A TK OpenGL widget

Claudio Esperança

1 Introduction

OpenGL is becoming a *de facto* standard as an 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, but for one or other reason, they did not match my expectations. As a consequence, I have decided to write my own package, which I hope will serve as a starting point to a more complete Tcl/Tk binding of the OpenGL API.

This project started out before I had a more complete understanding of OpenGL or, perhaps, just because of that. For this reason, my choice of a subset of the OpenGL functionality may not be powerful enough for some, and the way this functionality has been implemented as a series of *tk* widget commands may prove to be a bit too awkward. On the other hand, I included a few features that are not part of the OpenGL standard, but which I felt would increase the usefulness of this package.

The package was tested both with the IBM implementation of OpenGL 1.0 and with a free implementation of the OpenGL API, namely, the Mesa 3-D graphics library version 1.2.5. Both versions were tested on a IBM RS-6000 workstation running AIX v3.2.5. The package may be used as a library or by means of an extended Tcl/Tk windowing shell (i.e., *wish*). The latter is called either `glwish` or `mesawish` according to whether a standard OpenGL library or the Mesa library was used to link it.

2 Creating an OpenGL window

An OpenGL window can be created in a way similar to other Tk widgets by means of the `OGLwin` command, which has the following format:

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

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

- `-width` *width* specifies the width of the window in pixels. Default:300.
- `-width` *width* specifies the height of the window in pixels. Default:300.
- `-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.
- `-doublebuffer` *doubleFlag* specifies whether or not a double buffered visual will be used (true, by default).
- `-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.
- `-alphasize` *alphaSize* specifies that the color buffer should support *alphaSize* bit planes for the alpha component. By default, no bit planes are requested.
- `-accumsize` *accumSize* specifies that the accumulation buffer should support *accumSize* bit planes for each of the red, green and blue components. If *alphaSize* is non-zero, that number of bit planes is also requested for the alpha component of the accumulation buffer. By default, no accumulation buffer is requested.
- `-stencilsize` *stencilSize* specifies the number of bit planes requested for the stencil buffer (zero, by default).

`OGLwin` uses GLX extension subroutine *glXChooseVisual* to allocate a proper X visual for the window. The set of requested capabilities are set by means of the above configuration options. The default behaviour of OGLwin is to try to allocate an RGBA visual with double buffer and a depth buffer of at least 16 bit planes. If a visual with the requested capabilities cannot be allocated, then OGLwin fails, producing a standard Tcl error result.
An OpenGL window is usually created for visualizing a series of graphical objects. Thus, a sensible management of an OpenGL window (such as the one

created by the `OGLwin` command) requires that the these graphical objects be rendered anew every time the window needs to be redrawn, e.g., when the window is resized or when a portion of it which was obscured by another window becomes visible. In order to make this process as painless as possible, an `OGLwin` widget reserves an OpenGL display list containing all rendering commands that are to be executed whenever the window needs to be redrawn. We refer to this list in this document as the *main list*. Also, the widget takes care of issuing a final **glXSwapBuffers** command for double buffered visuals or **glFlush** command for single buffered visuals, so that the effect of the drawing commands contained in the main list are reflected in the window. The contents of the main display list can be redefined by means of the `mainlist` widget command, which has the following format:

pathName `mainlist` ?*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.

For example, a very minimal script that creates a window displaying a line in it can be written as follows:

Example 1: A simple script to display a diagonal line.

```
OGLwin .gl
pack .gl
.gl main -clear colorbuffer -begin line \
    -vertex 0 0 -vertex 1 1 -end
```

Notice that the above script 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 line will be drawn in white over a black background.
Instead of using the default 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 a default display list, the same OpenGL commands can be issued via the `eval` widget command, which has the following syntax:

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

Thus, Example 1 could be rewritten in the following way:

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

```
OGLwin .gl
pack .gl
bind .gl <Expose> {
    .gl eval -clear colorbuffer -begin line -vertex 0 0 \
        -vertex 1 1 -end
}
```

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.

3 OGLwin widget commands

pathName `configure` ?*option*? ?*value*? ?*option value ...*?

Query or modify the configuration options of the widget. If no *option* is specified, returns a list describing all of the available options for *pathName*. If *option* is specified with no *value*, then the command returns a list describing the one mentioned option. If one or more *option-value* pairs is specified, then the command modifies the given widget option(s). Note: the `-context` option can only be specified at the time the widget was created and cannot be modified afterwards by the `configure` widget command.

pathName `cylinder` ?`-normals` *normals*? ?`drawstyle` *drawStyle*? ?`-orientation` *orientation*? ?`-texture` *texture*? *baseRadius topRadius height slices stacks*

Renders a cylinder using the GLU facilities for quadrics (refer to the **gluCylinder** function). The rendering is compiled into a display list whose number is returned as the result of the widget command. The options correspond to rendering styles as implemented by functions **gluQuadricNormals**, **gluQuadricDrawStyle**, **gluQuadricOrientation** and **gluQuadricTexture**, respectively. The possible option values are lowercase strings derived from corresponding symbolic constants. Thus, for instance, `-normals flat` corresponds to calling **gluQuadricNormals** with an argument equal to **GLU_FLAT**.

pathName `deletelist` *listNumber*

Deallocates the display list specified by *listNumber*.

pathName `disk` ?`-normals` *normals*? ?`drawstyle` *drawStyle*? ?`-orientation` *orientation*? ?`-texture` *texture*? *innerRadius outerRadius slices loops*

Renders a disk using the GLU facilities for quadrics (refer to the **gluDisk** function). The rendering is compiled into a display list whose number is returned as the result of the widget command. The options are the same as those of the `cylinder` command.

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

Sends the OpenGL commands defined by the given *options* directly to the OpenGL engine. Note that the effect of the these commands will only be visible after the window is refreshed. This happens automatically after an *Expose* event is handled by the widget, but can be forced programatically by issuing a `redraw` widget command.

pathName `mainlist` ?*option ... option*?

Creates a display list containing the OpenGL commands defined by the given *options* (for a description of these, refer to the next section) and sets up the widget to call that list every time an Expose or Configure event is sent to the corresponding window. By default, this main list contains no OpenGL commands, i.e., nothing is drawn.

pathName `newlist` ?*listNum*? ?*option ... option*?

Defines a new display list containing the OpenGL commands specified by the given *options* (for a description of these, refer to the next section). If *listNo* is specified, the contents of the corresponding display list is redefined. Otherwise, a new display list is allocated. The number corresponding to the redefined or newly allocated display list is returned, so that it can be invoked by means of the `-call` option. The `newlist` widget command combines the functionality of OpenGL subroutines **glGenLists**, **glNewList** and **glEndList**.

pathName `nurbssurface` `-uknots` *knot ?knot ...?* `-vknots` *knot ?knot ...?* `-controlpoints` *coord ?coord ...?* ?`-type` *type*? ?`-uorder` *order*? ?`-vorder` *order*? ?`-samplingtolerance` *tol*? ?`-displaymode` *mode*? ?`-culling` *cull*?

Implements a simple interface to **gluNurbsSurface** and other related GLU functions. It renders a nurbs surface into a display list whose number is returned as the result of the command. The only mandatory options are `-uknots`/`-vknots` which specify the sequence of knots in the u and v directions, respectively, and `-controlpoints`, which is followed by the coordinates of the control points. The remaining options control other parameters of **gluNurbsSurface** and rendering parameters usually set with **gluNurbsProperty**. The default values for these are as follows. `-type map2vertex3` (i.e., GL_MAP2_VERTEX_3); `-uorder 4` and `-vorder 4` (cubic polynomials); `-samplingtolerance 50` (pixels); `-displaymode`: fill (i.e., GLU_FILL); `-culling f` (i.e. GL_FALSE). Notice that the command takes care of computing remaining parameters such as *uStride* and *vStride* by counting the number of knot values and control point coordinates given.

pathName `partialdisk` ?`-normals` *normals*? ?`drawstyle` *drawStyle*? ?`-orientation` *orientation*? ?`-texture` *texture*? *innerRadius outerRadius slices loops startAngle sweepAngle*

Renders a partial disk using the GLU facilities for quadrics (refer to the **gluPartialDisk** function). The rendering is compiled into a display list whose number is returned as the result of the widget command. The options are the same as those of the `cylinder` command.

pathName `project` *worldX worldY worldZ*

This command provides access to the functionality of the **gluProject** OpenGL utility function. *worldX*, *worldY* and *worldZ* are the world coordinates of a point and, as a result, a list containing the corresponding three window coordinates is returned. The command takes care of retrieving from the the other arguments of **gluProject**, such as the viewport, projection and modelview matrices. It also takes care of the correction of the value for the y coordinate due to the fact that the y axis in OpenGL is defined to run from the bottom to the top of the screen, while the window coordinate system in X defines y to run from top to bottom.

pathName `redraw`

Forces the window to be redisplayed. This involves calling the main display list (see the `mainlist` widget command) and flushing all pending commands by calling either **glXSwapBuffers** or **glFlush**, according to whether the visual for the window is double or single buffered.

pathName `select` *hitBufferSize* ?*option ... option*?

Evaluates the OpenGL commands corresponding to the given *options* in selection mode and returns the contents of the hit buffer as a Tcl list. In other words, this command is equivalent to calling **glRenderMode (GL_SELECT)**, issuing the commands corresponding to each *option* and then calling **glRenderMode (GL_RENDER)**. A hit buffer containing *hitBufferSize* words is dynamically allocated to store the contents of the hit buffer for the duration of the command.

The format of the returned Tcl list mimics that of the hit buffer. Each hit record corresponds to one element of the list. The first element of the hit record is the number of names on the name stack; the second (third) element is the minimum (maximum) z value of all vertices of the primitives that intersected the viewing volume since the last recorded hit in floating point format; and the remaining elements are the contents of the name stack at the time of the hit, with the bottommost element first. See the discussion in Chapter 12 of the *OpenGL Programming Guide* for more details.

pathName `tesselate` *x y z ...* ?`-contour` *x y z ...*?

Renders a complex polygon using the facilities of the GLU tesselator. The vertices of the polygon are specified by their coordinates *x, y* and *z*. The `-contour` option can be used to specify the different polygon boundaries (i.e, "holes"). The rendering is compiled into a new display list whose number is returned by the widget command.

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

Renders a sphere using the GLU facilities for quadrics (refer to the **gluSphere** function). The rendering is compiled into a display list whose number is returned as the result of the widget command. The options are the same as those of the `cylinder` command.

pathName `unproject` *windowX windowY windowZ*

This command provides access to the functionality of the **gluUnProject** OpenGL utility function. *windowX*, *windowY* and *windowZ* are window coordinates and the result is a list with the corresponding world coordinates. The handling of the remaining values necessary to compute this operation is done in the same way described for the `project` command.

4 Summary of OpenGL option commands

Many OGLwin widget commands (e.g., `eval`, `newlist`, `mainlist`, `select`) 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 OpenGL command. The string corresponding to a given OpenGL procedure is the name of that procedure with all letters in lower case and stripped of its **'gl'** prefix and of eventual data type suffix. Thus, for instance, procedure

glMatrixMode corresponds to option `-matrixmode`, 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 an all lowercase string with the same spelling as that of the equivalent constant, except that the **'GL'** prefix is dropped as well as any underscore ('_') characters (e.g., **GL_DEPTH_TEST** becomes `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 (**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 specific case of procedure **glClear**, which requires a bit mask argument, 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, all *args* are *or*'ed together so that, for instance, **glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)** becomes `-clear colorbuffer depthbuffer`.

Not all 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. We list below the syntax of all OpenGL functions currently implemented and the syntax of their associated option commands. The full description of each OpenGL function can be found in the OpenGL reference manual.

glAccum	`-accum` *operation value*
glAlphaFunc	`-alphafunc` *function reference*
glBegin	`-begin` *primitive*
glBlendFunc	`-blendfunc` *sFactor dFactor*
glCall	`-call` *displayListNo*
glClear	`-clear` ?*bitMaskConst ... bitMaskConst*?
glClearAccum	`-clearaccum` *red green blue alpha*
glClearColor	`-clearcolor` *red green blue* ?*alpha*
glClearDepth	`-cleardepth` *depth*
glClearStencil	`-clearstencil` *stencil*
glColor	`-color` *red green blue* ?*alpha*?
glColorMask	`-colormask` *red green blue alpha*
glColorMaterial	`-colormaterial` *face mode*

glCopyPixels	`-copypixels` *x y width height*
glDepthMask	`-depthmask` *flag*
glDisable	`-disable` *capability*
glDrawPixels	`-drawpixels` *photoImageName*
glEnable	`-enable` *capability*
glEnd	`-end`
glEvalCoord1	`-evalcoord1` *u*
glEvalCoord2	`-evalcoord2` *u v*
glEvalMesh1	`-evalmesh1` *mode i1 i2*
glEvalMesh2	`-evalmesh2` *mode i1 i2 j1 j2*
glFog	`-fog` *param paramValue ?paramValue ...?*
glFrontFace	`-frontface` *mode*
glFrustum	`-frustum` *left right bottom top near far*
glInitNames	`-initnames`
glLight	`-light` *light parameterName parameter ?parameter ...?*
glLightModel	`-lightmodel` *parameterName parameter ?parameter ...?*
glLineStipple	`-linestipple` *factor pattern*
glLineWidth	`-linewidth` *width*
glLoadIdentity	`-loadidentity`
glLoadMatrix	`-loadmatrix` $m_{0,0}$ $m_{1,0}$ $m_{2,0}$ $m_{3,0}$ $m_{0,1}$ $m_{1,1}$ $m_{2,1}$ $m_{3,1}$ $m_{0,2}$ $m_{1,2}$ $m_{2,2}$ $m_{3,2}$ $m_{0,3}$ $m_{1,3}$ $m_{2,3}$ $m_{3,3}$
glLoadName	`-loadname` *name*

gluLookAt	`-lookat` *eyeX eyeY eyeZ centerX centerY centerZ upX upY upZ*
glMap1	`-map1` *target u1 u2 stride order pointCoord ?pointCoord ...?*
glMap2	`-map2` *target u1 u2 uStride uOrder v1 v2 vStride vOrder pointCoord ?pointCoord ...?*
glMapGrid1	`-mapgrid1` *uN u1 u2*
glMapGrid2	`-mapgrid2` *uN u1 u2 vN v1 v2*
glMaterial	`-material` *face paramName param ?param param*
glMatrixMode	`-matrixmode` *mode*
glMultMatrix	`-modelview` *mode*
glNormal	`-normal` *x y z*
glOrtho	`-ortho` *left right bottom top near far*
gluPerspective	`-perspective` *fieldOfViewY aspectRatio zNear zFar*
gluPickMatrix	`-pickmatrix` *x y width height*
glPixelTransfer	`-pixeltransfer` *paramName param*
glPixelZoom	`-pixelzoom` *xFactor yFactor*
glPointSize	`-pointsize` *size*
glPolygonMode	`-polygonmode` *face mode*
glPopMatrix	`-popmatrix`
glPopName	`-popname`
glPushMatrix	`-pushmatrix` $m_{0,0}$ $m_{1,0}$ $m_{2,0}$ $m_{3,0}$ $m_{0,1}$ $m_{1,1}$ $m_{2,1}$ $m_{3,1}$ $m_{0,2}$ $m_{1,2}$ $m_{2,2}$ $m_{3,2}$ $m_{0,3}$ $m_{1,3}$ $m_{2,3}$ $m_{3,3}$
glPushName	`-pushname` *name*
glRasterPos	`-rasterpos` *x y ?z? ?alpha?*

glReadBuffer	`-readbuffer` *mode*
glReadPixels	`-readpixels` *x y photoImageName*
glRotate	`-rotate` *angle x y z*
glScale	`-scale` *x y z*
glScissor	`-scissor` *x y width height*
glShadeModel	`-shademodel` *mode*
glStencilFunc	`-stencilfunc` *function reference mask*
glStencilMask	`-stencilmask` *mask*
glStencilOp	`-stencilop` *fail zFail zPass*
glTexCoord	`-texcoord` *s ?t? ?q? ?r?*
glTexEnv	`-texenv` *target paramName param ?param ...?*
glTexGen	`-texgen` *coordinate paramName param ?param ...?*
glTexImage1D	`-teximage1d` *level border photoImageName*
glTexImage2D	`-teximage2d` *level border photoImageName*
glTexParameter	`-texparameter` *target paramName param ?param ...?*
glTranslate	`-translate` *x y z*
glVertex	`-vertex` *x y ?z? ?alpha?*

5 OGLwin extensions

The OGLwin widget provides a mechanism for incorporating user-defined extension commands written in "C". This mechanism is implemented by means of procedure **RegisterTkOGLExtension** with the following syntax:

int **RegisterTkOGLExtension** (Tcl_Interp* *interp*,

char* *extname*,
TkOGLExtProc* *extproc*)

where

interp is a pointer to the main Tcl interpreter.

extname is a string that will designate the extension widget command.

extproc is a pointer to a procedure which implements the extension. This procedure should have the following syntax:

int **MyExtensionProc** (Tcl_Interp* *interp*,
int *argc*,
char ** *argv*)

In order to register a given extension to the OGLwin widget, **RegisterTkOGLExtension** must be called in the initialization code for the application, just after calling **TkOGL_Init**, the OGLwin initialization routine. As an example, consider the initialization code used to build the *glwish* and *mesawish* applications (see file `tkAppInit.c` in the distribution sources):

```
int Tcl_AppInit(Tcl_Interp *interp)
{
   Tk_Window main;
   main = Tk_MainWindow(interp);

   if (Tcl_Init(interp) == TCL_ERROR) return TCL_ERROR;
   if (Tk_Init(interp) == TCL_ERROR) return TCL_ERROR;
   if (TkOGL_Init(interp, main) == TCL_ERROR) return TCL_ERROR;

   if (RegisterTkOGLExtension (interp, "gencyl",
                 GenericCylinder) != TCL_OK) return TCL_ERROR;

   tcl_RcFileName = "~/.wishrc";
   return TCL_OK;
}
```

This code defines a default extension built into *glwish* (and *mesawish*): the generic cylinder extension (`gencyl`), which is described in the next section.

6 The `gencyl` extension

The generic cylinder extension provides a relatively straightforward way of rendering arbitrary surfaces. The surface is obtained by sweeping a curve (termed a *cross-section* shape) along a path in 3D space. Initially, the cross-section shape is a square of side length equal to 2 centered at the origin and lying over the x-y plane. The movement of the cross-section is controlled by applying affine linear transformations (e.g., translation, rotation, scale) to the cross-section shape. For each two consecutive positions of the cross-section, triangles are rendered joining corresponding vertices and edges.
The syntax of the `gencyl` extension is:

> *pathName* `gencyl` *option* ?*option ...*?
>
> where *pathName* is the name of the OGLwin widget and *option* can be one of the following:
>
> `-cross` *x y z* ?*x y z ...*?
>
> Replaces the current cross-section shape with the polygon defined by the vertices with the given coordinates x, y, z.
>
> `-identity`
>
> Replaces the current transformation matrix with the identity matrix.
>
> `-plot`
>
> Plots a new cross-section, i.e., applies the currently defined transformation matrix to the cross-section shape. If it is not the first cross-section to be generated, draws triangles linking vertices of this cross-section with corresponding vertices of the previous cross-section.
>
> `-polygon` *radius nSides*
>
> Replaces the current cross-section shape with a regular polygon of radius *radius* and *nSides* sides. The polygon is centered at the origin and lies over the x-y plane.
>
> `-rotate` *angle x y z*
>
> Multiplies the current transformation matrix by a matrix corresponding to the rotation of *angle* degrees around the

vector defined by $x\ y\ z$. This is similar to the effect of procedure **glRotate**.

`-scale` $x\ y\ z$

Multiplies the current transformation matrix by a general scaling matrix defined by parameters $x\ y\ z$. This is similar to the effect of procedure **glScale**.

`-shade` *shadeModel*

Defines the type of vertex normals generated for the triangles. If *shadeModel* is `flat`, the same normal is generated for all three vertices, namely, the triangle normal. If *shadeModel* is `smooth` (the default), vertex normals are averaged over all incident triangles.

`-stitch` *stitchStyle*

Defines the overall topology of the generated surface. If *stitchStyle* is `loops` (the default), cross-sections are generated as closed curves, i.e., the first and last vertex of each cross-section are linked together. If *stitchStyle* is `ends`, the first and the last generated cross-sections are linked together. If *stitchStyle* is `both` the effect of `loops` and `ends` is combined. Finally, if *stitchStyle* is `none`, no linking takes place. Generally speaking, `both` corresponds to the topology of a torus, while `none` corresponds to the topology of a disk.

`-texgen` *minS maxS minT maxT*

Specifies the generation of texture coordinate values. *minS* and *maxS* refer to the range of values for texture coordinate s, whereas *minT* and *maxT* refer to the range of values for texture coordinate t. Texture coordinates are generated so as to have the s coordinate varying along each cross-section, while the t coordinate is associated with the path each point of the cross-section shape describes as each cross-section is plotted.

`-translate` $x\ y\ z$

Multiplies the current transformation matrix by the translation matrix defined by $x\ y\ z$.

Notice that `gencyl` does not use the model view matrix to compute transformations. In fact, the only OpenGL commands emitted by `gencyl` are **glNormal**, **glVertex** and *glTexCoord*.
The result string returned by the `gencyl` command is a property list of the form

> {`displaylist` *d*} {`min` *minX minY minZ*} {`max` *maxX maxY maxZ*}

where

d is the number of the display list generated. This number can be used in conjunction with the `-call` option command to display the surface.

minX minY minZ and *maxX maxY maxZ* define the coordinates of a bounding box for all the generated vertices. These can be used in order to calculate apropriate viewing parameters.

As an example, consider the following script that renders a 8-sided approximation of a circular cylinder of unit height and radius:

```
OGLwin .gl
pack .gl
set l [.gl gencyl -polygon 1 8 -plot \
                  -translate 0 0 1 -plot]
set d [lindex [lindex $l 0] 1]
.gl eval -matrixmode projection \
         -loadidentity \
         -perspective 30 1 1 10\
         -matrixmode modelview \
         -loadidentity \
         -lookat 5 5 5 0 0 0 0 0 1\
         -enable light0 \
         -enable lighting
.gl main -clear colorbuffer -call $d
```