0 are (%f, %f, %f)" \
     [lindex $winList 1] [lindex $winList 2] [lindex $winList 3]]
```

3.2 Extension of the Togl wdget

Now that we have a Tcl binding of the OpenGL functionality, we need to be able to display the 3D contents.

Togl is an actively maintained Tk widget with support to display OpenGL graphics, but the drawing commands have to be specified in C.
To be usable from the Tcl level, it has been extended to support 3 new configuration options for specifying Tcl callback commands:

`-createproc TclCommandName`	Called when a new widget is created.
`-reshapeproc TclCommandName`	Called when the widget's size is changed.
`-displayproc TclCommandname`	Called when the widget's content needs to be redrawn.

These configuration options behave like standard Tcl options as shown in the example below:

```
% package require Togl
1.6
% togl .t
% .t configure -displayproc tclDisplayFunc
% .t configure -displayproc
-displayproc displayproc Displayproc {} tclDisplayFunc
```

So a minimal 3D application looks like the following “Hello, World” OpenGL program.

```
# hello.tcl

package require tclogl
package require Togl

proc tclDisplayFunc {} {
    glClear GL_COLOR_BUFFER_BIT

    # draw white polygon (rectangle) with corners at
    # (0.25, 0.25, 0.0) and (0.75, 0.75, 0.0)
    glColor3f 1.0 1.0 1.0
    glBegin GL_POLYGON
        glVertex3f 0.25 0.25 0.0
        glVertex3f 0.75 0.25 0.0
        glVertex3f 0.75 0.75 0.0
        glVertex3f 0.25 0.75 0.0
    glEnd
    glFlush
}

proc tclCreateFunc {} {
    # select clearing color
    glClearColor 0.0 0.0 0.0 0.0

    # initialize viewing values
    glMatrixMode GL_PROJECTION
    glLoadIdentity
    glOrtho 0.0 1.0 0.0 1.0 -1.0 1.0
}

proc tclReshapeFunc { w h } {
    .fr.toglwin postredisplay
}
```

```
frame .fr
pack .fr -expand 1 -fill both
togl .fr.toglwin -width 250 -height 250 -double false \
                 -createproc tclCreateFunc
.fr.toglwin configure -displayproc tclDisplayFunc \
                      -reshapeproc tclReshapeFunc
grid .fr.toglwin -row 0 -column 0 -sticky news

bind . <Key-Escape> "exit"
```

Note that `-createproc` is not effective, when specified in the configure subcommand. It has to be specified at widget creation time.

The changes in the widget code allow Togl to execute Tcl callbacks with the help of Tcl_Eval, while still maintaining 100% of it's original functionality. Only a few lines had to be added or changed in the Togl source code:

1. Add the 3 new configuration options to the `Tk_ConfigSpec` list.
2. Declaration and definition of the 3 new internal evaluation functions: tcloglCreateProc, tcloglDisplayProc, tcloglReshapeProc.
3. Change the default callbacks to point to the new internal evaluation functions.

These 3 changes are shown with the create callback as example:

1.
```
    {TK_CONFIG_STRING|TK_CONFIG_NULL_OK, "-createproc", "createproc",
    "Createproc", NULL, Tk_Offset(struct Togl, createCallback), 0, NULL},
```

2.
```
    static int tcloglCreateProc (struct Togl *togl) {
      if (togl->createCallback) {
        if (Tcl_Eval (Togl_Interp(togl), togl->createCallback) != TCL_OK) {
          Tcl_BackgroundError (Togl_Interp(togl));
          free (togl->createCallback);
          togl->createCallback = NULL;
          return TCL_ERROR;
        }
      }
      return TCL_OK;
  }
```

3.
```
  static Togl_Callback *DefaultCreateProc  = tcloglCreateProc;
```

3.3 Utility functions

All of the features listed in this chapter are not necessary for operation, but offer extended or easier functionality.

3.3.1 The Vector command

As stated in chapter 3.1, some of the OpenGL functions need a pointer to a contiguous block of allocated memory. SWIG already provides a feature to automatically generate wrapper functions for allocating and freeing memory of any type. This feature `%array_functions` also creates setter and getter functions for accessing the allocated memory.

The following definitions provided in file ***tclogl.i*** create the accessor functions for the OpenGL base types:

```
// Generate array functions (new, delete, getitem, setitem) for the
// following types.

%array_functions(unsigned char,GLboolean)
%array_functions(signed char,GLbyte)
%array_functions(unsigned char,GLubyte)
%array_functions(short,GLshort)
%array_functions(unsigned short,GLushort)
%array_functions(int,GLint)
%array_functions(unsigned int,GLuint)
%array_functions(float,GLfloat)
%array_functions(double,GLdouble)
```

The generated wrapper code looks like this (Example shown for `GLdouble`):

```
static double *new_GLdouble(int nelements) {
  return (double *) calloc(nelements,sizeof(double));
}

static void delete_GLdouble(double *ary) {
  free(ary);
}

static double GLdouble_getitem(double *ary, int index) {
    return ary[index];
}

static void GLdouble_setitem(double *ary, int index, double value) {
    ary[index] = value;
}
```

The file ***tcloglVector.tcl*** contains additional Tcl commands for encapsulation of these low-level accessor functions.

Tcl command	Explanation
`Vector`	Call the memory allocation routine new_* and create an OO like Tcl interface. (See example below)
`VectorFromList`	Create a new Vector from given Tcl list.
`VectorFromArgs`	Create a new Vector from given aruments.
`VectorFromString`	Create a new GLubyte Vector from given string.
`VectorToString`	Copy the contents of a GLubyte Vector into a string.
`VectorToList`	Copy the contents of a Vector into a Tcl list.

The following example shows the usage of the base `Vector` command.

```
set ind 23
set  vec [Vector GLfloat 123] ; # Create a new Vector of size 123 GLfloats
set x [$vec get $ind]         ; # Get element at index 23
$vec set $ind 1017.0          ; # Set element at index 23 to 1017.0
$vec delete                   ; # Free the allocated memory
```

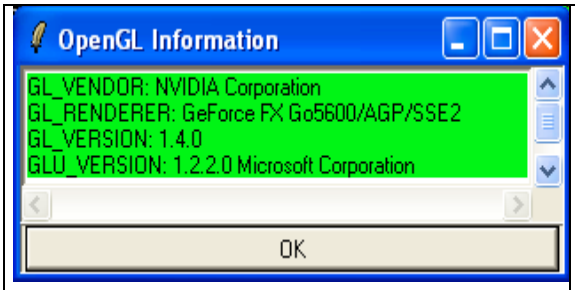
3.3.2 Information utilities

In file ***tcloglInfo.tcl*** three utility functions are currently implemented to get information about the OpenGL version, the installed extensions, as well as the current OpenGL state.

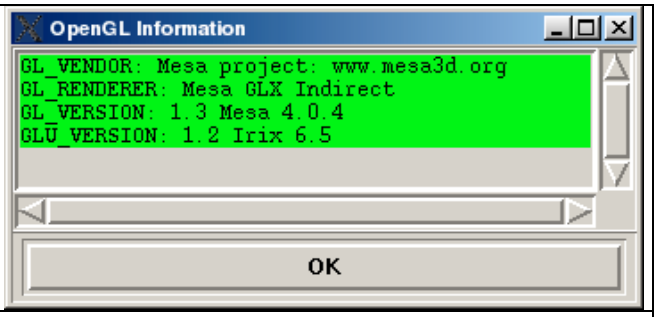
`tcloglGetVersions`	Query the OpenGL library with the keys `GL_VENDOR`, `GL_RENDERER`, `GL_VERSION`, `GLU_VERSION` and return the results as a list of key-value pairs.

The following code snippet shows how to call `tcloglGetVersions` and place the result in a text widget.

```
foreach glInfo [tcloglGetVersions] {
        set msgStr "[lindex $glInfo 0]: [lindex $glInfo 1]\n"
        $textId insert end $msgStr
}
```



Example output of `glGetVersions`



Example output of `glGetVersions` on SGI/Linux

`tcloglGetExtensions`	Query the OpenGL library with the keys `GL_EXTENSIONS` and `GLU_EXTENSIONS` and return the results as a list of key-value pairs.

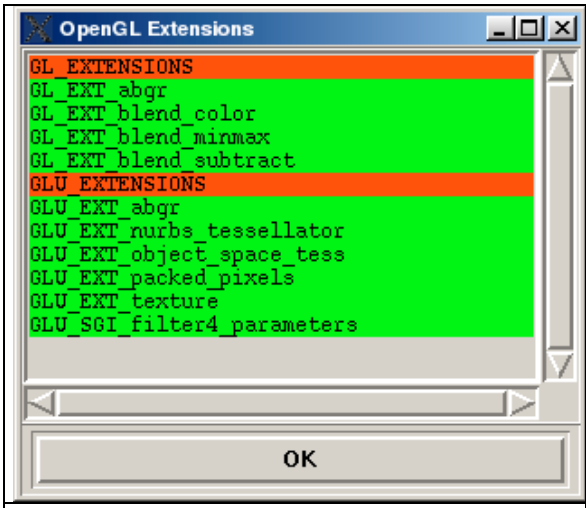
The following code snippet shows how to call `tcloglGetExtensions` and place the result in a text widget.

```
foreach glInfo [tcloglGetExtensions] {
    set msgStr "[lindex $glInfo 0]\n"
    $textId insert end $msgStr type
    foreach ext [lsort [split [string trim [lindex $glInfo 1]]]] {
        set msgStr "$ext\n"
        $textId insert end $msgStr name
    }
}
```

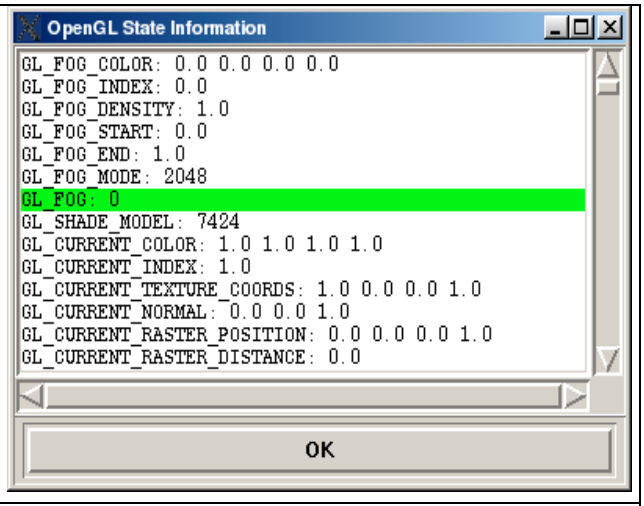
`tcloglGetStates`	Query all state variables of the OpenGL library and return the results as a list of sub-lists. Each sublist contains the querying command used, the key and the value(s).

The following code snippet shows how to call `tcloglGetStates` and place the result in a text widget.

```
foreach glState [tcloglGetStates] {
    set msgStr "[lindex $glState 1]: [lrange $glState 2 end]\n"
    if { [string compare [lindex $glState 0] "glIsEnabled"] == 0 } {
        set tag bool
    } else {
        set tag other
    }
    $textId insert end $msgStr $tag
}
```



Example output of `glGetExtensions`



Example output of `glGetStates`

Note:
The functions `glGetString` and `gluGetString` as well as the corresponding high-level functions `tcloglGetVersions` and `tcloglGetExtensions` only return correct values, if a Togl window has been opened, i.e. a rendering context has been established.

3.3.3 Tk photo mapping

In file ***tkphoto.i*** the following C functions are implemented to provide access to the Tk photo image functionality.

Tcl command	Usage
`PhotoChans`	Return the nuber of channels of a Tk photo.
`Photo2Vector`	Copy a Tk photo into a `Vector` in OpenGL raw image format. The `Vector` must have been allocated with the approriate size and type.
`Vector2Photo`	Copy from OpenGL raw image format into a Tk photo. The photo image must have been initialized with the appropriate size and type.

These functions are best explained by looking at the following code excerpts from the simple image viewer ***imgViewer.tcl***:

Example 1: Read an image into a Tk photo and use it as a texture map. Note: Texture map images must have width and height, that are multiples of 2.

```
proc ReadImg { imgName } {
    global gPo

    set retVal [catch {set phImg [image create photo -file $imgName]} err1]
    if { $retVal != 0 } {
        puts "Failure reading image $imgName"
    } else {
        set w [image width  $phImg]
        set h [image height $phImg]
        set sqr [GetBestSquare $w $h]
        set gPo(texScaleS) [expr double ($w) / $sqr]
        set gPo(texScaleT) [expr double ($h) / $sqr]
        set sqrPhoto [image create photo -width $sqr -height $sqr]
        $sqrPhoto copy $phImg -from 0 0 $w $h -to 0 [expr $sqr -$h]
        update
        set vecImg [Vector GLubyte [expr $sqr * $sqr * 4]]
        Photo2Vector $sqrPhoto $vecImg
        image delete $phImg
        image delete $sqrPhoto
        glTexParameteri GL_TEXTURE_2D GL_TEXTURE_WRAP_S $::GL_CLAMP
        glTexParameteri GL_TEXTURE_2D GL_TEXTURE_WRAP_T $::GL_CLAMP
        glTexParameteri GL_TEXTURE_2D GL_TEXTURE_MAG_FILTER $::GL_NEAREST
        glTexParameteri GL_TEXTURE_2D GL_TEXTURE_MIN_FILTER $::GL_NEAREST
        glTexImage2D GL_TEXTURE_2D 0 4 \
                     $sqr $sqr \
                     0 GL_RGBA GL_UNSIGNED_BYTE $vecImg
        tclDisplayFunc
    }
}
```

Example 2: Read an image from the OpenGL framebuffer and save it with the Img library.

```
proc SaveImg { imgName } {
    global gPo

    set w $gPo(toglWidth)
    set h $gPo(toglHeight)
    set numChans 4
    set vec [Vector GLubyte [expr $w * $h * $numChans]]
    glReadPixels 0 0 $w $h GL_RGBA GL_UNSIGNED_BYTE $vec
    set ph [image create photo -width $w -height $h]
    Vector2Photo $vec $ph $w $h $numChans
    set fmt [string range [file extension $imgName] 1 end]
    $ph write $imgName -format $fmt
}
```

The actual size of the Togl window (`gPo(toglWidth)`, `gPo(toglHeight)`), which is needed in command `SaveImg`, can be saved in a global variable when the reshape callback is executed.

```
proc tclReshapeFunc { w h } {
    global gPo

    set gPo(toglWidth)  $w
    set gPo(toglHeight) $h
    ...
}
```

3.3.4 Additional tclogl utilities

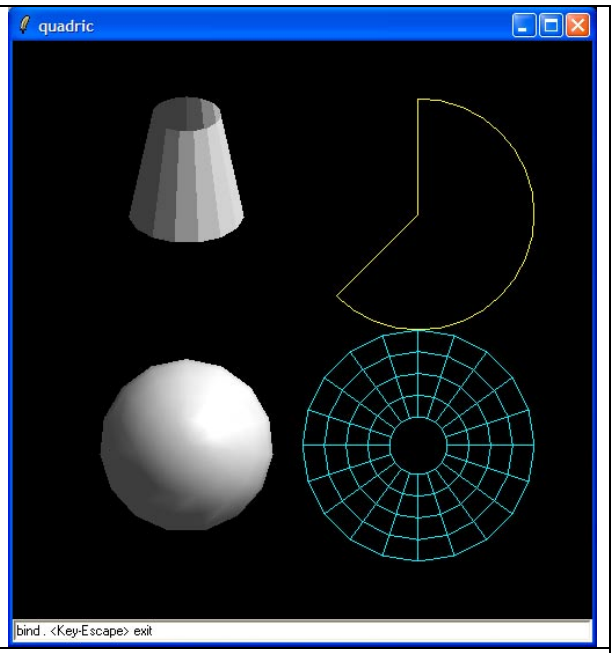
The utilities in this chapter have been added for testing and demonstration purposes.

GLUT shapes library

The shape objects implemented in the GLUT library are available under the same names for running the test programs of the OpenGL redbook ([1]).

Solid shapes	Wire shapes
`glutSolidCube`	`glutWireCube`
`glutSolidCone`	`glutWireCone`
`glutSolidSphere`	`glutWireSphere`
`glutSolidTorus`	`glutWireTorus`
`glutSolidTetrahedron`	`glutWireTetrahedron`
`glutSolidOctahedron`	`glutWireOctahedron`
`glutSolidDodecahedron`	`glutWireDodecahedron`
`glutSolidIcosahedron`	`glutWireIcosahedron`
`glutSolidTeapot`	`glutWireTeapot`

Some GLUT shapes



Quadrics examples

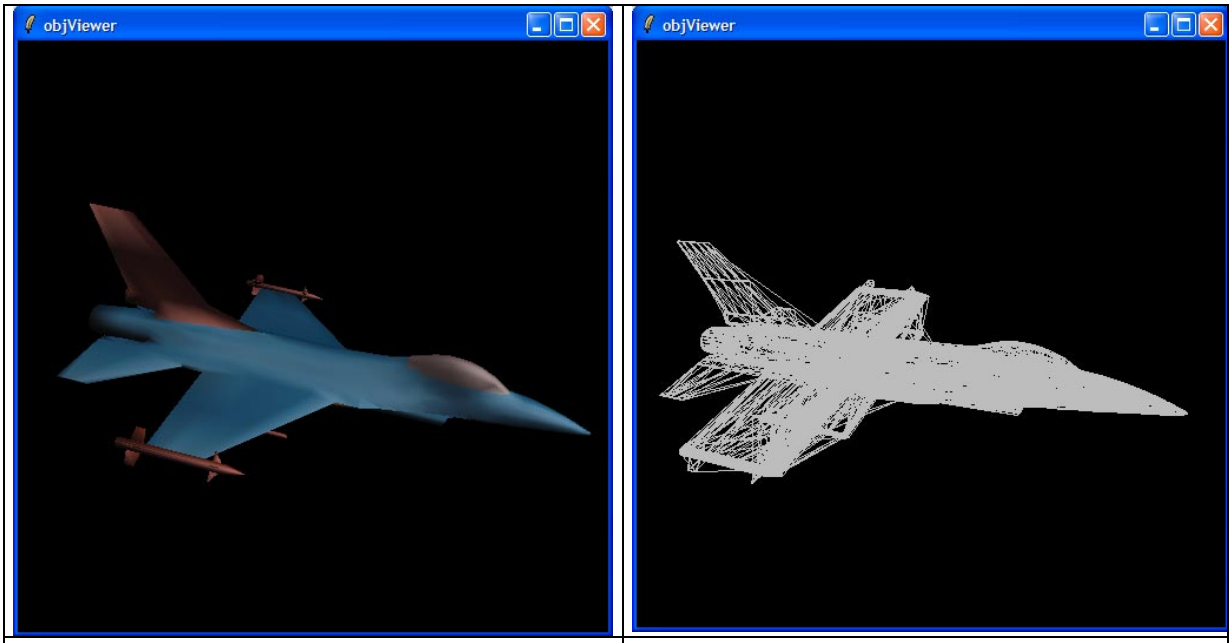
The shapes library consists of the C files (***teapot.c*** for the teapot, ***shapes.c*** for all other shapes and the common header file ***shapes.h***) and the Tcl file ***tcloglShapes.tcl***.

The shape library also acts as a demonstration, how to extend the tclogl package with C code.
The steps necessary are:

1. Compile your C source files (***shapes.c***, ***teapot.c***)
2. Put the name of the header file (***shapes.h***) into SWIG interface file ***util.i***.
3. Call SWIG to create a new wrapper file.
4. Relink your dynamic library with the new object files (***shapes.o, teapot.o***).

Alias/Wavefront modelfile reader

A simple viewer for 3D models has been implemented in ***objViewer.tcl***
It can read model files in Alias/Wavefront format. The code to read and draw the models is taken from Nate Robin's OpenGL tutorial ([4]). The corresponding files are ***glm.c*** and ***glm.h***.



Smooth shaded model file	Line drawing of model file

4 Caveats / Common pitfalls

Some OpenGL functions expect an integer or floating point value, which is often given in C code examples with an enumeration, as shown in the next example:

```
extern void glTexParameteri ( GLenum target, GLenum pname, GLint param );
```

It is called in C typically as follows:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

As the 3rd parameter is not of type `GLenum`, you have to specify the numerical value here:

```
glTexParameteri GL_TEXTURE_2D GL_TEXTURE_WRAP_S $::GL_REPEAT
glTexParameteri GL_TEXTURE_2D GL_TEXTURE_MAG_FILTER $::GL_NEAREST
```

If called with the enumeration name:

```
glTexParameteri GL_TEXTURE_2D GL_TEXTURE_WRAP_S GL_REPEAT
```

you will get an error message like this: expected integer but got "GL_REPEAT"

To correctly wrap the OpenGL libraries, a version of SWIG greater or equal to 1.3.19 is needed.

For performance reasons use OpenGL display list, where possible.

5 Open issues

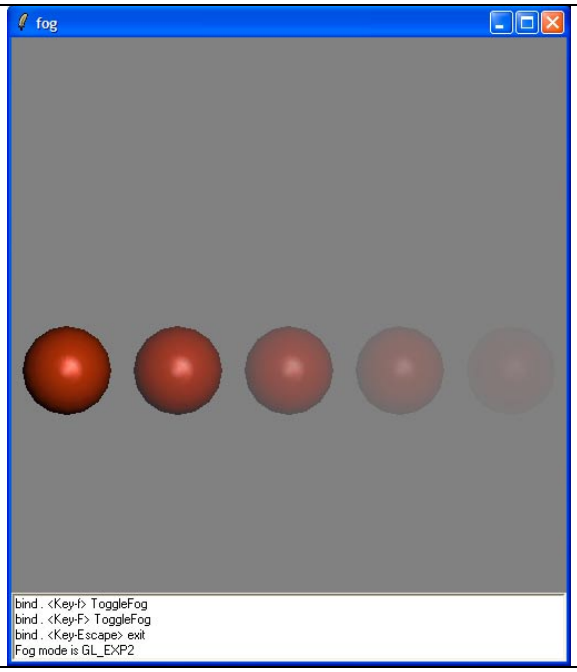
- GLU callbacks are currently not supported. This implies, that tesselation does not work, because this functionality relies heavily on the usage of C callback functions.
- There is currently no possibility to specify a color map for OpenGL indexed mode. As color maps depend on the underlying windowing system, this feature should be handled by the Togl widget.
- Picking with depth values does not work correctly, as depth is returned as an unsigned int, mapping the internal floating-point depth values [0.0 .. 1.0] to the range [0 .. 2^{32} –1]. As Tcl only supports signed integers, some depth values are incorrectly transferred into the Tcl commands.
- The handling of Tcl errors inside of Togl callbacks could be improved.
- To evaluate the Tcl callbacks, Tcl_Eval is currently used, which does not compile the script into bytecode. Use the object-interface instead.

6 Results

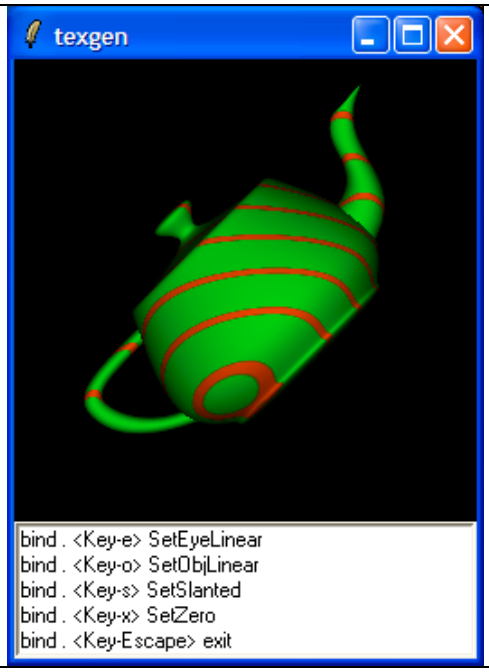
To test the correctness and completeness of the wrapped OpenGL library, the examples of the Redbook ([1]), which are available as C code ([2]), were ported into equivalent Tcl code.

The Redbook contains 56 examples, showing many aspects of OpenGL features.
52 of them have been successfully converted into equivalent Tcl scripts and compared on different computers against the C version. All of them gave identical results, except depth-picking in some cases (see above).

Two of the missing four examples deal with tesselation, which is currently not supported, as stated in the previous chapter. The other two test programs not yet ported deal with color index mode, which is not yet implemented, too.
Tesselation and color index mode both are rarely used features, at least in my applications.

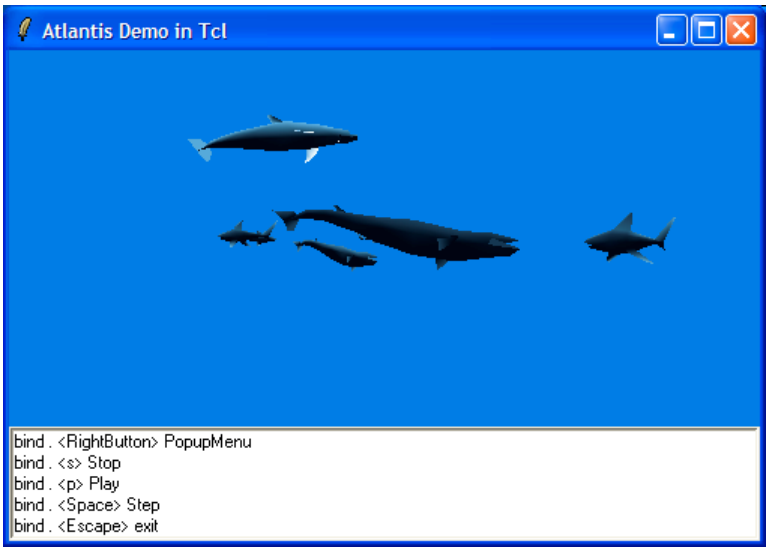


Redbook example ***fog.tcl***



Redbook example ***texgen.tcl***

To demonstrate the easy transition of C to Tcl code, a more complex program, the "Atlantis demo" ([3]) has been ported. It behaves like it's C pendant, but performs a lot slower, as it has been not been optimized for running as a Tcl script.



Finally a simple image viewer has been implemented that allows realtime scaling of the image. The images can be read from files in all formats supported by the Img extension. The stretched image may also be written out to an image file.

The Togl and tclogl packages have been generated and tested on the following platfoms:

Operating system	Compiler version	SWIG version
Windows XP	Visual C++ 6.0	1.3.19
SuSE Linux 9.0	gcc 3.3.1	1.3.19
IRIX 6.5	MIPSpro cc 7.30	1.3.24

The source code for the tclogl package, i.e. the modified Togl code and the SWIG interface files for the OpenGL wrapper, as well as the test and demo programs can be downloaded from my home page ([7]). A binary version of the actual SWIG version 1.3.24 for IRIX is available there, too.

7 References

[1] Woo, Neider, Davis: OpenGL Programming Guide, Addison-Wesley, **"The Redbook"**

[2] Redbook C examples: http://www.opengl.org/resources/code/basics/redbook

[3] Atlantis demo: http://www.opengl.org/resources/code/glut/glut_examples/demos/demos.html

[4] Nate Robins OpenGL tutorial: http://www.xmission.com/~nate/tutors.html

[5] OpenGL Wiki page: http://wiki.tcl.tk/2237

[6] SWIG (Simplified Wrapper and Interface Generator): http://www.swig.org

[7] Paul Obermeier's Portable Software: http://www.poSoft.de