A Tk OpenGL widget

Claudio Esperança
COPPE, Engenharia de Sistemas e Computação
Universidade Federal do Rio de Janeiro

A Tk OpenGL widget

Claudio Esperança
COPPE, Engenharia de Sistemas e Computação
Universidade Federal do Rio de Janeiro
Cidade Universitária, CT, Sala H-319
Rio de Janeiro, RJ 21949-900, Brazil

Abstract

We present TkOGL, a Tk widget that enables the creation and display of 3D graphics using the OpenGL API. Our approach features a reasonably complete Tcl binding to the core OpenGL functionality, as well as a set of extensions that implement a higher-level interface to many common utility functions such as those provided by the OpenGL utility library (GLU).

1. Introduction

OpenGL [1] is becoming a standard Application Program Interface (API) for writing portable 3D computer graphics programs. On the other hand, the *Tk* toolkit offers a portable and powerful environment for the development of graphical user interfaces. It is to be expected then, that the merging of both capabilities should appeal to a wide audience. In fact, many attempts to do exactly that have been reported. Among these, we cite the *Tiger* system [2] and the *Togl widget* [3]. For various reasons, however, these packages did not meet my expectations. For instance, *Tiger* mimics the OpenGL API almost exactly, which makes the creation of simple 3D graphics unnecessarily complicated due to the inexistence of higher-level constructs. On the other hand, *Togl,* while providing the means to open a window for displaying OpenGL graphics, does not include Tcl bindings for any of the OpenGL rendering functions, thus forcing the user to program in C or C++.

Our primary purpose in writing the Tkogl widget was to enable both the experienced and novice users to generate and display 3D models in a concise manner. Moreover, the widget takes care of low-level tasks related to the embedding of OpenGL on a given window system, such as adjusting viewports to reflect window resize events and buffer swapping for double-buffered visuals.

The package was initially developed on an IBM RS-6000 workstation running AIX v3.2.5 and tested both with "real" OpenGL and with a free implementation of the OpenGL API, namely, the Mesa 3-D graphics library [4]. It was later ported to PCs running Microsoft's OpenGL implementation under Windows95. Currently, the package is known to work with Tcl 7.5/Tk 4.1 and Tcl 7.6/Tk 4.2. The distribution contains source code and Makefiles for some popular architecture/operating system combinations. In order to facilitate its installation in PCs, a pre-compiled DLL (dynamically loadable library) is also provided. The TkOGL home page address can be found at [5].

2. Design Issues

OpenGL [1] is a software interface to 3D graphics which was designed to provide optimum performance on client/server architectures. Thus, a typical application (client) consists of calls to OpenGL functions which are translated into messages that are sent to the graphics hardware (server), where they are interpreted and executed. Since it was designed to be portable to different architectures, the exact protocol involved in OpenGL messaging may vary. While this flexibilty is one of the strong points of OpenGL, in practice a usable OpenGL implementation must define several interface details in a non-portable manner. In particular, creating a window for displaying OpenGL graphics is non-trivial and eminently architecture-dependent. This task is further complicated when we consider that our aim is to embed such windows in frames managed by *Tk.* Some of the issues involved in this task are:

- How to create a child window which can be recognized by *Tk.*
- How to generate the architecture-dependent OpenGL runtime data structures (also called *contexts*).
- How to cope with window system-specific events. For instance, Microsoft's implementation of OpenGL only allows contexts to be created in response to an event which is not handled by the system-independent event dispatching mechanism of *Tk*.

- How to allocate other window system-specific data structures such as color maps.

These issues, although important, are not addressed further in this paper since they are not of interest to the typical user. The curious reader will be able to find some solutions to these problems by perusing the source code included in the Tkogl distribution [5].

Another important aspect concerning the integration of OpenGL and *Tcl/Tk* is the design of an appropriate set of Tcl bindings. Ideally, we would like an OpenGL widget to provide an interface which is similar to other *Tk* widgets. For instance, the `canvas` widget might serve as a model, since it provides the funcionality for generating 2D drawings. Unfortunately, however, 3D graphics are substantially more involved, and the overall approach used by `canvas` (i.e., to provide a few graphical item types such as rectangles and ovals which can be created and configured) cannot be employed in quite the same manner.
The rest of this paper describes our approach to this problem.

3. A Simple Application Using Tkogl

The integration between OpenGL and Tk is achieved by a `package` called `Tkogl`, which in Unix-based installations is statically linked in the extended Tcl/Tk windowing shell called `glwish`. Under Windows95, the package can be dynamically loaded by executing a corresponding `package require` command. In any case, a Tcl script which uses the package should include the following line:

```
package require Tkogl
```

Once it is ascertained that the package is loaded, one or more windows can be created for displaying OpenGL graphics. Such windows can be created in a similar way to other Tk widgets by using the `OGLwin` command, which has the following format:

`OGLwin` *pathName ? option ... ?*

where each *option* can be one of the following:

`-accumsize` *accumSize* specifies that the accumulation buffer should support *accumSize* bit planes for each of the red, green and blue components. If an alpha component for the color buffer has been requested, the same number of bit planes is also requested for the alpha component of the accumulation buffer. By default, no accumulation buffer is requested.

`-alphasize` *alphaSize* specifies that the color buffer should support *alphaSize* bit planes for the alpha component. By default, no alpha bit planes are requested.

`-aspectratio` *ratio* forces the viewport of the window to the width/height fraction given by *ratio*, which should be a positive floating point number. The viewport is then defined as the biggest possible rectangle with the specified aspect ratio centered inside the window. If *ratio* is 0.0 (the default), no aspect ratio is enforced, which means that the viewport will always take the same shape as the window.

`-context` *pathName2* makes the OpenGL context of *pathName* share display lists with that of *pathName2*, which should also be the name of an OGLwin widget.

`-depthsize` *depthSize* specifies the number of bit planes for the depth buffer (also called *z-buffer*). By default, this number is 16. A *depthSize* of 0 means that no depth buffer is required.

`-doublebuffer` *doubleFlag* specifies whether or not a double buffered visual will be used (true, by default).

`-height` *height* specifies the height of the window in pixels. Default:300.

`-stencilsize` *stencilSize* specifies the number of bit planes requested for the stencil buffer (zero, by default).

`-width` *width* specifies the width of the window in pixels. Default:300.

Currently, OGLwin can only be used to create windows which will use the RGBA color model. By default, OGLwin creates a double-buffered RGB window with the biggest number of bitplanes supported by the current software/hardware environment. The configuration options described above can be used to allocate additional buffers, e.g., an accumulation or a stencil buffer. If the requested buffers cannot be allocated, then OGLwin fails, producing a standard Tcl error result.

An OpenGL window is typically created for visualizing a series of graphical objects. In most window systems, the contents of the window must be redrawn every once in a while, for instance, when the window is resized or deiconified. Usually, quite a few OpenGL rendering commands must be executed in order to reproduce the contents of the window. Although we aim to be able to generate any OpenGL command from within a Tcl script, it would be very time-consuming to interpret a very long sequence of Tcl commands every time a given OpenGL window needed to be redrawn. Fortunately, OpenGL offers a display list capability, whereby several commands can be pre-compiled and stored in the display server, ready to be re-executed as needed. Thus, a sensible management of an OpenGL window (such as the one created by the `OGLwin` command) is to reserve a display list which will contain all rendering commands that are to be executed whenever the window needs to be redrawn. In this document, we refer to such a list as the *main list*. In addition to calling the main list whenever a redraw is needed, the widget issues **glFlush** command and takes care of swapping the front and back buffers (when a double-buffered visual is being used). The contents of the main display list can be redefined by means of the `main` widget command, which has the following format:

pathName `main` *? option ... option ?*

where

pathName	is the name of an OpenGL window.
option	is one of the OpenGL commands currently supported by the package. These will be described later on.

The program listing in Example 1 below shows a very minimal script that creates a window to display a triangle. The display produced by that program is shown in Figure 1.

```
package require Tkogl
OGLwin .gl
pack .gl
.gl main -clear colorbuffer \
   -begin triangles \
   -vertex -1 -1 \
   -vertex 0 1 \
   -vertex 1 -1 \
   -end
```

Example 1: A simple script to display a triangle.

Notice that the script above relies on several variables of the OpenGL state machine having their initial default values. For instance, the default value of the **Color** state is white, while the the default value of the **ClearColor** state is black, which means that the triangle will be drawn in white over a black background.

Instead of using the main display list mechanism for keeping the window updated, it is also possible set up a script to be executed every time an *Expose* event is caught by Tk. In this case, instead of using the main widget command to set up the main display list, the

Figure 1: Display produced by the script of Example 1

same OpenGL commands can be issued by means of the `eval` widget command, which has the following syntax:

pathName `eval` *? option ... option ?*

where *pathName* and *option* have the same meanings as in the `main` command.

For instance, it is possible to rewrite our minimal script to catch Expose events directly. This is shown in Example 2 below.

It should be noticed that the default display list mechanism is usually superior to catching events and redisplaying the picture. This is because in the former case all OpenGL commands are already stored in a display list in the server, while in the latter case, all commands must be reinterpreted and transmitted from the client to the server every time the window must be redrawn.

```
package require Tkogl
pack .gl
bind .gl <Expose> {
   .gl eval -clear colorbuffer \
      -begin triangles \
      -vertex -1 -1 \
      -vertex 0 1 \
      -vertex 1 -1 \
      -end
}
```

Example 2: Displays a diagonal line by catching *Expose* events and redrawing the picture with the `eval` widget command.

4. OpenGL option commands

Many OGLwin widget commands (e.g., `eval`, `main`) require a list of options that denote OpenGL commands. The overall format of such options is

glCommandName ? arg ... arg ?

where

glCommandName is a Tcl string that denotes an equivalent OpenGLcommand. The string corresponding to a given OpenGL procedure is the name of that procedure stripped of its **gl** prefix and of eventual data type suffix. Mixed upper- and lowercase characters can be used. Thus, for instance, procedure **glMatrixMode** corresponds to option `-matrixmode` (other lower/uppercase combinations such as `-MatrixMode` are also acceptable), procedure **glColor3f** corresponds to option `-color`, and so on.

arg is a Tcl string equivalent to an argument in the corresponding OpenGL command. The following rules are useful to determine how OpenGL procedure arguments are mapped into equivalent Tcl strings:

- Arguments of type **GLenum** are mapped into a string with the same spelling as that of the equivalent constant, except that the GL prefix is dropped, as well as any underscore ('_') characters. Mixed upper- and lowercase characters can be used. For example, constant **GL_DEPTH_TEST** may be written either as `depthtest` or `DepthTest`.
- Numeric arguments are represented by equivalent Tcl strings. Integer types (e.g. **GLint**, **GLuint**) are parsed as integer Tcl values and floating-point types (e.g., **GLfloat**, **GLdouble**) are parsed as floating-point values.
- When the same OpenGL function supports both integer and floating-point variants of the same function, the floating-point (**GLfloat**) variant is implemented. For example, command

  ```
  -color 1 0 0
  ```

  is the same as

  glColor3f (1.0, 0.0, 0.0);
- If an OpenGL procedure requires a vector argument, this is supported by spelling out the contents of the vector as discrete *arg*'s. For instance, the "C" code fragment

  ```
  GLfloat ctrlpoints [4][3] = {
     {-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, &ctrlpoints[0][0]);
  ```

  would be translated into Tcl as

  ```
  -map1 map1vertex3 0 1 3 4 \
     -4 -4 0  -2 4 0 \
     2 -4 0   4 4 0
  ```
- In the case of procedures such as **glClear**, which require bit masks as arguments, the individual bit mask constants are mapped to strings in much the same way as **GLenum** constants, except that the **_BIT** suffix is also dropped. Furthermore, the bit mask is assumed to be a bitwise "or" ofall *arg*'s. For instance,

  **glClear (GL_COLOR_BUFFER_BIT |
  GL_DEPTH_BUFFER_BIT);**

  becomes

  ```
  -clear colorbuffer \
         depthbuffer
  ```

Most OpenGL procedures have equivalent option commands. In a few cases, the argument lists of an option command and its associated OpenGL procedure have slightly different argument lists, chiefly those that deal with textures and images. Also, some procedures of the OpenGL Utility Library (**glu**) were also implemented as option commands. The complete list of option commands can be found in the documentation included in the Tkogl distribution [5].

5. OGLwin widget commands

Although most of OpenGL's capabilities could be exercised by using the `eval` and `main` widget commands, we felt that certain common tasks might be more easily programmed if a suitable set of additional commands were provided. For example, the OpenGL Utility Library (**glu**) provides several functions that can be used to render quadric surfaces such as spheres, cones and cylinders. These functions control many aspects of the rendering process, such as the dimensions of the surface, whether texture coordinates should be generated, etc. This functionality can be captured in a more natural way within Tcl scripts with a corresponding widget command corresponding to the surface type to be rendered. Consider for instance the `sphere` widget command described below:

pathName `sphere` *?*`-displaylist` *dlist?* *?*`-normals` *normals?* *?*`-drawstyle` *drawStyle?* *?*`-orientation` *orientation?* *?*`-texture` *texture? radius slices stacks*

Renders a sphere using the GLU facilities for quadrics (refer to the **gluCylinder** function). By default, the rendering is compiled into a new display list whose number is returned as the result of the widget command. If a display list number *dlist* is specified by means of the `-displaylist` option, then that list is used. As a special case, if *dlist* is specified as `none`, the rendering is performed immediately, and no display list is generated or overwritten. The remaining options correspond to rendering styles as implemented by functions **gluQuadricNormals**, **gluQuadricDrawStyle, gluQuadricOrientation** and **gluQuadricTexture**, respectively. The possible option values are strings derived from corresponding symbolic constants. Thus, for instance, `–normals flat` corresponds to calling **gluQuadricNormals** with an argument equal to **GLU_FLAT**.

Figure 2: Display produced by the script of Example 3

The `sphere` command encapsulates all the functionality of the **glu** library procedures for rendering spheres. This is illustrated in the script labeled "Example 3" below which renders a shaded, textured sphere (see Figure 2).

```
package require Tkogl
# Create a checkerboard image
image create photo tmp -width 2 \
   -height 2
image create photo img -width 64 \
   -height 64
tmp put {{white red} {black white}
img copy tmp -zoom 4 4 -to 0 0 64 64

# Create an OpenGL window
pack [OGLwin .gl]

# Configure point of view and texture
.gl eval \
    -matrixmode projection \
    -ortho -2 2 -2 2 -2 2 \
    -matrixmode modelview \
    -rotate 30 1 1 0 \
    -enable lighting \
    -enable light0 \
    -enable depthtest \
    -enable texture2d \
    -texparameter texture2d \
     texturewraps repeat \
    -texparameter texture2d \
     texturewrapt repeat \
    -texparameter texture2d \
     texturemagfilter nearest \
    -texparameter texture2d \
     textureminfilter nearest \
    -texenv textureenv \
     textureenvmode modulate\
    -teximage2d 0 0 img

# Create sphere object
set quadric [.gl sphere -texture yes \
   -drawstyle fill -normals smooth \
    1.8 20 20]

.gl main -clear colorbuffer depthbuffer\
    -call $quadric
```

Example 3: Displays a textured shaded sphere.

Note in Example 3 that the texture image was created by means of *Tk*'s `photo` extensions. The Tkogl package interacts with images created with Tk in order to provide a smooth integration with OpenGL capabilities. Thus, the argument list of OpenGL function **glTexImage2d** was slightly modified in the corresponding Tkogl option command `-teximage2d` so that an image name could be used instead of an array of bytes.

The Tkogl package implements several widget commands that encapsulate capabilities usually accessed by means of the **glu** library such as the rendering of quadric and NURBS surfaces, polygon tesselation, etc. Additional widget commands are also included to implement other useful rendering and modeling extensions not supported by the **glu** library. For example, Tkogl includes the `gencyl` command, which supports the creation of generalized cylinders (i.e., objects obtained by sweeping a two-dimensional shape along a curve in 3D). Figure 3 depicts one of these objects obtained with a demo application included in the distribution.

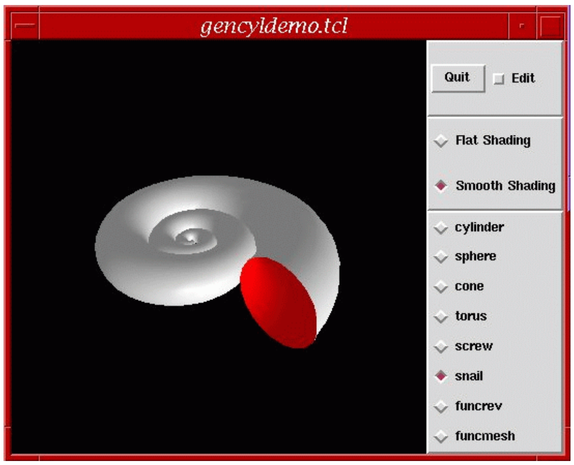

Figure 3: Generalized cylinder object obtained with a demo application**.**

6. Input Events

One of the nicest features of Tk is the simple way user-input events can be handled by means of the `bind` command. The OpenGL standard, on the other hand, offers limited facilities for managing input. These facilities, however, were included in the Tkogl package and can be easily put into use within a Tcl script. For instance, one of the most bothersome difficulty lies in establishing the correspondence between window coordinates and world coordinates. The GLU library provides two functions – **gluProject** and **gluUnProject** – for exactly that purpose. While these functions require a long list of arguments which include the current viewport and transformation matrices, their Tkogl counterparts – the `project` and `unproject` widget commands – only require three arguments representing the three coordinate values of the point to be transformed. Example 4 below demonstrates the use of the `unproject` command in a program for drawing lines. Notice how points which are input by clicking the mouse button are captured with a `bind` command and passed into procedure `newvertex` which computes the corresponding world coordinates. A sample output of this program is shown in Figure 4.

```
package require Tkogl

pack [OGLwin .gl]

proc newvertex { x y } {
    global vertices
    append vertices -vertex \
      " [.gl unproject $x $y 0] "
    eval .gl main -clear colorbuffer\
      -begin lines $vertices -end
}

.gl main -clear colorbuffer

bind .gl <Button-1> {newvertex %x %y}
```

Example 4: A simple line drawing program.

Other OpenGL facilities for handling user input are similarly supported in Tkogl. In particular, object selection and “picking” can be handled in Tkogl through the use of the `select` command. This command takes care of allocating a hit buffer and processing the list of hit objects, returning a single Tcl list which can then be easily parsed within the script.

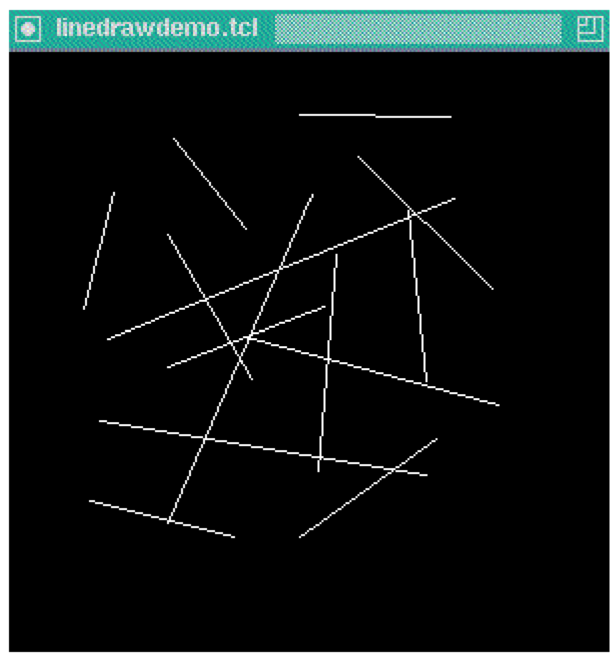

Figure 4: Sample output of the script “Example 4”.

7. Conclusions

We described our implementation of a Tk widget for generating and displaying 3D graphics through the use of the OpenGL API. Our approach, in contrast with similar ports of OpenGL to the Tcl/Tk environment, combines the accessibility of most OpenGL functions through widget commands and options with a repertoire of extensions that enable users to model several objects with compact Tcl scripts.

References

1. J. Neider, T. Davis, M. Woo, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1,* Addison-Wesley, Reading, Massachusetts,1993.
2. Tiger 1.2 ftp site. URL: `ftp://metallica.prakinf.tu-ilmenau.de/pub/PROJECTS/TIGER1.2`
3. Togl home page. URL: `http://www.ssec.wisc.edu/~brianp/Togl`.
4. Mesa home page. URL: `http://www.ssec.wisc.edu/~brianp/Mesa.html`
5. TkOGL home page. URL: `http://aquarius.lcg.ufrj.br/~esperanc/tkogl.html`
6. OpenGL Architecture Review Board, *OpenGL Reference Manual: The Official Reference Document for OpenGL, Release 1*, Addison-Wesley, Reading, Massachusetts, 1992.