PyOpenGL Manual

by Tarn Weisner Burton and Mike C. Fletcher

PyOpenGL Manual

by Tarn Weisner Burton and Mike C. Fletcher

PyOpenGL [http://pyopengl.sourceforge.net] is a cross-platform open source Python binding to OpenGL [http://www.opengl.org] which provides a standard 2D and 3D graphics API. PyOpenGL also supports GLU, GLE [http://linas.org/gle], GLUT [http://reality.sgi.com/mjk/glut3], WGL, and Togl.

Table of Contents

Chapter 1. Introduction

PyOpenGL is a cross-platform open source Python binding to the standard OpenGL API providing 2-D and 3-D graphic drawing. PyOpenGL supports the GL, GLU, GLE, and GLUT libraries. The library can be used with almost any Python windowing library which can provide an OpenGL context. Currently supported libraries include:

- FXPy [http://fxpy.sourceforge.net]
- pygame [http://pygame.seul.org]
- Tkinter
- win32gui
- wxPython [http://www.wxpython.org]

All PyOpenGL modules are contained in the package `OpenGL`. For the sake of brevity, the top level package name is omitted when referring to modules, variables and functions in this document. For example, instead of `OpenGL.GL` this document uses the label `GL`.

Interfaces To Other Libraries

PyOpenGL currently requires either Python 1.5.2 or greater because of the use of the Distutils setup mechanisms. For versions of Python earlier than 1.5.2, the PyOpenGL 1.5.5 release (available from the PyOpenGL project download page) may be usable.

PyOpenGL includes support for the Tkinter widget, Togl [http://togl.sourceforge.net]. Togl 1.5 is currently shipped with PyOpenGL.

PyOpenGL's setup will attempt to enable certain functionality (primarily array support) if it can import the Python Numeric Extensions available at: http://numpy.sourceforge.net [http://numpy.sourceforge.net]. Most people interested in using PyOpenGL for nontrivial purposes will want to install this package before installing PyOpenGL.

PyOpenGL does not directly interface with the Python Imaging Library, or PIL [http://www.pythonware.com/products/pil/index.htm], but PIL images can be used within PyOpenGL easily. Demonstrations of this are available within the PyOpenGL package.

Installation

Before installing PyOpenGL be sure to uninstall or remove any previous versions. Failure to do this may result in irratic behavior of Python scripts which use PyOpenGL.

There are two main methods of installing PyOpenGL. On Win32 systems executable installers are available, for other platforms you will have to build PyOpenGL from the source. To do so you will need Distutils 1.0.2pre or higher. Python 2.1 users should already have this installed. To get the latest version of see the distutils-sig [http://www.python.org/sigs/distutils-sig] page.

You will also need the GLUT library. On UNIX systems this will probably already be present, for other systems see the GLUT homepage [http://reality.sgi.com/mjk/glut3/glut3.html].

As discussed in the previous section PyOpenGL can interface to the Numeric library [http://numpy.sourceforge.net]. To enable this support Numeric and its associated headers must be installed at build time.

To build PyOpenGL type

```
$ python setup.py build
```

at the shell prompt.

Note

SWIG [http://swig.sourceforge.net] is not needed to build PyOpenGL, but if SWIG is present on your system then the **build_w** setup command can be used to rebuild the generated wrappers. Currently only SWIG 1.3a5 is supported.

After building, to install type

```
$ python setup.py install
```

This will also execute the build command if required. To find out what various options are accepted by the setup script one can also

```
$ python setup.py build -help
```

or

```
$ python setup.py install -help
```

Important

The setup script will try to build Togl if the `Tkinter` module can be imported. Togl version 1.5 is shipped with this version of PyOpenGL. The building of Togl can be bypassed by changing a setting in a platform specific configuration file. The name of this file is based on `sys.platform`. For instance, on Linux platforms it will be called `config/linux.cfg`. For more information on this file see the Porting Section.

Bug Reporting

Please submit bug reports using the bug tracker at the PyOpenGL project page [http://sourceforge.net/projects/pyopengl]. There are also support and feature request trackers at this page.

Note

PyOpenGL development, usage and bugs are discussed on the PyOpenGL mailing list [http://groups.yahoo.com/group/PyOpenGL].

Before you send any bug reports, try the demos and if they show the same problem. When submitting a bug report always include:

1. complete traceback

2. the HTML output of the script `OpenGL/scripts/info.py`

If you have problems compiling PyOpenGL, also include:

1. description of your system and your compiler
2. output of following python statements

```
import sys, distutils
print sys.platform
print sys.version
print distutils.__version__
```

Chapter 2. Upgrading From PyOpenGL 1.5

This section presents various notes regarding upgrading from PyOpenGL 1.5 to the newer PyOpenGL 2 binding. This complete rewrite of the PyOpenGL system has a slightly different API from the original PyOpenGL binding. Most of these changes are minor, with the obvious exception of the error handling mechanism (which is always strict in PyOpenGL 2, while only optionally strict in PyOpenGL 1.5).

These upgrading notes are by no means a complete discussion of the PyOpenGL 2 binding and are only meant as a guide to the most likely issues when upgrading.

Error Handling

PyOpenGL 2 always uses "strict" OpenGL operation, which is closer to the operation of Python itself, i.e. errors are raised as exceptions, rather than silently being passed back to the user. `glGetError` is not available as it would never return anything save `GL_NO_ERROR`.

To convert, you will want to search for each instance of `glGetError` and introduce the appropriate exception handling mechanisms. The exceptions raised are:

- `GL.GLerror` by all modules except `WGL`
- `GLU.GLUerror` by some `GLU` functions. Note that `GLU` can also throw `GL.GLerror`
- `WindowsError`/`SystemError` by `WGL`

GL

`glCleanRotMat` – Not Available. This utility function attempted to recover from precision errors encountered with multiple matrix rotations. This seems like an inelegant approach to the situation. Consider using quaternion addition (see `OpenGL/quaternion.py`) or single rotation calls.

`glSaveTIFF`, `glSaveJPEG`, `glSavePPM` – Not Available. These functions attempted to save a file in a particular format directly from the OpenGL context. They largely duplicated effort both from the Python Imaging Library (PIL) (image handling) and python itself (file handling). Since `glReadPixels` is fully functional under PyOpenGL, these functions were not provided. See `OpenGLContext/tests/saveimage.py` for demonstration code showing how to save an image using PIL

`glTetrahedra` – Not Available. This does not seem like something that needs to be in the core library.

`glTriangleNormals`, `glIndexedGeomDSPL`, `glTrianglesWithNormals`, `glLines`, `glPoints` – Not Available . These functions provide somewhat higher level rendering mechanisms for a small subset of geometry types. They do not really seem to be something that belongs in the core library. Although they may be implemented in a utility library at a later date for legacy purposes, at the present time they are not supported.

`glGetError`. See Error Handling above for rationale. You will need to either define a nop function `glGetError`, or remove all instances of `glGetError`.

`glGetBooleanv`, `glGetFloatv`, `glGetDoublev`, `glGetIntegerv` – Changed Behavior. The `glGetX` family of functions no longer chooses a single default type for each argument, instead, it returns the type appropriate to the explicitly named function. For instance,

```
>>> glGetInteger( GL_MODELVIEW_MATRIX )
[[1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]]
```

You will need to search for each `glGetX` call and determine if the call really intended to use the function named, or whether it was relying on the looser semantics of the original functions.

`glXPointer` Family – Changed Behavior. There are now two major variants of these functions. The first variant, spelled with the appropriate type decoration `{d|f|i|b|ub}` takes a two- dimensional python (or Numeric) array as its single argument:

```
glVertexPointerd([[0.0, 0.0, 0.0], # note need for 2-d array!
                  [5.0, 0.0, 0.0], # Changed with PyOpenGL 2
                  [5.0, 5.0, 0.0],
                  [0.0, 5.0, 0.0]])
```

The second variant allows you to specify a string argument, as well as the stride argument, and matches the semantics of the underlying OpenGL function.

```
glVertexPointer(3, GL_DOUBLE, 0, vertices.tostring())
```

Note

`glInterleavedArrays` is not part of this family and requires a string, not an array argument

`glVertex` - Changed Behavior. `glVertex` no longer supports passing in arrays of points, you will need to convert your code to using `glVertexPointer{d|f}`.

`glColor2Vertexd`, `glColor2Vertex` – Not Available. Like `glVertex` this functionally should be replaced using `glVertexPointer` and `glColorPointer`.

Function Aliases. PyOpenGL 1.5 provided a number of convenience names which were aliases for decorated function names. To preserve code compatibility, many of these functions are now available within PyOpenGL 2.

Function Aliases

`glGetBooleanv` aliases	`glGetBoolean`
`glGetDoublev` aliases	`glGetDouble`
`glGetIntegerv` aliases	`glGetInteger`
`glColor`	
`glColor3`	
`glColor4`	
`glEvalCoord`	

`glEvalCoord1`
`glEvalCoord2`

`glFogfv` aliases `glFog`
`glIndexd` aliases `glIndex`
`glLightfv` aliases `glLight`
`glLightModelfv` aliases `glLightModel`
`glMaterialfv` aliases `glMaterial`
`glNormal`
`glNormal3`
`glNormal4`
`glRasterPos`
`glRasterPos2`
`glRasterPos3`
`glRasterPos4`

`glRotated` aliases `glRotate`
`glScaled` aliases `glScale`
`glTexCoord`
`glTexCoord1`
`glTexCoord2`
`glTexCoord3`
`glTexCoord4`

`glTexGendv` aliases `glTexGen`
`glTexParameterfv` aliases `glTexParameter`
`glTranslated` aliases `glTranslate`
`glVertexd` aliases `glVertex`

GLUT

Callback Registration – Changed Behavior. The method for registering callbacks in `GLUT` has changed. Instead of calling:

```
glutSetReshapeFuncCallback(OnResize)
glutReshapeFunc()
```

You will call:

```
glutReshapeFunc(OnResize)
```

If you need to unbind the callback then just use `None` instead of a callback function in the callback registration function, i.e.

```
glutReshapeFunc(None)
```

If you need your code to run under both versions, you can catch a raised `NameError` on `glutSetReshapeFuncCallback` and call `glutReshapeFunc`. See `OpenGLContext.GLUTContext` for a demonstration.

`glutInit` – Changed Behavior. This method initializes the GLUT windowing system taking a non-string sequence of strings such as available in sys.argv. PyOpenGL would allow you to use a single string as the argument. `glutInit` returns the arguments not used by GLUT as a list of strings.

Chapter 3. User's Guide

Most of the functions which appear in PyOpenGL 2 are identical in calling method and functionality to that of the appropriate specification. There are a few exceptions because of the differences between C and Python. Most of these exceptions are due to the difference between C and Python in the way that they access arrays. For example, a C function like this:

```
void foo(int count, const int *args);
```

will have the Python binding:

```
foo(args) -> None
```

Also C functions which write array data to a function argument like:

```
void bar(int args[4]);
```

will have the Python binding:

```
bar() -> args[]
```

The following sections will document changes other than simple changes like the above examples. When in doubt about the Python binding for a specific function one can always check the docstring explicitly. For instance, to check the Python binding of the C function `glDeleteTextures` which has the C binding

```
void glDeleteTextures(GLsizei n, const GLuint *textures);
```

one could do the following from the Python prompt

```
>>> from OpenGL.GL import *
>>> glDeleteTextures.__doc__
'glDeleteTextures(textures[]) -> None'
```

Notice that the parameter *`n`* is missing from the Python prototype since this can automatically be determined from the parameter *`textures`*. Also note that the notation *`textures[]`* is used in docstrings to indicate that the *`textures`* parameter should be a Python sequence. If the *`textures`* parameter was required to be a two dimensional array than this would be indicated by *`textures[][]`*. If one of the array dimensions is expected to have a fixed size than this will be indicated by a number enclosed in the brackets. For instance, `glePolyCone` of the GLE module has the prototype

```
glePolyCone(point_array[][3], color_array[][3], radius_array[]) -> None
```

Of course in many Python IDEs the first line of the function docstring will be displayed in a tool-tip when you type the function name in a Python script. Another way to see all the functions, attributes, or docstrings of a specific module is to use the pydoc utility. Python 2.1 includes this utility. For other versions of Python see Ka-Ping Lee's Python Things [http://web.lfw.org/python].

GL

OpenGL Library; API versions 1.0 and 1.1

Selection and Feedback buffers

Normally in OpenGL to use a selection buffer one would do the following:

```
GLuint buffer[SIZE];
glSelectBuffer(SIZE, buffer);
glRenderMode(GL_SELECT);
/* draw some stuff */
GLint count = glRenderMode(GL_RENDER);
/* parse the selection buffer */
```

In Python this accomplished like this:

```
glSelectBuffer(SIZE) # allocate a selection buffer of SIZE elements
glRenderMode(GL_SELECT)
# draw some stuff
buffer = glRenderMode(GL_RENDER)
for hit_record in buffer:
    min_depth, max_depth, names = hit_record
    # do something with the record
```

Feedback buffers are used in the same way except that each item in the buffer is tuple `(token, value)`, where value is either a passthrough token or a list of vertices. Note that if `glRenderMode` returns a buffer than it also resets OpenGL's pointer for the corresponding buffer. This means that the buffer returned by `glRenderMode` is independent of future calls to `glRenderMode`, i.e. it will not be overwritten by any such future calls. This makes the returned buffer somewhat thread-safe. It also means that every call to `glRenderMode(GL_SELECT | GL_FEEDBACK)` needs to preceded by a call to `glSelectBuffer` or `glFeedbackBuffer` first, i.e. the following code will not work

```
glSelectBuffer(SIZE) # allocate a selection buffer of SIZE elements
glRenderMode(GL_SELECT)
# draw some stuff
buffer = glRenderMode(GL_RENDER)
# do another selection
glRenderMode(GL_SELECT)
# draw some stuff
buffer = glRenderMode(GL_RENDER)
```

Instead one must

```
glSelectBuffer(SIZE) # allocate a selection buffer of SIZE elements
glRenderMode(GL_SELECT)
# draw some stuff
buffer = glRenderMode(GL_RENDER)
# do another selection
glSelectBuffer(SIZE) allocate a new selection buffer
glRenderMode(GL_SELECT)
# draw some stuff
buffer = glRenderMode(GL_RENDER)
```

In previous versions of PyOpenGL in order to perform selection one would use the `glSelectWithCallback` function. This function is still available, but the only thing that it does in addition to the method described above is setup the projection matrix using `gluPickMatrix`.

Array Routines

Each call which sets an array pointer, such as `glVertexPointer`, will have many variants. First there will a function which is identical that of the specification. For the *`pointer`* argument one should pass a string. Also note that the stride values are used.

Next there will a set of functions named

`glXPointer{ub|b|us|s|ui|i|f|d}`

These will usually take as a single argument a multidimensional array of values. The type argument is controlled by the suffix of the function (`ub` is unsigned byte, etc.) All other arguments are derived from the dimensions of the array.

So for `glColorPointer` we have:

```
glColorPointer(size, type, stride, pointer) -> None
glColorPointerub(pointer[][]) -> None
glColorPointerb(pointer[][]) -> None
glColorPointerus(pointer[][]) -> None
glColorPointers(pointer[][]) -> None
glColorPointerui(pointer[][]) -> None
glColorPointeri(pointer[][]) -> None
glColorPointerf(pointer[][]) -> None
glColorPointerd(pointer[][]) -> None
```

This same decoration strategy is used for other array functions besides `glXPointer`. For instance, `glDrawElements` has the Python binding:

```
glDrawElements(mode, count, type, indices) -> None
```

where *`indices`* is expected to be a string. There are also the decorated bindings

```
glDrawElementsub(mode, indices[]) -> None
glDrawElementsus(mode, indices[]) -> None
glDrawElementsui(mode, indices[]) -> None
```

where "indices" is now a single dimensional array.

When calling a `glColorPointer`, `glVertexPointer`, etc. Python needs to allocate memory to store the values that OpenGL needs. This memory is reference counted and takes into account function calls like `glPushClientAttrib` and `glPopClientAttrib`. To force this memory to be released one just need to make a call `glColorPointerub(None)`.

Currently `glPushClientAttrib` will always set the `GL_CLIENT_VERTEX_ARRAY_BIT` flag as `glPopClientAttrib` has no way of knowing that flag was set and the state of the flag is needed to know whether or not to decrement the pointer locks on the allocated memory. This may change in the future. That said, surrounding array use `glPushClientAttrib`/`glPopClientAttrib` is a good way to force the release of any allocated memory, but make sure that all calls to `glXPointer`, etc. are within the `ClientAttrib` block if you chose to use this scheme.

Note

Since all the memory allocation is automatic there is no need for `glGetPointerv` function, so it is excluded.

Note

The function `glInterleavedArrays` is also present, but it does not have the variants that the others do (i.e., no `glInterleavedArraysf`)

Warning

If you write an extension module which makes calls to the pointer functions and expects to interact with PyOpenGL safely it should push the client state (from within the extension) before and after array use.

Image Routines

`glDrawPixels` and the other image/texturing functions have much the same decoration scheme as the array functions. For `glDrawPixels` there is the standard function which expects a string as the pixel data:

```
glDrawPixels(width, height, format, type, pixels) -> None
```

This function will respect the parameters set by `glPixelStore{i|f}`. There is also a collection of variants which take a multidimensional array as the data source and set `glPixelStore{i|f}` automatically. For example:

```
glDrawPixelsub(format, pixels) -> None
```

Notice that *`width`* and *`height`* are inferred from the pixel data and the type is GLubyte.

Error Handling

In OpenGL the current error status is retrieved with the function `glGetError`. This function returns an error code and clears the current error status at the same time. On some distributed systems multiple errors may be generated at a time. In that case it is necessary to call `glGetError` multiple times.

PyOpenGL does not expose the `glGetError`. Instead it automatically checks the error status and throws an instance of `GL.GLerror` on an error. This is a more "Pythonic" approach. `GL.GLerror` is a subclass of `EnvironmentError` and has a `errno` and `msg` like `EnvironmentError`. In the case of multiple errors the `errno` attribute will be set to a tuple of error numbers and `msg` will be a concatenation of the error messages for each error.

GLU

GL Utility Library; API versions 1.0, 1.1, 1.2 and 1.3

For the most part the Python bindings of the GLU module are identical to that of the GLU specification. The first main difference is in the callback implementation of `GLUnurbs`, `GLUquadric` and `GLUtesselator`. Callbacks can be set with the appropriate function, for instance

```
def begin(type):
    print 'begin', type

tess = gluNewTess()
gluTessCallback(tess, GLU_TESS_BEGIN, begin)
```

except that the error callbacks `GLU_TESS_ERROR`, `GLU_TESS_ERROR_DATA`, or `GLU_ERROR` cannot be set. Instead a `GLU.GLUerror` exception is thrown on an error. The value of the exception will be a 2-tuple except when the error is a tessellation error, in which case the value will be set to `(error_code, error_string, polygon_data)`.

Some other GLU functions can also throw a `GLUerror`. These are primarily functions which would normally return an error code. For instance, `gluScaleImage`

```
try:
    dataout = gluScaleImage(format, widthin, heightin, typein,
                            datain, widthout, heightout, typeout)
except GLUerror:
    pass
```

The last major difference between the `GLU` module and the `GLU` specification is that it also implements the variant decoration scheme as discussed previously for the GL module. This is implemented for the image functions: `gluScaleImage`, `gluBuild1DMipmaps`, `gluBuild2DMipmaps`, `gluBuild3DMipmaps`, `gluBuild1DMipmapLevels`, `gluBuild2DMipmapLevels`, and `gluBuild3DMipmapLevels`.

GLUT

GL Utility Toolkit; API versions 1.0, 2.0, 3.0, 3.4, 3.6, and 3.7

PyOpenGL provides coverage of all GLUT API versions and the calling convention for the exposed functions are identical to that of the C binding, with a few minor exceptions.

First, the `glutInit` function takes a list of arguments (usually `sys.argv`, but not `sys.argv[1:]`) and returns all non GLUT arguments. For example

```
import sys

my_argv = glutInit(sys.argv)
```

Secondly, setting callbacks in Python is done in the same way as done in C, but Python has no `NULL` pointer so `None` is used to clear a callback instead, i.e.

```
def on_display():
    pass
```

```
glutDisplayFunc(on_display) # set the callback
glutDisplayFunc(None)       # clear the callback
```

For more information about GLUT see the Man Pages [http://pyopengl.sourceforge.net/documentation/ref/glut.html], the GLUT homepage [http://reality.sgi.com/mjk/glut3/glut3.html], or OpenGL.org's GLUT documentation [http://www.opengl.org/developers/documentation/glut.html].

WGL

Win32 OpenGL Library; API version 4

PyOpenGL has complete coverage of the WGL API, with the exception that the `GetEnhMetaFilePixelFormat` function is currently not supported.

Like the rest of PyOpenGL, the WGL module has a strict error mechanism. If a Windows error occurs during a WGL call then a `WindowsError` exception error is thrown. For Python versions less than 2.1, an `OSError` will be thrown instead.

For more information about WGL see the MSDN Library [http://msdn.microsoft.com/library/?url=/library/en-us/opengl/hh/opengl/ntopnglo_0e0o.asp?frame=true].

GLE

Tubing and Extrusion Library; API version 3

PyOpenGL supports version 3 of the GLE (Tubing and Extrusion) library. The function prototype exposed by the Python binding are identical to that of the GLE C binding, except that passing array lengths explicitly is not needed. For more information consult the docstring of each function or the Man Pages [http://pyopengl.sourceforge.net/documentation/ref/gle.html]. For information regarding GLE itself see the GLE homepage [http://linas.org/gle].

Extensions

PyOpenGL includes support for many GL, GLU and WGL extensions. These extensions are implemented as sub-modules of the `GL`, `GLU` or `WGL` package. For example, the `GL_ARB_multitexture` extension would be implemented in the module `GL.ARB.multitexture`. For every extension that PyOpenGL supports the corresponding module will exist regardless of whether or not that extension is implemented by the user's OpenGL library. A placeholder module will also exist even if the extension defines no new tokens or functions. If the extension is not supported by the current context than any attempt to use that extension will throw a `GLerror` exception with a code of `GL_INVALID_OPERATION`.

Each OpenGL implementation has its' own method for linking to OpenGL extensions. These methods vary from dynamic linking to static linking and even some dynamic/static combinations. Regardless of which case applies to a specific implementation of OpenGL, one still needs to verify that a particular extension is supported before attempting to use any functions or tokens that it declares. This is usually done by calling `glGetString(GL_EXTENSIONS)` and looking for extension name in the returned string.

Extensions are potentially context dependent, which means that an extension supported by one OpenGL context may not be supported by another. This means that verification that extension is supported and loading of the extension procedure addresses (if the OpenGL extension mechanism is dynamic) needs to be done for each context that intends to use a particular extension. PyOpenGL simplifies this somewhat by providing a single initialization function for each extension which does the following:

1. verify the extension is supported by the current context
2. load all procedure addresses for extension if needed
3. return a boolean indicating success of the previous steps

The naming scheme of the this initialization is designed to be consistent with the naming scheme used with OpenGL extensions. For instance, here is a function which takes an extension name and returns the initialization function name:

```
def init_name(extension_name):
    parts = string.split(extension_name, '_')
    return string.join([string.lower(parts[0]), 'Init'] +
                       map(string.capitalize, parts[2:]) +
                       [parts[1]], '')

>>> init_name('GL_ARB_multitexture')
'glInitMultitextureARB'

>>> init_name('GLU_SGI_filter4_parameters')
'gluInitFilter4ParametersSGI'
```

As an example of extension usage, here is a GLUT program that needs the GL_ARB_multitexture extension:

```
from OpenGL.GLUT import *
from OpenGL.GL.ARB.multitexture import *
import sys

# initialize GLUT
argv = glutInit(sys.argv)

# create a window, needs to be done before glInitMultitextureARB
glutCreateWindow('foo')

# see if GL_ARB_multitexture is supported
if not glInitMultitextureARB():
    # it's not supported...panic!
    print "Help, I'm lost without GL_ARB_multitexture!"
    sys.exit(1)

# do something with it...
```

It is important to note that the initialization function for an extension *must* be called by each context that intends to use that extension. Failure to do so will result in a `GL.GLerror` with a code of `GL_INVALID_OPERATION` even if the extension is actually supported by that context.

Writing Portable Code

For many modules PyOpenGL is designed to support multiple versions of the API exposed by that particular module. For instance, the `GLU` module supports GLU API versions 1.0, 1.1, 1.2 and 1.3. In many cases different versions of an API have different functions and constants. There are several ways for Python code to determine at runtime which API is exposed.

All PyOpenGL modules have an attribute named `__api_version__` which is set to an integer which defines the current version. Usually this a combination of the major version number in the high word and the minor in the low word. So `GLU` has

```
__api_version__ = 0x100 | 0x101 | 0x102 | 0x103
```

Currently the only exception to this is the `GLUT` module which only uses the low word to store the Xlib implementation number (see your GLUT header for information about this.)

Often times a C header which defines an API has some macro or macros which define the version number. PyOpenGL includes the same macros as module attributes to make porting C code easier. For the `GLU` module these macros are `GLU_VERSION_1_1`, `GLU_VERSION_1_2`, and `GLU_VERSION_1_3`. So if your Python code needs GLU 1.2 to run then you could test for this at runtime by doing the following:

```
from OpenGL.GLU import *

try:
    GLU_VERSION_1_2
except:
    print "Help! I'm lost without GLU 1.2!"
    sys.exit(1)
```

Chapter 4. Internals

Porting

There are three main files which contain platform specific information. The first is the configuration file used by the setup script. The following code illustrates how this file is selected

```
name = sys.platform
config_name = os.path.join('config', name + '.cfg')

while len(name) and not os.path.exists(config_name):
    name = name[:-1]
    config_name = os.path.join('config', name + '.cfg')
```

The net effect of this method is to allow the names of configuration files to proceed from fine to coarse in granularity. For instance, suppose that the "linux-alpha" platform required a special configuration file but all other Linux platforms (linux, linux1, linux2, linux-i386, etc.) used the same configuration file. To cover both of these cases one would create two configuration files: `config/linux.cfg` and `config/linux-alpha.cfg`.

This configuration file controls various build options, such as include directories or lib directories. For example, here is a verbatim copy of this file for the "linux" platform:

```
; General config options
;
; Setting build_togl to zero will avoid trying to build Togl
;
; gl_platform is the name of the platform specific OpenGL module
; For X-windows this GLX, Windows has WGL, etc.
;
; include_dirs and library_dirs are a sys.pathsep separated list of
; additional directories for headers and libraries.  No quotes
; are needed
[General]
build_togl=1
gl_platform=GLX
include_dirs=/usr/include:/usr/local/include:/usr/X11/include
library_dirs=/usr/lib:/usr/local/lib:/usr/X11/lib

; a sys.pathsep separated list of the libs needed when linking GL
[GL]
libs=GL:X11:Xext

; a sys.pathsep separated list of the libs needed when linking GLU
; the GL libraries are included automatically
[GLU]
libs=GLU

; a sys.pathsep separated list of the libs needed when linking GLUT
; the GL and GLU libraries are included automatically
```

```
[GLUT]
libs=glut:Xi:Xmu

; a comma separated list of the libs needed when linking Togl
; the GL and GLU libraries are included automatically
[Togl]
libs=Xmu:Xt:m
```

Most of the entries are explained by the embedded comments, but one is worth commenting on further. This is the "gl_platform" entry. This entry is used tell the setup script the name of the platform specific GL platform. This name is used in the `src/config.h` and the `src/interface_util/platform.c` files to include the right code for that platform. For instance in `src/config.h`

```
#if defined(WGL_PLATFORM)

#include <windows.h>

/* Do extension definitions define the C prototype
   or just the enumerants? */
#define EXT_DEFINES_PROTO 0

/* A function which returns true if the current
   context is valid */
#define CurrentContextIsValid() wglGetCurrentContext()

/* Get a hash code (long) corresponding to the
   current context */
#define GetCurrentContext() ((long)wglGetCurrentContext())

/* Does the platform have a dynamic extension loading
   mechanism? */
#define HAS_DYNAMIC_EXT 1

#elif defined(AGL_PLATFORM)

.
.
.
```

The last platform specific file is the `src/interface_util/platform.c` file. This file defines functions to retrieve platform specific extension addresses and extension strings. For more information see the source code.

Contributing

PyOpenGL uses the wrapper generator SWIG to convert wrapper descriptions (interfaces) into Python extension modules. In order to add to PyOpenGL you need SWIG 1.3a5 which available at the SWIG

homepage [http://swig.sourceforge.net]. If you haven't used SWIG before then you might consider playing with it a bit before you contemplate trying to contribute to PyOpenGL.

PyOpenGL makes extensive use of the SWIG typemap feature. Chances are that you won't have to write any complex typemaps yourself as most these have probably already been created to support the `GL` module, etc. You may have to write some simple scalar typemaps.

The Build System

You probably won't have to modify the build system to add a module unless the module needs some external library or such. To add a module just put the interface file (.i extension) in the interface directory. For example, `GL.ARB.multitexture` goes into `interface/GL/ARB/multitexture.i`.

Interface Files

Here are some lines common to every interface file (using the GLUT module as an example)

```
/*
# BUILD api_versions [1, 2, 5, 7, 9, 11, 13]
# BUILD macro_template 'GLUT_XLIB_IMPLEMENTATION >= %(api_version)d'
# BUILD headers ['GL/glut.h']
# BUILD libs ['GLUT']
*/

%module GLUT

#define __version__ "$Revision: 1.23 $"
#define __api_version__ API_VERSION
#define __doc__ "http:\057\057reality.sgi.com/mjk/glut3/glut3.html"

%include util.inc
```

First off notice that the `__doc__` define uses "\057\057" and not "//". This is because the preprocessor of SWIG is a bit buggy and sometimes won't handle double slashes.

Second, notice the inclusion of the `util.inc` file. It defines a lot of the typemaps and such needed by PyOpenGL modules, chances are you'll need it.

PyOpenGL has the ability to support multiple versions of an API. This is accomplished by this line

```
# BUILD api_versions [1, 2, 5, 7, 9, 11, 13]
```

For each number listed in this list the build system will run SWIG with the macro API_VERSION defined to be the API version to be generated. So suppose that I want to put in a feature that is only supported in version 11 of the API. Then the function prototype in the interface would look like this:

```
#if API_VERSION >= 11
void foo(int bar);
#endif
```

Note that the SWIG preprocessor also doesn't like comparisons between hexadecimal numbers so don't use this

```
#ifdef API_VERSION >= 0xb
```

When the generated C code for the interface is compiled it needs to pick out the right code for API version that is supported. Usually the API version is defined by some macro in the header file which defines the API. For instance, GLUT has the macro GLUT_XLIB_IMPLEMENTATION. In order to test this macro first the build system needs to know what header the macro is located in. This is accomplished by the

```
# BUILD headers ['GL/glut.h']
```

line. The build system also needs to know how to make a comparison on the macro. This is accomplished by the

```
# BUILD macro_template 'GLUT_XLIB_IMPLEMENTATION >= %(api_version)d'
```

line. The string defined by the macro template line should be a Python format string. The substitution names allowed are `api_version_underscore` which splits the API_VERSION into high word and low word separated by an underscore, and `api_version` which is just API_VERSION as an integer. `api_version_underscore` is useful for macros like `GL_VERSION_1_1`.

If the API that you are wrapping defines structures or classes which need to be exposed than you need to use the "shadow" feature of SWIG. I won't go into all the details of how this feature works here. To see an example of its use look at the WGL module. To turn on shadowing use this line:

```
# BUILD shadow 1
```

There are other "BUILD" headers allowed by PyOpenGL. Here are some examples:

- To only build the module on GLX platforms:
  ```
  # BUILD gl_platforms ['GLX']
  ```
- To specify what libraries the module needs:
  ```
  # BUILD libs = ['GLUT']
  ```

  Note that the names in the in the libs header are platform independent names. The setup script transforms these into a platform specific name or names. For example on Win32, GLUT = glut32.

Docstrings

One nice feature of Python is the docstring feature. SWIG does not support this natively, but PyOpenGL has a workaround to enable their use. To add a docstring to a function call "foo" just include in the interface the line:

```
DOC(foo, "foo() -> bar")
```

Exception Handlers

PyOpenGL doesn't expose the `glGetError` function, instead it throws a `GL.GLerror` exception. To turn on the GL exception handler you need a line like this:

```
GL_EXCEPTION_HANDLER()
```

Some GL commands are designed to run between a `glBegin` and a `glEnd`. For these it is not possible to call `glGetError`, so before these functions you need this:

```
EXCEPTION_HANDLER()
```

To restore the ordinary exception handler just call `GL_EXCEPTION_HANDLER` again.

WGL, AGL, etc. need their own OS specific exception handler, so you'll need use the SWIG **%except** command. Look at the `WGL` module for an example.

Appendix A. Changes

2.0.0.44

1. Fixed PyMem_Del -> PyObject_Del bug; Patch #455646.
2. Made `glDrawRangeElementsEXT` more like other array functions including adding decoration variants
3. Added some aliases for various extension initialization functions.
4. Added the GL_OML_subsample, GL_OML_resample, and GL_OML_interlace extensions
5. Fixed Numeric import bug in the interface_util library.
6. Added quaternion class.
7. Changed comments over to C style.
8. Fixed install bug which prevented installation of demos.

2.0.0.42beta

1. Added GL 1.0 compatibility.
2. Moved `info.py` into the new scripts directory
3. Added a "__build__" attribute to `OpenGL/__init__.py`
4. Added the GL_EXT_texture_object, GL_EXT_polygon_offset, and the GL_EXT_vertex_array extensions
5. Made `glDrawElements` more like other array functions including adding decoration variants

2.0.0.40beta

1. Fixed the export name of `gluPwlCurve` and `gluNurbsCurve`
2. Fixed mishandling of count parameter in `gluPwlCurve` and changed the prototype to `gluPwlCurve(nobj, points, type)`.
3. Changed proto of `gluNurbsCurve` to `gluNurbsCurve(nobj, knot, ctlarray, type) -> None`
4. Changed prototype of `gluNurbsSurface` to `gluNurbsSurface(nobj, sknot, tknot, ctlarray, type) -> None`
5. Added initialization functions to extension modules.
6. Added the platform specific source `platform.c`
7. Killed the `OpenGL.extensions` module, no longer needed.
8. Added a `trackball` module.

2.0.0.34b3

1. Fixed the export name of `gluTessVertex`, was `_gluTessVertex`.
2. Fixed misuse of `PyObject_CallFunction` which resulted in wrong number of arguments passed when only one argument was being passed to a callback. Affected several functions in the GLU and GLUT modules.
3. Added locking mechanism to `GLUtesselator` and `GLUnurbs` to prevent user data from being collected by the garbage collector.
4. Fixed prototype of `gluUnProject4`, was missing *`clipW`* argument.
5. Fixed prototypes of `gluBuild1DMipmapLevels`, was missing some arguments.

6. Fixed `gluNurbsCallbackData wrapper`, wrong object pointer cast.
7. Changed list separator in config files to `os.pathsep` instead of a comma.
8. Fixed bug in `PyObject_Dimension` on non-Numeric arrays.
9. Fixed docstrings of `glTexImage2D`
10. `setup.py` now adds the current directory to the search path. Needed since **build_py** has to execute various configure programs.

2.0.0.27b2

1. When testing for a `GLerror` an attempt is now made to verify that the current context is valid. This avoids strange behavior like `glutDestroyWindow` throwing `GLerror` when it is used to destroy the current window. Also in 2.0b1 a maximum limit of 16 errors could be return by `GLerror`. This was done to avoid infinite loop result resulting from calls to `glGetError` without a valid context. This is not needed anymore, so the hard-coded limit has been removed.
2. Fixed `sys.argv` bugs in knot and molehill demos.
3. Added short name aliases like `glTranslate` = `glTranslated`.
4. Fixed `glGetBooleanv`, now needs only one argument.
5. Added build number to setup.
6. The current version number of the entire PyOpenGL system is now stored in `__version__`.
7. Fixed various Python 1.5/1.6 incompatibilities.
8. Fixed image routines which return packed images like `glGetPolygonStipple`.
9. Added `glGetPolgonStippleub` and `glPolygonStippleub` to use arrays instead of packed strings.
10. Fixed bugs in `glMap{1|2}{f|d}` in the automatic selection of *`ustride`* and *`vstride`* arguments.
11. Fixed `glRenderMode` and `glSelectBuffer` to avoid the strange behavior of `glSelectBuffer(0, NULL)` on some systems.
12. `glutInit` now raises TypeError if its single argument is not a non-string sequence.
13. Removed the GL_EXT_polygon_offset extension since it is a standard part of OpenGL 1.1 and some systems seem to define GL_EXT_polygon_offset yet don't export `glPolygonOffsetEXT`.
14. Fixed bug in `glGetBooleanv` which caused it to return the wrong number of values, i.e. not parameter specific.
15. Fixed a bug in WGL routines in which the previous error wasn't cleared.
16. Added support for GLU 1.0 and 1.1 to support Mesa.
17. Added the `info` script.
18. Added new platform specific config files.
19. Fixed broken prototypes in `GL.EXT.coordinate_frame`, `GL.EXT.cull_vertex`, `GL.EXT.multi_draw_arrays` and `GL.EXT.fog_coord`.
20. Changed the generation of shadow wrappers a bit and turned the `GL` and `GLU` modules into shadow wrappers.

2.0b1

1. `WGL` modules now throw `WindowsError` (`OSError` for Python 2.0) on a windows error.
2. Added `GLerror` to `GL` and `GLUerror` to `GLU`. `"except GLerror, e:"` now works.
3. `GLerror` now handles multiple errors resulting from distributed systems.

2.0a4

1. Made the viewport argument to `gluPickMatrix` a keyword.
2. Added a `glSelectWithCallback` function. This function is pretty trivial since `glRenderMode` already returns a selection buffer. See the release notes.
3. Made the order arguments to `glMap{1|2}{f|v}v` automatic and removed the stride arguments.
4. Changed `GLexception` and `GLUexception` to `GLerror` and `GLUerror`.
5. Killed the `RTS` module.
6. Killed the `GL.Autodesk.facet_arrays` module. Doesn't look as though it is possible to support it using the current memory management model.
7. `glRenderMode` now returns the instance of `glFeedbackBuffer` or `glSelectBuffer` instead of a copy. It also resets the GL's pointer to the buffer to `NULL` after it's done.
8. Fixed build bug in which modules with the same name like `WGL.ARB.extensions_string` and `WGL.EXT.extensions_string` had conflicting obj names (really a work around for a distutils bug.)
9. Fixed `WGL.EXT.extensions_string` prototype and reference in `extensions.py`
10. Removed `checkExtension` and `raiseExtension`.
11. Made considerable changes to the `extensions` module.
12. `GLerror` with a code of `GL_INVALID_OPERATION` is now thrown on an attempt use an unsupported extension. This is done to match the OpenGL 1.2 behavior of GL_ARB_imaging.

2.0a3

1. Improved font and menu code of `GLUT`.
2. Added the `GLE` module.
3. Fixed exported name of `gluNurbsSurface`, was `_gluNurbsSurface`.
4. Added `RTS` module and `hello2rts` demo.
5. The array code of PyOpenGL now works even if Numeric support was compiled in, but Numeric is not found at runtime.
6. A `NotImplementedError` is thrown if an extension is not available even on systems that use static extensions. If the extension is available and `glGetProcAddress` fails then an `ImportError` is thrown.
7. Fixed Joystick callback in GLUT 4.0 code (*`poll_interval`* parameter was missing.)
8. Added support for preliminary GLUT 4.0 code. It's inclusion depends on the macro `GLUT_XLIB_IMPLEMENTATION` not on the macro `GLUT_API_VERSION` since `GLUT_API_VERSION` does not include the minor version number.
9. Added `glTexSubImage1D{ub|b|us|s|ui|i|f|d}` and `glTexSubImage2D{ub|b|us|s|ui|i|f|d}`
10. Added support for 3D and 4D texturing.
11. Added GLU 1.3 support.
12. Added lots more extensions (the running tally is now at 165.)

2.0a2

1. Changed the base class of `GLUexception` to `EnvironmentError` instead of `Exception`
2. Added support for `glInterleavedArrays`.
3. Removed nonsense variants of `glVertexPointer`, etc. such as `glVertexPointerub`. No corresponding GL calls exist.
4. Added collection of convenience functions (`glVertexd`, `glVertexf`, etc.) to avoid silly calls like `glVertex3f([0, 1, 2])` or `glColor4f(0, 1, 2, 3)`

5. Changed the exception thrown on an attempt to use an unsupported extension to `NotImplementedError` instead of `ImportError`.
6. Added the module `extensions` which does various useful things like getting the list of implemented or supported extensions.
7. Added extension support for `GLU`.
8. Removed `glGetError` since any `GL` error will now throw an instance of `GLexception` which is derived from `EnvironmentError`.
9. Added multi-window support to `GLUT`. Window callbacks are now window specific as per the specification.
10. Added `gluScaleImagef`, `glScaleImagei`, etc. to `GLU`.
11. Changed implementation of `GLUnurbs` and `GLUquadric` to match that of `GLUtesselator` to prepare for addition of GLU 1.3 support.
12. The model view matrix, projection matrix, and viewport arguments to `gluProject` and `gluUnProject` are now optional.
13. Added lots more extensions....

Appendix B. Licenses

Forward Note: This forward is for informational purposes only.

The PyOpenGL project on SourceForge was started by David Ascher, previous maintainer of the PyOpenGL package, to further development of the binding while David was unable to continue such development. At the time of the project's start, PyOpenGL was version 1.5.5. You can find this version of the package on the downloads page of the project.

Because of this history, there are two licenses which pertain to the library, both of which are "BSD Style" licenses (though with minor variations in wording).

For all versions up to and including 1.5.5, the PyOpenGL 1.5.5 License is applicable.

For all versions greater than 1.5.5, both the PyOpenGL 1.5.5 License and the PyOpenGL 1.5.6 License are applicable.

For all versions of PyOpenGL greater than or equal to 2.0 the following licenses also apply

- the GLE License applies to the GLE module
- the SGI Free Software License B [http://oss.sgi.com/projects/FreeB] applies to GL and GLU modules
- the PyGLUT License applies to the GLUT module since PyOpenGL's GLUT module is a derivative of PyGLUT.

PyOpenGL 1.5.5 License

Copyright © 1997-1998 by James Hugunin, Cambridge MA, USA, Thomas Schwaller, Munich, Germany and David Ascher, San Francisco CA, USA.

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the names of James Hugunin, Thomas Schwaller, or David Ascher not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

JAMES HUGUNIN, THOMAS SCHWALLER AND DAVID ASCHER DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL JAMES HUGUNIN, THOMAS SCHWALLER AND DAVID ASCHER BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

PyOpenGL 1.5.6 License

Appendix B. Licenses

PyOpenGL 1.5.6 is based on PyOpenGL 1.5.5, Copyright © 1997-1998 by James Hugunin, Cambridge MA, USA, Thomas Schwaller, Munich, Germany and David Ascher, San Francisco CA, USA.

Contributors to the PyOpenGL project in addition to those listed above include:

- David Konerding
- Soren Renner
- Rene Liebscher
- Randall Hopper
- Michael Fletcher
- Thomas Malik
- Thomas Hamelryck
- Jack Jansen
- Michel Sanner

Contributors to the PyOpenGL 2 project in addition to those listed above include:

- Tarn Weisner Burton
- Andrew Cox

PyOpenGL 1.5.6 Copyright © 1997-1998, 2000-2001

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The names of the contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

GLE License

SOFTWARE AGREEMENT

PLEASE READ THIS AGREEMENT CAREFULLY BEFORE INSTALLING OR USING THIS SOFTWARE. IF YOU INSTALL OR USE THIS SOFTWARE, YOU AGREE TO THESE TERMS.

This software is owned by International Business Machines Corporation ("IBM"), or its subsidiaries or IBM’s suppliers, and is copyrighted and licensed, not sold. IBM retains title to the software, and grants you a nonexclusive license for the software.

Under this license, you may:

1. use the software on one or more machines at a time;
2. make copies of the software for use or backup purposes within your enterprise;
3. modify this software and merge it with another program; and
4. make copies of the original file you downloaded and distribute it, provided that you transfer a copy of this license to the other party.

The other party agrees to these terms by its first use of this software.

You must reproduce the copyright notice and any other legend of ownership on each copy or partial copy of the software.

This software, as provided by IBM, is only intended to assist in the development of a working software program. The software may not function as written: additional code is required. In addition, the software may not compile and/or bind successfully as written.

IBM PROVIDES THE SOFTWARE "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE SOFTWARE IS WITH YOU. SHOULD ANY PART OF THE SOFTWARE PROVE DEFECTIVE, YOU ASSUME THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. IN NO EVENT, UNLESS REQUIRED BY APPLICABLE LAW, SHALL IBM BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE SOFTWARE TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF IBM HAS BEEN ADVISED IN ADVANCE OF THE POSSIBILITY OF SUCH DAMAGES.

IBM does not warrant that the contents of the software will meet your requirements, that the software is error-free or that the software does not infringe on any intellectual property rights of any third party.

IBM may make improvements and/or changes in the software at any time.

Changes may or may not be made periodically to the information in the software; these changes may be reported, for the software included herein, in new editions.

References, if any, in the software to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in the software is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used.

The laws of New York State govern this agreement.

PyGLUT License

PyGLUT file is derived from the glut.h distributed with GLUT 3.7. It is a complete wrapper for GLUT API version 4 (provisional) including the game functionality.

The conversion to a SWIG interface file was done by Andrew Cox [mailto:acox@globalnet.co.uk].

I (Andrew Cox) place no additional limitations on what can be done with the contents of this file beyond those it inherits from GLUT.

DISCLAIMER: PyGlut is provided AS IS without warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. In no event shall Andrew Cox be liable for any damages whatsoever including direct, indirect, incidental, consequential, loss of business profits or special damages, even if Andrew Cox has been advised of the possibility of such damages.

Copyright © Mark J. Kilgard, 1994, 1995, 1996, 1998.

This program is freely distributable without licensing fees and is provided without guarantee or warrantee expressed or implied. This program is -not- in the public domain.